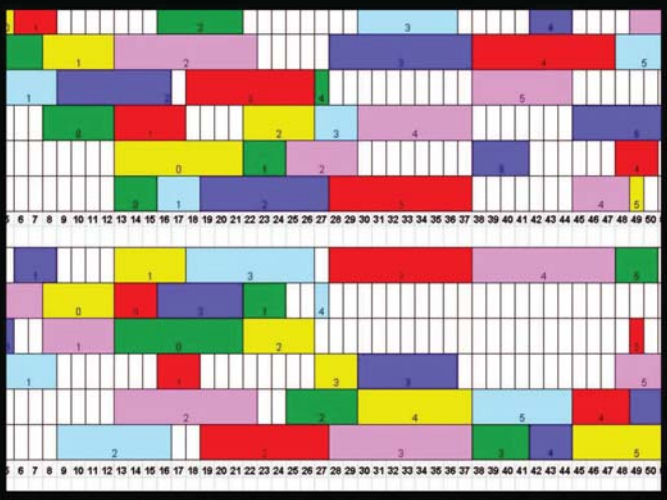


Antoni Niederliński

Programowanie w logice z ograniczeniami

Łagodne wprowadzenie dla platformy ECL^{i}PS^e

Wydanie 3 poprawione, rozszerzone i uściślone

Programowanie w logice z ograniczeniami
Łagodne wprowadzenie dla platformy ECL^{i}PS^e
Wydanie 3 poprawione, rozszerzone i uściślone
Antoni Niederliński

Antoni Niederliński

Programowanie w logice z ograniczeniami

Łagodne wprowadzenie
dla platformy ECL^{i}PS^{e}

Wydanie 3 poprawione, rozszerzone i uściślone

pkjs.com.pl

Gliwice 2014

Antoni Niederliński, 1937 -
Katedra Inżynierii Wiedzy
Wydział Informatyki i Komunikacji
Uniwersytet Ekonomiczny w Katowicach
ul. Bogucicka 3
40-226 Katowice
e-mail: antoni.niederlinski@ae.katowice.pl

Copyright © Antoni Niederliński

Wydrukowano z plików PDF dostarczonych przez autora

Książka ani żadna jej część nie może być drukowana
bez zgody autora. PDF książki, dostępny na witrynie
http://www.pwlzo.pl, może być powielany, przechowywany
w systemach komputerowych i rozpowszechniany bez zgody autora.

ISBN 978-83-60716-95-3

Wydanie 1, Gliwice 2010, PKJS
Wydanie 2 poprawione, rozszerzone i uściślone, Gliwice 2012, PKJS
Wydanie 3 poprawione, rozszerzone i uściślone, Gliwice 2014, PKJS

Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego
ul. Pszczyńska 44, 44-100 Gliwice
tel. +48 32 7750086, 506132960
Gliwice 2014

"Słodkie są pożytki z przeciwności!"
William Shakespeare (1564-1616), *"Jak wam się podoba"*

"Wszelkie ograniczenia uszczęśliwiają."
Arthur Schopenhauer (1788-1860), *"Parerga i paralipomena"*

"Na cóż zdają się książki bez obrazków i historyjek?"
Lewis Carroll (1832-1898), *"Alicja w Krainie Czarów"*

Trzech przyjaciół, Polityk, Lekarz i Matematyk, wybrało się na wycieczkę w piękne górskie tereny Beskidu Śląskiego. Wychodząc z lasu zobaczyli w oddali łakę, na której pasła się czarna owca. Polityk skomentował to tak: "W Beskidzie Śląskim wszystkie owce są czarne." Na to Lekarz: "Chyba nie masz racji, w Beskidzie Śląskim zapewne tylko niektóre owce są czarne." A Matematyk, po chwili zastanowienia się, powiedział: "W Beskidzie Śląskim istnieje co najmniej jedna łąka, na której pasie się co najmniej jedna owca, która jest czarna co najmniej z jednej strony."
Anonim, *"Matematyczno-śląski folklor"*

Spis treści

Spis rysunków

Spis tabel

Przedmowa

0.1 O czym jest książka?

Książka jest elementarnym i bezbolesnym wprowadzeniem do ciekawej technologii informatycznej zwanej *programowaniem w logice z ograniczeniami* i znanej pod angielska nazwą *Constraint Logic Programming*, w dalszym ciągu określanej skrótem *CLP*. Celem książki jest nauczenie modelowania i rozwiązywania problemów decyzyjnych za pomocą *CLP*. Jest przeznaczona dla wszystkich zainteresowanych szybkim wyznaczeniem rozwiązań dopuszczalnych i optymalnych dla kombinatorycznych i ciągłych problemów decyzyjnych za pomocą wypróbowanego narzędzia. Książka stwarza również okazję nabycia pewnego praktycznego doświadczenia w stosowaniu *CLP* tym wszystkim, którzy zamierzają zająć się trudnymi i ekscytującymi problemami teoretycznymi, leżącymi u podstaw *CLP*. Dlatego:

- książka zaczyna się wprowadzeniem do poprzednika *CLP*, którym jest język *Prolog*. W *Prolog*u po raz pierwszy zastosowano przełomowe idee deklaratywnego programowania stosującego logikę dla opisu rozwiązywanego problemu. Idee te zostały rozwinięte i udoskonalone w językach *CLP*;

- książka zawiera ciąg obszernie komentowanych przykładów o wzrastającym stopniu trudności. Autor uważa bowiem, że - w odniesieniu do problematyki książki - *gram zastosowań jest wart tyle,* co *tona abstrakcji*[1]. Autor uważa, że najlepszym sposobem opanowania trudnych abstrakcji (których jest pełno w *Prolog*u i w *CLP*) to widzieć i praktykować ich zastosowania dla konkretnych przykładów[2]. Przykłady bowiem - szczególnie

[1] Nazywa się to czasami Prawem Booker'a.
[2] "Lepszy przykład niźli wykład!" - tego zdania był Jan Sztaudynger.

w dyscyplinie nasyconej logiką, a taką jest *CLP* - są bardziej zrozumiałe i lepiej przyswajalne przez początkujących aniżeli teorie;

- książka przedstawia podstawowe idee i metody *CLP* z naciskiem na *intuicyjne*, a nie *teoretyczne* zrozumienie. Jest rzeczą oczywistą, że nie każda osoba interesująca się *CLP* ma zamiar robić z *CLP* doktorat. Większość jest zainteresowana tym, co można zrobić z *CLP* i w jaki sposób. Właśnie dla tych osób książka ta jest przeznaczona. Aczkolwiek kandydaci do doktoratów z *CLP* mogą z pożytkiem przeczytać szereg rozdziałów książki przed zatopieniem się w lekturze zaawansowanych tekstów matematycznych;

- wszystkie przykłady z książki można uruchomić dla jednej z najbardziej popularnych i intensywnie wspieranych platform *CLP*, którą jest:

$$ECL^iPS^e\ Constraint\ Programming\ System\ \ (ECL^iPS^e\ CPS)$$

(zob. [ECLiPSe-12]), darmowo dostępna w ramach *Cisco-style Mozilla Public License* z witryny `http://www.eclipseclp.org/`.

Książka została zainspirowana serią wykładów i projektów z problematyki prologowej i clp-owej, prowadzonych w latach 1984-2008 na Wydziale Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach, oraz w latach 2009-2013 na Wydziale Informatyki i Komunikacji Uniwersytetu Ekonomicznego w Katowicach. Pierwsza seria zajęć była prowadzona w oparciu o platformy programistyczne *Visual Prolog* i *CHIP*, druga - w oparciu o ECL^iPS^e.

Z doświadczenia autora jako nauczyciela akademickiego wynika, że istotną trudnością napotykaną przez osoby uczące się *Prolog*u i *CLP* jest *modelowanie*, tzn. przetłumaczenie werbalnego sformułowania problemu na programy prologowe lub clp-owe. Trudność tę można przezwyciężyć rozwiązując serię różnorodnych zadań tekstowych o stopniowanej trudności. Autor ma nadzieję, że zadania tekstowe zamieszczone w tej książce, zarówno rozwiązane, jak i nierozwiązane, są właśnie zadaniami wiodącymi do prologowych i clp-owych kompetencji. Istotą książki są więc zadania; większość jej treści poświęcono na ich prezentację, analizę, przedstawienie rozwiązujących je programów i przedstawienie ich rozwiązań. Rozwiązania te korzystają z reguły z mocnych "czarnych skrzynek" w postaci *standardowych predykatów* (inaczej: *wbudowanych predykatów*) ECL^iPS^e: mają one precyzyjnie definiowane właściwości, użytkownik zawsze wie, czym je karmić i co zostanie mu zwrócone. Jednakże mechanizm ich

działania - stanowiący część pominiętej w książce teorii - jest przed użytkownikiem zakryty. Zainteresowani mogą znaleźć szczegóły w szeregu bardzo dobrych monografii, np. [Apt-03], [Apt-07], [Bratko-01], [Dechter-03], [Jaffar-94], [Marriott-98], [Rossi-06]. Godne polecenia są również witryny i prezentacje internetowe [Bartak-10] i [Simonis-10].

Sztukę tłumaczenia problemów werbalnych na programy prologowe lub clp-owe można w sposób przyjemny i bezbolesny opanować rozwiązując różnego rodzaju łamigłówki. Rozwiązywanie łamigłówek jest nie tylko doskonałym sposobem opanowania sztuki modelowania. Autor zgadza się z poglądem przedstawianym przez Michalewicza (patrz [Michalewicz-07] i [Michalewicz-08]):

1. Łamigłówki mają wartość edukacyjną, gdyż pokazują szereg użytecznych i mocnych technik rozwiązywania problemów w zabawny sposób.

2. Łamigłówki wciągają i skłaniają do myślenia.

3. Cały szereg użytecznych technik programistycznych (jak np. symulacja, optymalizacja) oraz szereg dziedzin zastosowań (np. w biznesie, zarządzaniu, inżynierii przemysłowej, finansach) można przedstawić korzystając z prostych łamigłówek.

Co ważniejsze, szereg istotnych kombinatorycznych problemów biznesowych, zarządzania i inżynierii przemysłowej (jak alokacja resursów, tworzenie harmonogramów, uniwersyteckich rozkładów zajęć i rozkładów egzaminów, marszrutyzacja pojazdów, planowanie inwestycji i dużo dużo innych) to nic innego jak właśnie *mega-łamigłówki* lub wręcz *tera-łamigłówki* z przeogromnymi liczbami zmiennych; aby uzyskać solidną podstawę dla rozwiązywania takich problemów w oparciu o techniki clp-owe, dobrze jest zacząć uczyć się *clp-owego myślenia* na przykładzie serii *mikro-łamigłówek*

I jeszcze ostatni - lecz równie ważny argument: obecnie podręcznik dla studentów musi współzawodniczyć o ich uwagę z szeregiem dystrakcji: Internet, gry komputerowe, portale społecznościowe, by wymienić tylko najważniejsze. By zupełnie nie odpaść w tej konkurencji, podręcznik nie powinien być nudny. Od dobrego wykładowcy oczekuje się, że od czasu do czasu powie coś niezwykłego, coś paradoksalnego, jakiś dowcip, aby powstrzymać studentów przed zaśnięciem i aktywować ich umysły. Autor uważa, że to samo tyczy się podręczników: podręczniki nie powinny być nudne. To właśnie łamigłówki są doskonałym sposobem wprowadzania chwilowego relaksu do długotrwałego umysłowego wysiłku.

0.2 Co zawiera książka?

Treść książki jest zawarta w siedmiu rozdziałach.
Rozdział 1 (*Przedmowa*) jest wprowadzeniem do zasadniczych idei leżących u podstaw *Prologu* i *CLP*, omawia założenia poczynione dla napisania książki, krótko charakteryzuje zawartość rozdziałów..
Rozdział 2 (*Na początku był Prolog*) przedstawia *Prolog*, który jest ojcem wszystkich języków *CLP*. *Prolog* był pierwszym językiem umożliwiającym pisanie programów zawierających *wyłącznie* wiedzę o rozwiązywanym problemie i deklarujących cel, który należy osiągnąć, abstrahując jednocześnie od algorytmicznego mechanizmu przetwarzania zadeklarowanej wiedzy. W rozdziale tym położono szczególny nacisk na idee, które później zostały rozwinięte i poszerzone w językach *CLP*, lecz które łatwiej zrozumieć w prostszych ramach języka *Prolog*. Przedstawione w tym rozdziale przykłady należą (podobnie jak wszystkie dalsze przykłady w rozdziałach poświęconych *CLP*) do następujących czterech kategorii, patrz rozdział 1.7:

1. Przykłady wyznaczania *dopuszczalnego stanu* (DS) w przestrzeni stanu problemu, obejmujące m. in. zestaw przykładów konfigurowania systemu trój-elementowego spełniającego pewne wymagania odnośnie do kompatybilności elementów i kosztu całości. Przykłady te przedstawiają czytelnikowi podstawowe mechanizmy *poszukiwań w drzewach* i *propagacji ograniczeń* na drodze unifikacji. Poszukiwania w drzewach są na razie ograniczone do poszukiwań metodą przeglądu zupełnego i metodą standardowych nawrotów. Propagacja ograniczeń jest na razie ograniczona do unifikacji.

2. Przykłady wyznaczania *optymalnego stanu* (OS) w przestrzeni stanu problemu, tzn. wyznaczania takiego stanu dopuszczalnego, który optymalizuje określony *wskaźnik jakości*, przedstawiający koszt, zysk lub straty. Ilustruje to przykład wyznaczania najtańszej konfiguracji systemu trój-elementowego, rozwiązany za pomocą poszukiwań metodą *gałęzi i ograniczeń*.

3. Przykłady wyznaczania *dopuszczalnej trajektorii stanów* (DTS) w przestrzeni stanu problemu, tzn. wyznaczania ciągu stanów dopuszczalnych od znanego *stanu początkowego* do znanego *stanu końcowego*. Ilustruje to popularny przykład wież z Hanoi.

4. Przykłady wyznaczania *optymalnej trajektorii stanów* (OTS) w przestrzeni stanu problemu, tzn. wyznaczania takiego ciągu dopuszczalnych stanów, który optymalizuje określony *wskaźnik jakości*. Przykładami tymi

są przykłady znajdowania drogi w labiryncie, przykłady przepraw przez rzekę, przykład napełniania zbiorników.

Wszystkie przykłady z rozdziału 2 korzystają z *Prolog*u zaimplementowanego na platformie ECL^iPS^e i zwanego w dalszym ciągu ECL^iPS^e *Prolog*. Programy prologowe mają rozszerzenie *.pl*. W trakcie ich kompilowania ECL^iPS^e *Prolog* korzysta tylko z mechanizmów i standardowych predykatów *Prolog*u.

Czytelnik obeznany z materią *CLP* może w tym miejscu zadać zasadne pytanie: *Dlaczego rozpoczynać książkę poświęconą CLP rozważaniami i przykładami stosującymi "słabszy" Prolog, dlaczego nie przejść od razu do "silniejszego" CLP? Tym bardziej, że mechanizm poszukiwań w Prologu (standardowe nawroty z propagacja ograniczeń na drodze unifikacji) jest różny od mechanizmu poszukiwań w CLP (bardziej efektywne nawroty i propagacja ograniczeń za pomocą technik zgodnościowych)*. Odpowiedź na to pytanie wynika tylko z względów natury dydaktycznej, a mianowicie z potrzeby stopniowania trudności wykładu[3]:

1. *Prolog* jest koncepcyjnie prostszy od *CLP*, np. w odniesieniu do właściwości *deklaratywności*. *Prolog* zawiera wszystkie podstawowe mechanizmy wnioskowania stosowane w *CLP*, lecz w postaci prostszej, bardziej przejrzystej i łatwiejszej do zrozumienia: jedyną (standardową) strategią poruszania się w drzewach poszukiwań jest w *Prolog*u strategia poszukiwań w głąb ze standardowymi nawrotami, jedyną metodą propagacji ograniczeń jest unifikacja termów.

2. Podstawowe elementy programu prologowego są takie same jak programu clp-owego.

3. W *Prolog*u możliwe jest również stosowanie *przeglądu zupełnego*, co pokazano dla przykładów *konfigurowania* i *N hetmanów*. Przegląd zupełny ma zastosowanie tylko dla bardzo prostych (małych) przykładów, jednak jego znaczenie pojęciowe jest doniosłe: umożliwia definiowanie *pełnego drzewa poszukiwań* i ułatwia rozumienie bardziej efektywnych metod poszukiwań stosowanych w *CLP*. Metody te można uważać za modyfikacje przeglądu zupełnego.

4. Wreszcie niebagatelnym argumentem jest to, że ECL^iPS^e dopuszcza programowanie w czystym *Prolog*u.

[3] Zaczynaj od prostych problemów, opanuj ich rozwiązywanie, przejdź stopniowo do bardziej złożonych.

Szereg przykładów rozwiązywanych w rozdziale 2 w *Prolog*u jest ponownie rozwiązywanych w późniejszych rozdziałach z zastosowaniem *CLP*. Ma to na celu lepsze unaocznienie różnic w podejściu i lepsze zobrazowanie większej efektywności *CLP* w porównaniu z *Prolog*iem.

Wszystkie przykłady z rozdziału 3 i rozdziałów następnych korzystają wyłącznie z platformy ECL^iPS^e *CLP*. Odpowiednie programy będą mieć rozszerzenie *.ecl*. Kompilacja dowolnego programu z rozszerzeniem *.ecl* sprawia, że ECL^iPS^e kompiluje go z zastosowaniem tych mechanizmów i standardowych predykatów, które znajdują się w bibliotekach *CLP* zadeklarowanych na początku programu. Dla każdego predykatu standardowego jest - w dokumentacji systemu ECL^iPS^e - wymieniona biblioteka, w której predykat ten jest zdefiniowany.

Zagadnienia omawiane w rozdziałach 3,...,6 można podzielić na następujące rozłączne kategorie:

1. Kategorie *celu programu*:
   - celem programu jest wyznaczenie *rozwiązania dopuszczalnego*, którym może być *dopuszczalny stan* (DS) lub *dopuszczalna trajektoria stanu* (DTS);
   - celem programu jest wyznaczenie *rozwiązania optymalnego*, którym może być *optymalny stan* (OS) lub *optymalna trajektoria stanu* (OTS).

2. Kategorie stosowanych *predykatów standardowych*:
   - stosowane są wyłącznie *elementarne predykaty standardowe*;
   - stosowane są również *globalne predykaty standardowe*.

Niezależność wymienionych dwóch kategorii jest uzasadnieniem obecności następujących czterech rozdziałów:

1. Rozdział 3 (*CLP z predykatami elementarnymi dla rozwiązań dopuszczalnych*) rozpoczyna się wprowadzeniem kluczowego dla *CLP* pojęcia *dziedziny zmiennych* i przedstawieniem najważniejszych różnic pomiędzy *Prolog*iem a *CLP*. Istotą tych różnic są bardziej efektywne (aniżeli poszukiwania w głąb ze standardowymi nawrotami) strategie poruszania się w drzewach poszukiwań oraz bardziej efektywna *propagacja ograniczeń*, nie ograniczająca się do dziedziny termów. Predykaty elementarne wymienione w

tytule rozdziału są najbardziej podstawowymi predykatami standardowymi, definiowanymi przez konstruktorów platformy ECL^iPS^e. Przedstawiono najpierw grupę przykładów, których rozwiązanie jest możliwe tylko za pomocą propagacji ograniczeń. Następnie przedstawiono grupę przykładów, dla których propagację ograniczeń trzeba wspomóc poszukiwaniami. Przykłady te w dużej części pokrywają się z przykładami rozwiązywanymi w rozdziale 1; są tu jednak również przykłady nowe, których rozwiązywanie za pomocą *Prolog*u - jeżeli w ogóle możliwe - byłoby bardzo kłopotliwe. W rozdziale 3 nie są już spotykane clp-owe wersje przykładów rozwiązywanych w rozdziale 2 metodą przeglądu zupełnego. Stosowanie przeglądu zupełnego w *CLP* jest bowiem *merytorycznie bezzasadne*: *CLP* dysponuje znacznie bardziej efektywnymi metodami poszukiwań. Wszystkie przykłady tego rozdziału zostały rozwiązane w sposób typowy dla paradygmatu *CLP*, tzn. poczynając od definicji dziedzin zmiennych i zapisu wszystkich ograniczeń za pomocą zmiennych z tych dziedzin.

2. Rozdział 4 (*CLP z predykatami globalnymi dla rozwiązań dopuszczalnych*) rozpoczyna się definicjami dwóch bardzo popularnych predykatów globalnych, `alldifferent/1` i `element/3`. Predykaty globalne predykatami definiowanymi przez konstruktorów platformy ECL^iPS^e dla list o liczbie elementów teoretycznie dowolnej. Predykaty te są - bez przesady - źródłem mocy obliczeniowej programów we wszelkich językach *CLP*. Obszerne omówienie wszystkich spotykanych predykatów globalnych można znaleźć we witrynie "Global Constraint Catalog" (patrz [Baldiceanu-10]). Nie wszystkie predykaty globalne z tego zestawienia są stosowane na platformie ECL^iPS^e. Predykaty globalne `alldifferent/1` i `element/3` zostaną kolejno zastosowane do modyfikacji i poszerzenia pewnych przykładów rozwiązanych już w rozdziale 3. W rozdziale tym omówiono również ważniejsze struktury danych oraz techniki obliczeń iteracyjnych stosowane na platformie ECL^iPS^eCLP.

3. Rozdział 5 (*CLP z predykatami elementarnymi dla rozwiązań optymalnych*) rozpoczyna się rozważaniami na temat metody *gałęzi i ograniczeń* (ang. *branch-and -bound*) i możliwości udoskonalenia nawrotów stosowanych w tej metodzie. Zostało to zilustrowane przykładem z hetmanami. W dalszym ciągu przedstawiono technikę deklarowania problemu optymalizacyjnego za pomocą najbardziej podstawowych predykatów z platformy ECL^iPS^e CLP. Pierwszym z nich jest predykat `bb_min/3` realizujący poszukiwania metodą *gałęzi i ograniczeń* oraz predykat `search/6` umożliwiający parametryzację poszukiwań tą metodą.

4. Rozdział 6 (*CLP z predykatami globalnymi dla rozwiązań optymalnych*) omawia dwa ważne ograniczenia globalne `cumulative/4` i `disjunctive/2` w zastosowaniu do rozwiązywania szeregu różnych zadań szeregowania i harmonogramowania. Rozdział zawiera reprezentatywny wachlarz problemów harmonogramowania, od elementarnych poczynając, poprzez słynny benchmark MT10, a na problemie komiwojażera kończąc.

Rozdział 7 (*CLP dla zmiennych ciągłych*) - w odróżnieniu od poprzednich rozdziałów poświęconych wyłącznie problemom kombinatorycznym - omawia zastosowanie *CLP* dla problemów o *zmiennych ciągłych*. Rozpatruje się zarówno przykłady wyznaczania rozwiązań dopuszczalnych jak i przykłady wyznaczenia rozwiązań optymalnych. Te ostatnie wyznaczane są za pomocą biblioteki *eplex* udostępniającej efektywne *solwery* dla zagadnień programowania liniowego i programowania mieszanego. Przykłady te zostały tak dobrane, by podkreślić charakterystyczny dla *CLP* brak potrzeby doprowadzenia modelu matematycznego problemu do określonej postaci kanonicznej.

Każdy rozdział, z wyjątkiem pierwszego, kończy seria zadań. Większość kombinatorycznych tematów zadań została zaczerpnięta z internetowych witryn łamigłówkowych, gdzie pojawiają się w tylu miejscach i tylu odmianach, że można je uznać za element łamigłówkowego folkloru i zrezygnować z podawania ich adresów internetowych; jest to uzasadnione również tym, że zadania te nie były rozwiązywane metodami *CLP*. Większość zadań optymalizacji to znane problemy *badań operacyjnych*, zaczerpnięte ze znanych podręczników. Aczkolwiek żaden z nich nie został w wymienionych podręcznikach rozwiązany metodami *CLP*, ich pochodzenie zostało w książce udokumentowane.

W tym miejscu dobrze jest poruszyć problemy *terminologii*, *dyscypliny semantycznej* i *ekonomii słownictwa*. Niniejsza książka jest trzecim wydaniem pierwszego tego typu opracowania w języku polskim. Stąd potrzeba przedstawienia polskich terminów dla clp-owych pojęć funkcjonujących od dawna w języku angielskim. By nie tworzyć niepotrzebnej "przepaści terminologicznej", terminy te bardzo często są spolszczonymi terminami angielskimi (lub łacińskimi).

Autor starał się, by przedstawiony tekst nie zawierał synonimów, często spotykanych w literaturze, lecz będących dużą przeszkodą dla każdego stawiającego pierwsze kroki w *Prolog*u i *CLP*. Podejście takie można uzasadnić tak mocną zasadą jak *brzytwa Ockhama*[4]. Wszystkie techniczne terminy stosowane w książ-

[4]Łacińska sentencja "*Entia non sunt multiplicanda praeter neccessitatem*", oznaczająca że "*Nie należy mnożyć bytów ponad potrzeby*", jest przypisywana 14-wiecznemu angielskiemu logikowi, teologowi i franciszkańskiemu mnichowi, Williamowi z Ockhamu (1285–1349). Jest

ce zostały dodatkowo objaśnione w słowniku *Ważne pojęcia* na końcu książki. Dla potrzeb czytelników korzystających z anglojęzycznych witryn poświęconych *CLP* (do czego zdecydowanie namawiam), opracowano również krótki specjalistyczny słownik angielsko-polski.

Zdecydowana niechęć Autora do stosowania synonimów doprowadza często do konfliktu z terminologią stosowaną w ECL^iPS^e. Konflikt ten występuje szczególnie jaskrawo w przypadku takich terminów jak "predykat", "funkcja" i "złożone (compound) termy". Każda funkcja jest relacją, aczkolwiek nie każda relacja jest funkcją; relacje są opisywane predykatami, a więc stosując termin "predykat" nie ma (w ECL^iPS^e) potrzeby stosowania terminu "funkcja". Podobnie termin "złożony (compound) term" oznacza właściwie albo "predykat" albo "strukturę": nie ma "złożonych (compound) termów", które nie byłyby definiowane za pomocą struktur lub predykatów.

0.3 Jak korzystać z książki?

Przede wszystkim należy zainstalować na swym komputerze oprogramowanie *CLP* dla używanego systemu operacyjnego, udostępnione na witrynie ECL^iPS^e pod adresem

`http://www.eclipseclp.org/`.

Oprogramowaniem tym jest w roku 2013, w momencie pisania tej książki, wersja `Release 6.1_167`. W wyniku instalacji powstanie na dysku `C` katalog

`Program Files\ECLiPSe6.1`,

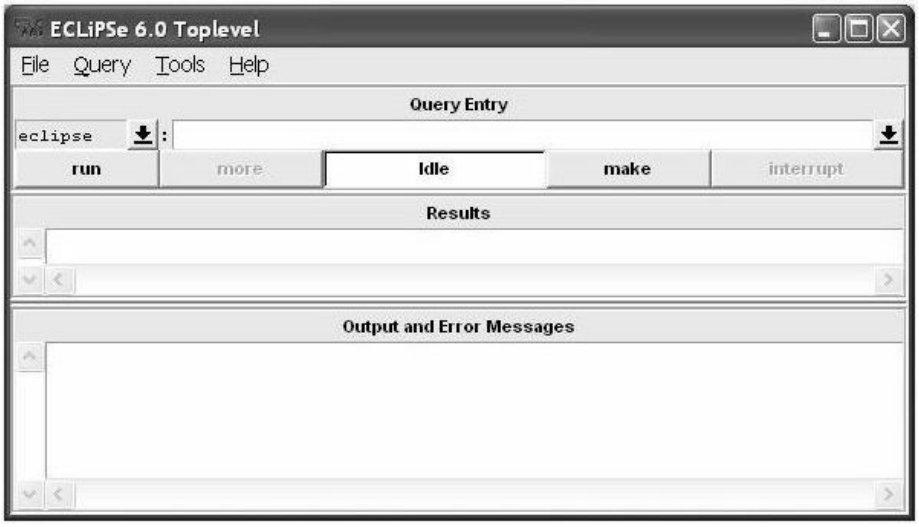
a z listy programów `ECLiPSe6.1` można wyciągnąć na pulpit ikonę `TKEclipse6.1`, pokazaną na rysunku 1.4.

Rysunek 1: Ikona ECL^iPS^e

Kliknięcie w ikonę wywołuje okno główne ECL^iPS^e, pokazane na rysunku 1.5.

ono heurystyką postulującą wyjaśnianie zjawisk w oparciu jak najmniejszą liczbę pojęć.

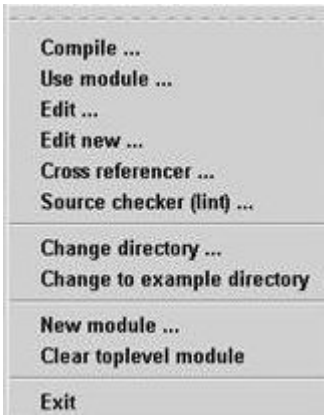
Rozwinięcie opcji *File* udostępnia menu pokazane na rysunku 3. Opcja *Compile* tego okna umożliwia kompilację wywołanego programu z rozszerzeniem *pl* (programy w języku *Prolog*) lub z rozszerzeniem *ecl* (programy typu *CLP*). Ponieważ sprzątanie śmieci w ECL^iPS^e nie jest idealne w przypadku wywoływania wszystkich programów przez *top*, dobrze jest przed kompilacją następnego programu wyczyścić pamięć korzystając z opcji *Clear toplevel module*. Rozwinięcie opcji *Help* udostępnia menu pokazane na rysunku 4.



Rysunek 2: Okno główne ECL^iPS^e

Najważniejszą opcją tego menu jest opcja *Full documentation....* Jej rozwinięcie jest pokazane na rysunku 5. Ta opcja daje użytkownikowi dostęp do definicji wszystkich predykatów ECL^iPS^e (opcja Alphabetical Predicate Index) oraz do obszernej dokumentacji (opcja *User Manual* i opcja *Constraint Library Manual*). Łatwy dostęp do tej dokumentacji sprawia, że w dalszym ciągu - poza nielicznymi wyjątkami - standardowe predykaty platformy ECL^iPS^e nie będą w książce definiowane. Krótkie programy *clp-owe* (nie *prologowe*!) można również uruchamiać w *trybie wiersza polecenia*. Jeżeli jest ustawiona ścieżka do pliku `eclipse.exe`, napisanie słowa "eclipse" w oknie polecenia wywołuje tryb polecenia, jak to pokazano na rysunku 6.

Obecnie można do tego okna wkleić testowany mały program i uruchomić go.



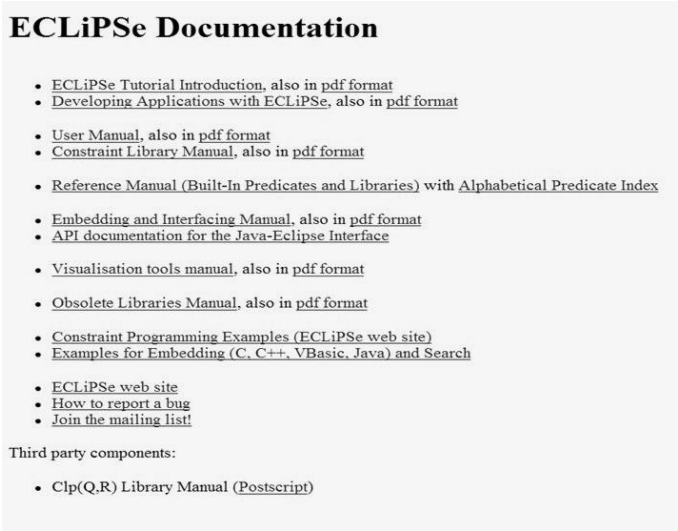
Rysunek 3: Menu *File* okna głównego ECL^iPS^e

Rysunek 4: Menu *Help* okna głównego ECL^iPS^e

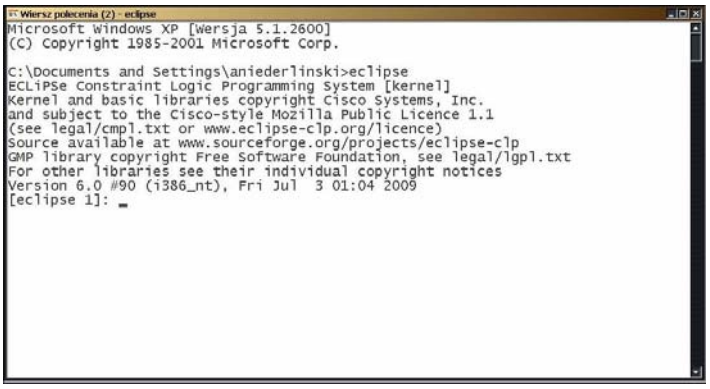
Z trybu polecenia można również korzystać odmiennie: okno polecenia ze skom-

pilowanym programem gotowym do uruchomienia pojawi się automatycznie, jeżeli kliknąć w nazwę programu z rozszerzeniem *.ecl*, pod warunkiem ustawienia odpowiedniej ścieżki dostępu.

Korzystamy z książki tak, jak z każdej innej książki informatycznej: rozwiązując przykłady podręcznikowe w postaci podanej i rozwiązując je w postaci przez siebie mniej lub bardziej dowolnie zmodyfikowanej. Przy uczeniu się *Prolog*u i *CLP* obowiązuje oczywiście stara "zasada narciarska": Jak się nie przewrócisz, to się nie nauczysz! Uczymy się na błędach. Ucząc się *Prolog*u i *CLP* mamy bardzo dużo okazji do popełniania błędów, od prostych błędów formalnych, wyłapywanych przez wbudowaną w ECL^iPS^e diagnostykę, po subtelne błędy logiczne, które może ujawnić tylko wielokrotne i różnorodne testowanie programów i konfrontowanie wyników wnioskowania ze zdrowym rozsądkiem: rychło nabierzemy przy tym przekonania, że człowiek jest równie niedoskonałym aparatem do logicznego wnioskowania, jak do wykonywania złożonych działań arytmetycznych. I w jednym i w drugim przypadku trudno obejść się bez komputera.



Rysunek 5: Rozwinięcie opcji *Full documentation...* okna głównego ECL^iPS^e



Rysunek 6: Uruchamianie ECL^iPS^e w oknie polecenia

0.4 Pewna niedogodność ECL^iPS^ea.

ECL^iPS^e wymaga stosowania (w nazwach predykatów, w nazwach zmiennych, w nazwach programów, w nazwach ścieżek dostępu do programów i w komunikatach generowanych przez program) wyłącznie liter alfabetu *łacińskiego* (zwanego często obecnie alfabetem *angielskim*). Co w takim razie zrobić z polskimi znakami diakrytycznymi? Możliwe są dwa rozwiązania:

- Zrezygnować ze stosowania polskich nazw "prywatnych" predykatów, "prywatnych" zmiennych, "prywatnych" ścieżek dostępu, "prywatnych" programów i "prywatnych" komunikatów. Rozwiązanie to utrudnia niestety proces uczenia, gdyż angielskie nazwy "prywatnych" predykatów mogą być (przez początkującego użytkownika) uważane za nazwy predykatów standardowych, o znaczeniu szukanym w dokumentacji ECL^iPS^ea. A dużą zaletą ECL^iPS^ea jest właśnie to, że odpowiednio dobrane nazwy "prywatnych" predykatów i "prywatnych" zmiennych czynią program ECL^iPS^eowy samodokumentującym.

- Stosować polskie nazwy predykatów, polskie nazwy zmiennych i polskie teksty w komunikatach i nazwach programów z pominięciem polskich znaków diakrytycznych. W efekcie pojawią się językowe potworki, np. sala będzie "Zolta" a nie "Żółta", zaś "rozwiązania" będą "rozwiazaniami".

Obydwa rozwiązania są niedobre, ale z punktu widzenia początkującego użytkownika (a dla takich jest przeznaczona książka), drugie rozwiązanie wydaje się być mniej niedobrym aniżeli pierwsze. Z góry liczę na wyrozumiałość czytelnika. Nagrodą będzie niezachwiana pewność, że każde słowo angielskie (z wyjątkiem słowa `top`) będzie słowem kluczowym, zdefiniowanym w dokumentacji ECL^iPS^e. Słowo `top` oznacza "prywatny" predykat służący do uruchomienia (po skompilowaniu) każdego z programów przedstawionych w książce. Słowo to wzięło się stąd, że jest na początku tekstu programu (jest wnioskiem pierwszej reguły programu). Zaletą takiego rozwiązania jest jednolitość i prostota uruchamiania dowolnego programu omówionego w książce.

0.5 Podziękowania

Autorowi nie było dane zetknąć się w początkach pracy zawodowej z osobami znającymi *Prolog* i *CLP* oraz entuzjazmującymi się nimi. Nie ma też w tym obszarze żadnego formalnego wykształcenia, nigdy nie słuchał wykładów z *Prolog*u lub *CLP*, ani nie uczęszczał na ćwiczenia z tej problematyki. Wszystko, co wie na ten temat, jest wynikiem samouctwa, bazującego na szeregu bardzo dobrych książkach i programach komputerowych, napisanych przez osoby, z którymi Autor nigdy się nie zetknął. Autor uważa jednak, że osoby te - ze względu na wiedzę i inspirację, którą im zawdzięcza - powinny zostać tu wymienione.

Pierwszą i z pewnością najważniejszą książką była autorstwa K.L. Clark'a i F.G. McCabe'a (por. [Clark-84]). Był to wspaniały podręcznik, który zainfekował Autora *Prolog*iem. W połowie lat 80-tych Autor przerobił wszystkie zamieszczone tam przykłady korzystając z mikrokomptera SINCLAIR ZX Spectrum i z interpretera `microProlog`, rozprowadzanego na taśmach magnetofonowych "audio" i pracującego pod systemem operacyjnym `CP/M`. Korzystając z tego oprogramowania Autor zaczął prowadzić zajęcia z przedmiotu "Systemy automatycznego wnioskowania" dla studentów kierunku "Automatyka i Robotyka" na Wydziale Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach.

Nieco później Autor korzystał z *Turbo-Prologu* firmy PDC [4] i jego następców - *PDC Prolog* i *Visual Prolog*. Z początkiem lat 90-tych, w czasie pobytu u Profesora Mieczysława Brdysia w University of Birmingham, UK, Author miał pierwszy kontakt z *CLP* czytając inspirującą książkę van Hentenryck'a ([van Hentenryck-89]). Wynikiem tej lektury był wykład, laboratorium, prace dyplomowe dla studentów specjalizacji "Komputerowe systemy sterowania"

[4] PDC to Prolog Development Center, firma software'owa z Kopenhagi.

oraz prace doktorskie kilku współpracowników Autora, prowadzone w oparciu o oprogramowanie CHIP 5.2 firmy Cosytec. Autor stosował to oprogramowanie przez szereg lat, podziwiając jego elegancję i siłę. Jedyną jego wadą jest wysoka cena i brak wersji edukacyjnej.

W 1997 r. Autor trafił na dwie witryny internetowe Profesora Romana Bartáka z Uniwersytetu Karola w Pradze, Czechy, por. [Bartak-10] i [Bartak-10a], co było początkiem serii owocnych i przyjaznych kontaktów. Wspaniałe referaty Profesora Bartáka na serii warsztatów "Workshops on Constraint Programming for Decision and Control", które w latach 1999-2005 odbywały się corocznie w Instytucie Automatyki Politechniki Śląskiej, były sporym wsparciem dla pracy Autora i jego doktorantów.

Dzięki życzliwej inicjatywie Profesora Jerzego Gołuchowskiego z Uniwersytetu Ekonomicznego w Katowicach, Autor ma - poczynając od roku 2009 - okazję pracy nad problematyką *CLP* w Katedrze Inżynierii Wiedzy na Wydziale Informatyki i Komunikacji tego Uniwersytetu. Została ona zapoczątkowana serią wykładów dla pracowników Katedry z problematyki *CLP* w środowisku ECL^iPS^e i jest kontynuowana, m. in. w ramach regularnych wykładów i ćwiczeń ze studentami kierunku *Informatyka* oraz *Informatyka i Ekonometria*. Autor jest wdzięczny Profesorowi Gołuchowskiemu i Kolegom z Katedry Inżynierii Wiedzy za zainteresowanie problematyką *CLP* i stworzenie warunków sprzyjających pisaniu książek z tej problematyki.

Wykłady Autora z *CLP* oraz niniejsza książka zyskały niewątpliwie dużo w wyniku dyskusji i analiz prowadzonych z doktorantami, dr inż. Tomaszem Szczygłem, autorem rozprawy [Szczygieł-03], dr inż. Wojciechem Legierskim, autorem rozprawy [Legierski-06], dr inż. Tomaszem Tatoniem, autorem rozprawy [Tatoń-09], dr inż. Wojciechem Pieprzycą, autorem rozprawy [Pieprzyca-09], dr inż. Rafałem Szklarczykiem, autorem rozprawy [Szklarczyk-09] i dr inż. Łukaszem Domagałą, autorem rozprawy [Domagala-11]. Dr inż. Łukasz Domagała jest autorem zamieszczonych w książce programów dla problemu komiwojażera.

Moi Koledzy z byłego Zakłady Komputerowych Systemów Sterowania Instytutu Automatyki Politechniki Śląskiej, dr inż. Jerzy Mościński, dr inż. Dariusz Bismor i dr inż. Krzysztof Czyż, pomogli mi bardzo wyjaśniając szereg osobliwości *Miktexa* stosowanego przy pisaniu tej książki. Pan dr inż. Jacek Loska ma duże zasługi w aktualizacji i reanimacji mojego sprzętu komputerowego i jego oprogramowania.

Autor jest wdzięczny Hakanowi Kjellerstrandowi, niezależnemu programiście ze Szwecji, za to że - w bardzo krótkim terminie - przeczytał angielskie wydanie tej książki ([Niederliński-11]) i przedstawił szereg istotnych uwag odnośnie jej treści.

Jeden z twórców ECL^iPS^e - Joachim Schimpf z Cisco - zasługuje na specjalne podziękowania za wyjaśnienie szeregu subtelności programistycznych tej platformy.

Autor jest wdzięczny Panu Jackowi Skalmierskiemu, którego wydawnictwo PKJS od lat jest niezmiernie pomocne przy drukowaniu kolejnych książek.

Autor jest wdzięczny Panu mgr inż. Michałowi Mazurowi (studentowi Studium Doktoranckiego na Wydziale Automatyki, Elektroniki i Informatyki Politechniki Śląskiej) za uważne przeczytanie książki i wniesienie dużej liczby poprawek językowych.

Dla postępu prac nad książką nieocenionym było jak zwykle zrozumienie i wsparcie żony Autora, Teresy, tolerującej od lat to, że Autor jest obecny w domu zdecydowanie częściej ciałem, aniżeli duchem.

Jest rzeczą oczywistą, że Autor jest jedyną osobą odpowiedzialną za ewentualne błędy, niedopowiedzenia i niewłaściwe interpretacje, które mogły w tej książce się pojawić.

Autor udostępnia nieodpłatnie PDF tej książki na swojej witrynie internetowej http://www.pwlzo.pl. Jest to (zapewne niedoskonałym) podziękowaniem składanym tym wszystkim, mniej lub bardziej anonimowym, członkom internetowej społeczności clp-owej, których opracowania czytał, z których opracowań uczył się, których opracowania były inspiracją. Jest to również spłatą zaciągniętego długu wdzięczności, spłatą przekazaną tym wszystkim, którzy pragną się *CLP* nauczyć.

Gliwice, styczeń 2014.

Rozdział 1

Wstęp

1.1 Czym jest programowanie w logice z ograniczeniami?

Programowanie w logice z ograniczeniami jest narzędziem do rozwiązywania *problemów spełniania ograniczeń* (*PSO*). Problemy takie można - dla problemów kombinatorycznych - zdefiniować następująco:

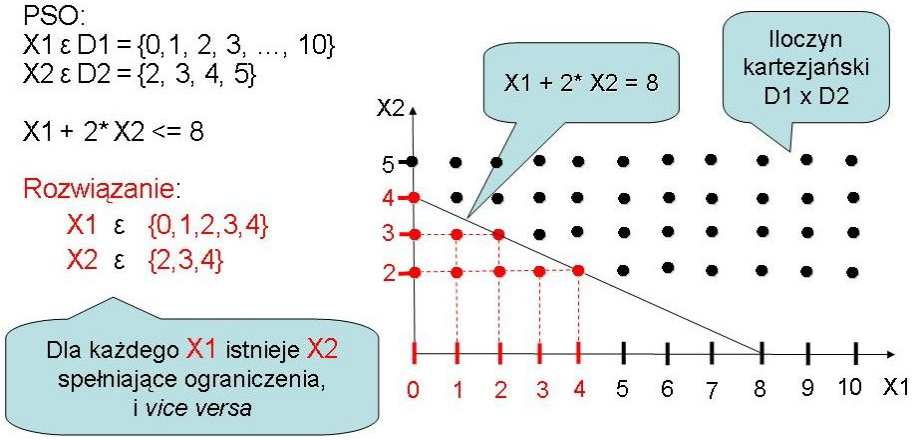
- dany jest skończony zbiór S zmiennych $X_1, X_2, ..., X_n$, przyjmujących wartości ze skończonych *dziedzin* $D_1, D_2, ..., D_n$;

- dany jest skończony zbiór ograniczeń pomiędzy wymienionymi zmiennymi. *Ograniczeniem i-tym* $C_i(X_{i_1}, X_{i_2}, ..., X_{i_k})$ pomiędzy k zmiennymi ze zbioru S nazywa się *relację* będącą takim podzbiorem *iloczynu kartezjańskiego* $D_{i_1} \times D_{i_2} \times, ..., \times D_{i_k}$, który określa wartości zmiennych *pasujące* do siebie w sensie zdefiniowanym przez rozpatrywany problem. Często ograniczenia nie muszą być definiowane za pomocą rzeczonych relacji, lecz mogą być definiowane za pomocą równań, nierówności, podprogramów. Liczba zmiennych występujących w ograniczeniu jest nazywana *arnością* ograniczenia. Ograniczenie dla pojedynczej zmiennej nazywa się *unarnym*, ograniczenie dla dwóch zmiennych - *binarnym*, ograniczenie dla `n > 2` zmiennych - *n-arnym*;

- *rozwiązaniem PSO* nazywa się każde takie przyporządkowanie wszystkim zmiennym wartości z ich dziedzin, które spełnia wszystkie ograniczenia;

- rozwiązanie takie może dodatkowo spełniać warunek minimalizacji lub maksymalizacji określonego *wskaźnika jakości* o charakterze ekonomicznym, np. przedstawiającym straty lub zysk. Mówimy wtedy o *problemie optymalnego spełniania ograniczeń* (*POSO*).

Definicje te ilustrują przykłady z rysunku 1.1 dla wielokrotnego rozwiązania *PSO*, z rysunku 1.2 dla unikatowego rozwiązania *PSO* i z rysunku 1.3 dla optymalnego rozwiązania *POSO*.

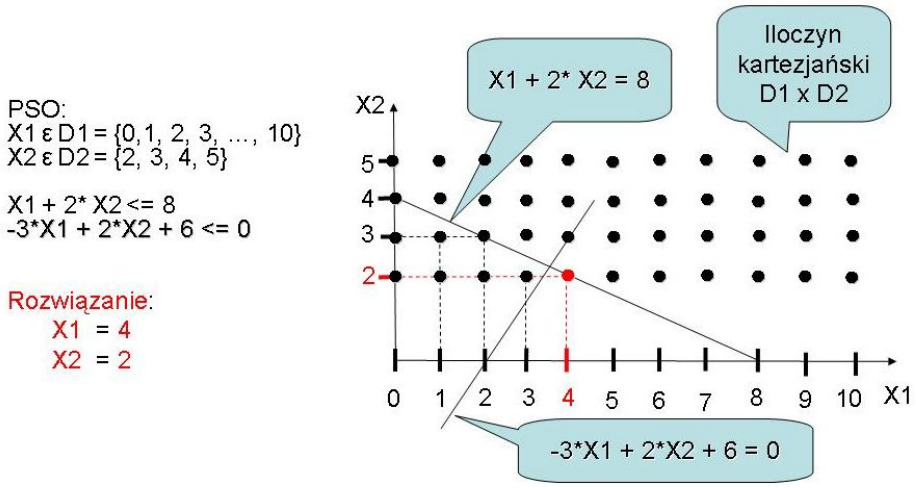


Rysunek 1.1: Prosty przykład *PSO* z wielokrotnym rozwiązaniem.

1.2 Jakie zalety ma programowania w logice z ograniczeniami?

Istotną cechą kombinatorycznych *PSO* i *POSO* jest to, że każda zmienna przyjmuje wartości ze *skończonej* dziedziny. Oznacza to, że omawiane problemy są *problemami rozstrzygalnymi*: stosunkowo proste algorytmy *przeglądu zupełnego*[1], mogą - teoretycznie rzecz biorąc - zawsze rozwiązać *PSO* lub *POSO* albo

[1]Przegląd zupełny polega na kolejnym generowaniu wszystkich możliwych permutacji wartości zbioru zmiennych $X_1, X_2, ..., X_n$ i sprawdzaniu, czy spełniają one wszystkie ograniczenia problemu.



Rysunek 1.2: Prosty przykład *PSO* z unikatowym rozwiązaniem.

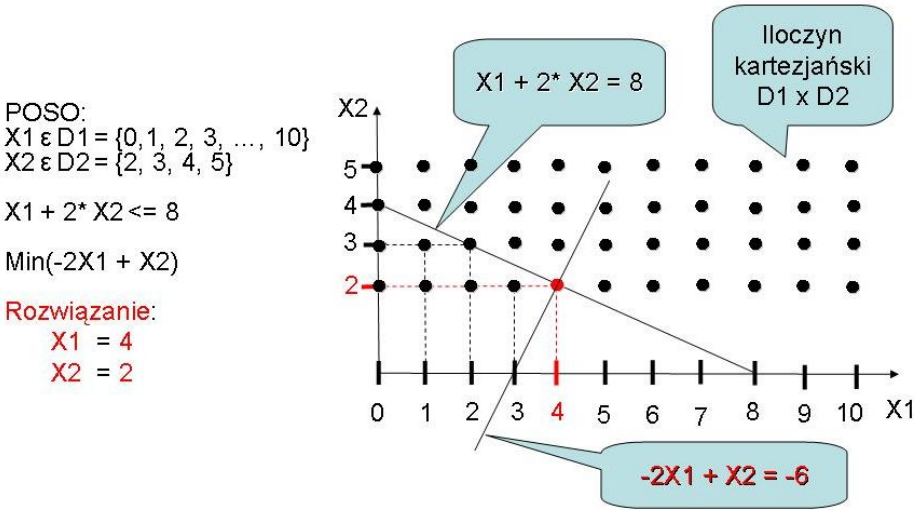
wykazać, że rozwiązanie nie istnieje. Dlaczego więc sięgamy po specjalne języki *CLP*? Odpowiedź na to pytanie składa się z dwóch części:

1. Ze względu na *efektywność* uzyskiwania rozwiązania metodami *CLP*, konkretnie ze względu na czas potrzebny do jego wyznaczenia. Liczba wyliczeń w przypadku stosowania przeglądu zupełnego może być iście kosmiczna. Rozpatrzmy np. przypadek 30 zmiennych, każda z których ma dziedzinę zawierającą 100 wartości[2]. Całkowita liczba zbiorów 30-elementowych (tworzących coś co zwykle nazywa się *przestrzenią stanu*) jest równa $100^{30} = 10^{60}$. Ponieważ ludzie są notorycznie niezdolni do oceny tego, jak *duża* jest *liczba duża*[3], dobrze jest takie liczby przekonwertować w *czas*. Załóżmy, że liczba wyliczeń jakiegoś zbioru ograniczeń dla dowolnego ze zbiorów 30 zmiennych zajmie jedną mikrosekundę. Wyliczenia dla wszystkich zbiorów zajmą 10^{54} sekund lub $10^{54}/3600$ godzin lub $10^{54}/(3600 * 24 * 365)$ lat[4]. Jest oczywistym, że:

[2]To jest naprawdę bardzo mały problem w porównaniu z np. przeciętnym problemem układania uniwersyteckiego rozkładu zajęć.

[3]To dobrze widać w trakcie dyskusji nad budżetami w dowolnym parlamencie.

[4]Dla przeglądu zupełnego robi się zazwyczaj konserwatywne założenie, że rozwiązanie uzyska się dopiero dla ostatniego wyliczanego zbioru.



Rysunek 1.3: Prosty przykład *POSO*.

$$10^{54}/(3600 * 24 * 365) > 10^{54}/(10000 * 100 * 1000) = 10^{45},$$

więc przegląd zupełny może trwać dłużej niż 10^{45} lat, co jest czasem dłuższym od estymowanego przez naukowców czasu istnienia wszechświata ($1.4 * 10^{10}$ lat). Nazwano to bardzo trafnie *kombinatoryczną eksplozją*, lub - zob. [Bellman-61] - *przekleństwem wymiarowości*. Z drugiej strony jednak wiadomo, że uniwersyteckie rozkłady zajęć koniec końców powstają, tworzone w wielkim trudzie przez ludzi stosujących różne heurystyki i odwołujący się do ubiegłorocznych rozwiązań. Techniki clp-owe aspirują do wsparcia tych wysiłków ludzkich: są one dla takich problemów bardziej użyteczne aniżeli przegląd zupełny dlatego, że dzięki odpowiednio wczesnemu i inteligentnemu korzystaniu z ograniczeń problemu[5] oraz prostocie implementowania specyficznych dla problemu heurystyk, znacznie zmniejszają liczbę wyliczanych relacji.

2. Ze względu na *deklaratywność* programów clp-owych. *Deklaratywność* oznacza, że *odpowiedni* opis rozwiązywanego problemu jest zarazem programem rozwiązującym ów problem. Oznacza ona również, że programy prologo-

[5] To właśnie są szekspirowskie *słodkie pożytki z przeciwności*!

we i clp-owe nie zawierają algorytmów rozwiązywania zadań, co jest charakterystyczne dla *programowania proceduralnego*; wystarczy *odpowiedni* opis tego zadania. Upraszczając nieco można powiedzieć, że sztuka programowania prologowego i clp-owego polega na tworzeniu takiego opisu rozwiązywanego problemu, który jest zrozumiały i efektywny dla szeregu uniwersalnych algorytmów rozwiązywania problemów typu *PSO* i *POSO*, wbudowanych w kompilatory *Prolog*u lub *CLP*[6].

1.3 Co oznacza termin "ograniczenie"?

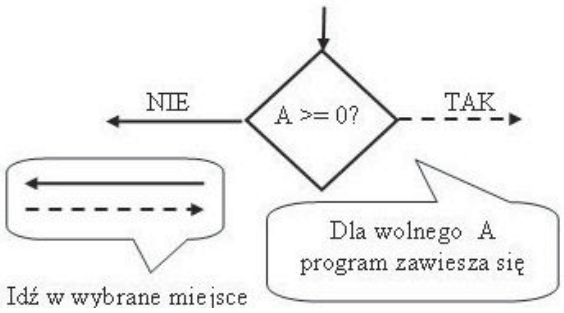
W tytule książki i w nazwie języków *CLP* występuje termin "ograniczenie", zasługujący na baczniejszą uwagę. *Ograniczeniami* nazywa się potocznie wszystko, co zawęża nasze pole działania. Z obsługą ograniczeń mamy do czynienia we wszystkich językach programowania. Jednakże sposób obsługi ograniczeń w językach imperatywnych (np. *Pascal*, *C*, *C++*) różni się zasadniczo od sposobu obsługi ograniczeń w *Prolog*u i w językach *CLP*. Ograniczenia w językach imperatywnych są *pasywne*, co znaczy że mogą być stosowane tylko wtedy, gdy są *uziemione*[7]. Ograniczenia pasywne służą tylko do wyboru dalszego sposobu realizacji programu w zależności od tego, czy ograniczenie jest, czy też nie jest spełnione. Ilustruje to rysunek 1.4.

Ograniczenia w *Prolog*u i w językach *CLP* są *aktywne*, co znaczy że mogą być stosowane również wtedy, gdy nie są uziemione, ba - nawet wtedy gdy żadna zmienna nie jest uziemiona. Ograniczenia aktywne (oznaczane często specjalnymi symbolami) służą bowiem do inicjowania *poszukiwań* wartości zmiennych je spełniających.

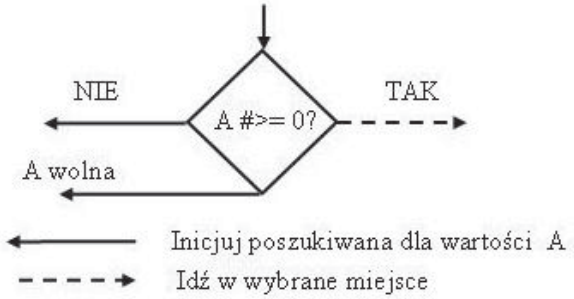
Ilustruje to rysunek 1.5. Poszukiwania te są realizowane w *dziedzinach* zmiennych, będących dopuszczalnymi zbiorami wartości, deklarowanymi dla każdej zmiennej.

[6] Aczkolwiek myślenie deklaratywne jest uważane za prostsze od myślenia proceduralnego (zob. [Apt-07]), a programy deklaratywne łatwiej opracować, zrozumieć i zmodyfikować aniżeli ich proceduralne odpowiedniki, to jednak pisanie deklaratywnych programów w *Prolog*u lub *CLP* nie jest z reguły przysłowiową *bułką z masłem*. Trudności związane z opracowaniem *efektywnych* algorytmów zostają bowiem zastąpione trudnościami (zdecydowanie mniejszymi) tworzenia *efektywnych* opisów problemów.

[7] *Uziemione wyrażenie* (*grounded expression*) jest terminem stosowanym w logice dla określenia wyrażeń nie zawierających zmiennych wolnych. Termin ten jest powszechnie stosowany również w *Prolog*u i językach *CLP*.



Rysunek 1.4: Przykład ograniczenia pasywnego (dla testowania)



Rysunek 1.5: Przykład ograniczenia aktywnego (dla inicjowania poszukiwań)

1.4 Programowanie w logice z ograniczeniami a sztuczna inteligencja

Sztuczną inteligencją zwykło się nazywać gałąź informatyki, która zajmuje się tworzeniem programów wykonujących działania uważane zwykle za przejaw inteligencji, patrz [Poole-98], [Luger-98], [Russel-03] i [Turban-11]. I tak np. program wyznaczający rozkład zajęć studenckich dla dużego wydziału wyższej uczelni (por. np. [Legierski-06]) będzie z pewnością na tę nazwę zasługiwał. Nazwa ta będzie również adekwatna dla programu wyręczającego człowieka przy tworzeniu skomplikowanych *marszrut* floty pojazdów w taki sposób, by - przy zapewnieniu dostaw rozwożonych towarów z licznych magazynów licznym od-

biorcom - zminimalizować długość trasy przebytej przez te pojazdy, patrz np. [Szklarczyk-09]. Tworzenie rozkładów zajęć lub marszrut *ręcznie* (tzn. przez ludzi) wymaga bowiem od nich bardzo nietrywialnej inteligencji, rozumianej jako zdolność dostrzegania zależności i zdolność wnioskowania dla tych zależności w sposób zmierzający do osiągnięcia celu, tzn. rozkładu nie kolidujących ze sobą zajęć, wygodnego dla studentów, lub marszruty o minimalnej długości trasy. Szereg autorów (patrz np. Puget [Puget-08] uważa, że programowanie z ograniczeniami jest jednym z największych sukcesów badań w obszarze sztucznej inteligencji. Puget wymienia m.in. następujące osiągnięcia istniejących współcześnie narzędzi *CLP* dla jednej z najczęściej spotykanych grup aplikacji jaką jest harmonogramowanie:

- harmonogramowanie pracy lakierni karoserii samochodowych, dla którego istotne jest minimalizowanie bardzo kosztownych zmian kolorów, wymagających gruntownego mycia instalacji lakierującej;
- harmonogramowanie prac montażowych u dużego producenta sprzętu gospodarstwa domowego w sposób zapewniający odpowiednio szybką realizację zamówień przy niskich poziomach zapasów materiałowych;
- harmonogramowanie pracy informatyków dużego banku komercyjnego, zatrudnionych w tzw. trybie 7/24 (siedem dni, dwadzieścia cztery godzin na dobę), z zachowaniem wszystkich skomplikowanych regulacji umowy o pracę.

Zestaw ten został znacznie rozszerzony przez Simonisa (patrz [Simonis-10]), który wymienia m. in. takie ciekawe i trudne zastosowania jak:

- projektowanie inteligentnych systemów okablowania dla dużych budynków (minimalizacja długości kabla, ograniczenie liczby wierceń, ograniczenia liczby przełączników)
- konfigurowanie sieci energetycznych, sterowanie sieci wodociagowych;
- sterowanie usługami typu *bandwidth on demand*, zwiększającymi dynamicznie przepustowość sieci komputerowych w zależności od zapotrzebowania użytkownika, np. dla potrzeb wideo-konferencji;
- planowanie transportu lotniczego, samochodowego i kolejowego;
- rozkładowanie pracy różnych załóg, np. szpitali, lotnisk, dworców, barów, restauracji, sklepów wielko-powierzchniowych;

- wybór drogi transmisji i częstotliwości transmisji w sieciach komputerowych.

Badacze, twórcy i użytkownicy z obszaru sztucznej inteligencji aspirowali zawsze do rozwiązywania bardzo złożonych, kompleksowych problemów, patrz np. [Luger-98] i [Turban-11]. Dokładnie taki sam cel przyświeca badaczom, twórcom i użytkownikom w obszarze *CLP*.

Dobrze jest również zwrócić uwagę na to, że związek *Prolog*u i *CLP* ze *Sztuczną Inteligencją* jest głębszy i bardziej podstawowy. Wynika to stąd, że za początek współczesnej sztucznej inteligencji można uznać zaniechanie prac nad *Uniwersalnym Rozwiązywaczem Problemów*[8] (ang. *GPS - General Problem Solver*). Okazało się bowiem, że rozwiązywania realistycznych problemów nie da się sprowadzić tylko do zastosowania bardzo wyrafinowanej logiki. Wymaga ono bowiem bardzo dużej wiedzy z dziedziny tych problemów, lecz zadawala się znacznie skromniejszym aparatem logicznym aniżeli *GPS*. Ponieważ nie udało się do tej pory zamodelować inteligentnego działania, które nie byłoby wynikiem wiedzy i poprawnego wnioskowania, naturalną tendencją w tworzonych programach w obszarze sztucznej inteligencji było rozdzielenie tych dwóch aspektów: wiedzy dziedzinowej (zapisanej w jakiejś deklaratywnej formie) i systemu wnios/-kującego nad tą wiedzą[9]. Z tym właśnie mamy do czynienia programując w *Prolog*u lub w *CLP*: program jest deklaracją wiedzy dziedzinowej z obszaru danego problemu, nad którą wnioskuje system wnioskujący będący elementem kompilatora *Prolog*u lub *CLP*.

1.5 Programowanie w logice z ograniczeniami a inżynieria wiedzy

Dla wyjaśnienia pojęcia *wiedza* w kontekście inżynierii wiedzy, dobrze jest przedstawić je w nawiązaniu do pojęć bardziej elementarnych jakimi są *dane* i *informacja*. Najczęściej definiuje się te pojęcia następująco:

- *dane* to zbiory wektorów zero-jedynkowych, przedstawiających liczby, znaki, wyrazy, obrazy, dźwięki. Są niezorganizowane i nieprzetworzone. Powstają z reguły na drodze szeroko rozumianego pomiaru. Reprezentacją danych są bity, bajty, słowa, listy, tablice, rekordy;

[8] Prace te zostały zapoczątkowane na przełomie lat 50 i 60 minionego stulecia przez Herberta Simona i Allena Newella z Carnegie-Mellon University w Pittsburghu, PA, USA.

[9] Taki typ programowania zwykło się nazywać *Knowledge Based Programming*, co tłumaczy się na *programowanie z wyodrębnioną bazą wiedzy.*

- *informacja = dane + znaczenie danych + przeznaczenie danych.* Informacja jest taką reprezentacją danych, która ma określone znaczenie i określone przeznaczenie. Informacja powstaje z reguły na drodze przetwarzania danych, systematyzacji danych i agregacji danych. Reprezentacją informacji są bazy danych i hurtownie danych;
- *wiedza = informacja + umiejętność korzystania z niej.* Wiedza jest zdolnością korzystania z informacji usystematyzowanej i zagregatyzowanej dla określonego celu, np. dla oceny sytuacji i podejmowania racjonalnych decyzji. Reprezentacją wiedzy są m. in. *fakty*, *reguły*, *modele matematyczne*, *rysunki techniczne*.

Wiedza do niedawna wydawała się być wyłącznie atrybutem i produktem racjonalnie myślących istot ludzkich. Ostatnie trzydziestolecie pokazało jednak, że procesy zdobywania wiedzy, gromadzenia wiedzy, przedstawiania wiedzy i jej stosowania można - dla ograniczonych i dobrze zdefiniowanych dziedzin wiedzy, najczęściej związanych z szeroko pojętym zarządzaniem przedsiębiorstwami - w pewnym zakresie automatyzować z zastosowaniem technik komputerowych. Okazało się również, że komputerowo wspomagane zdobywanie, prezentowanie i stosowanie wiedzy jest źródłem niebagatelnych efektów ekonomicznych i społecznych, przede wszystkim w sferze zarządzania. Znalazło to odzwierciedlenie w postaci powstania i rozwoju dyscypliny naukowej i specjalności dydaktycznej zwanej *inżynierią wiedzy* (ang. *Knowledge Engineering*). Można ją zdefiniować następująco (por. np. [Brachman-04]):

Inżynierią wiedzy nazywa się dziedzinę informatyki zmierzającą do wydobywania wiedzy z baz danych i hurtowni danych oraz jej prezentacji w sposób zrozumiały dla komputerów w celu umożliwienia im wyciągania wniosków z tej wiedzy.

Istotną dla tego celu właściwością języków *CLP* (a również ich protoplasty, którym jest *Prolog*), jest możliwość przedstawienia wiedzy w sposób *deklaratywny*, za pomocą logiki posługującej się *faktami* i *regułami* oraz - dodatkowo - odwołującej się do *modeli matematycznych* relacyjnych i arytmetycznych, por.np. [Niederliński-06] i [Niederliński-11a]. Okazuje się, że właśnie taka reprezentacja wiedzy jest szczególnie wygodna dla celów *zarządzania wiedzą* (patrz np. [Gołuchowski-07]), tzn. jej przechowywania, udostępniania, przetwarzania w systemach komputerowych i - *last but not least* - zastosowanie do podejmowania decyzji zarządczych i biznesowych (patrz np. [Morgan-08], [Ross-03] i [von Halle-02]).

1.6 Programowanie w logice z ograniczeniami a badania operacyjne

Przedmiotem badań operacyjnych (obszernie opisanych m. in. w monografiach [Wagner-75], [Winston-94], [Williams-99], [Trzaskalik-08] i [Taha-08]) jest modelowanie problemów decyzyjnych charakteryzujących się ograniczonością resursów oraz wyznaczanie optymalnych decyzji dla tych problemów za pomocą takich narzędzi numerycznych jak *programowanie liniowe*, *programowanie liniowe całkowitoliczbowe*, *programowanie mieszane*. Termin *programowanie* występujący w tych nazwach nie oznacza *programowania komputerów*, lecz jest historycznie uwarunkowaną nazwą pewnej grupy technik numerycznej optymalizacji, najczęściej opracowanych dla następującej *postaci kanonicznej* problemu decyzyjnego: wyznaczyć

$$\min_{\mathbf{x}} \mathbf{c}^T\mathbf{x}$$

przy ograniczeniu:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

gdzie $\mathbf{x}$ jest n-wymiarowym wektorem kolumnowym rzeczywistych lub zero-jedynkowych zmiennych decyzyjnych, $\mathbf{A}$ jest macierzą o wymiarach $m \times n$ i elementach rzeczywistych lub zero-jedynkowych, a $\mathbf{c}^T$ jest n-wymiarowym wektorem wierszowym o elementach rzeczywistych lub zero-jedynkowych. Współczesne platformy *CLP* (w tym również ECL^iPS^e) dysponują *solwerami* umożliwiającymi skuteczne rozwiązywanie takich problemów. Co więcej, rozwiązania takie nie wymagają doprowadzenia problemu optymalizacji do przedstawionej postaci kanonicznej: jest to ich poważną zaletą, gdyż w postaci kanonicznej ulegają *rozmyciu* parametry ekonomiczne charakteryzujące pierwotny problem optymalizacji; co z tym związane, rozwiązania takie charakteryzują się dużą prostotą wprowadzania skomplikowanych ograniczeń. Porównanie rozwiązań problemów optymalizacji metodami badań operacyjnych i metodami *CLP* jest przedmiotem dużej liczby publikacji i szeregu wartościowych opracowań monograficznych, por. np. [Hooker-00], [Hansen-03],[Milano-04] i [Hooker-07].

1.7 Klasyfikacja rozwiązywanych problemów

Względy natury dydaktycznej wymagają ustrukturyzowania materiału w sposób ułatwiający dostrzeżenia podobieństw pomiędzy różnymi problemami, a tym

samym ułatwiający ich rozwiązywanie. Problemy rozwiązywane za pomocą *Prolog*u i *CLP* można uznać za przynależne do jednej z poniższych czterech klas:

1. Problemy typu SD (*Stan Dopuszczalny*) polegają na znalezieniu stanu spełniającego wszystkie ograniczenia problemu , lub prościej problemy wyznaczania pełnych opisów pewnych sytuacji na podstawie znajomości pewnych częściowych lecz wystarczających danych, oraz znajomości ograniczeń, które pełny opis powinien spełniać. Do tej klasy należy większość popularnych zagadek i łamigłówek, dla których dany jest *stan częściowy*, a należy określić wartości pozostałych współrzędnych stanu tak, by otrzymać stan dopuszczalny spełniający ograniczenia problemu. O znaczeniu takich zagadek i łamigłówek dla uczenia się *Prolog*u (i *CLP*) świadczy istnienie dużej liczby specjalnych witryn internetowych (zob. [Edmund-10] lub http://brownbuffalo.sourceforge.net/). Przedstawiają one w bardzo uproszczonej postaci problemy występujące w takich realistycznych aplikacjach jak uniwersyteckie i przemysłowe rozkładowania. Niestety, złożoność wymienionych problemów i liczba opisujących je zmiennych jest tak duża, że nie nadają się do nauczania *Prolog*u i *CLP*. Problemy typu SD można podzielić na:

   - Problemy *konfigurowania*, których istotą jest wyznaczenie - z zadanych zbiorów - takiego zbioru, którego elementy spełniają zadane ograniczenia *przeznaczenia* i *kompatybilności*.
   - Problemy *przyporządkowania*, których istota jest wyznaczenie, dla każdego elementu jednego zbioru, jakiegoś elementu innego zbioru w sposób spełniający zadane ograniczenia .
   - Problemy *rozkładowania*, polegające na rozmieszczaniu zadań (ze zbioru zadań) w różnych przedziałach czasu (ze zbioru przedziałów czasu).

2. Problemy typu TSD (*Trajektora Stanów Dopuszczalnych*) polegają na znalezieniu ciągu stanów dopuszczalnych, od zadanego *stanu początkowego* do zadanego *stanu końcowego*, przy zachowaniu wszystkich ograniczeń dla przejść pomiędzy sąsiadującymi stanami. Problemy typu TSD są naogół trudniejsze aniżeli problemy SD. Są one reprezentowane zarówno przez zagadki i łamigłówki (np, przeprawy przez rzekę, znalezienie drogi w labiryncie) jak i przez poważne aplikacje harmonogramowania przemysłowego i biznesowego, jak np. wyznaczenie kolejności przezbrajania linii produk-

cyjnych lub wyznaczenie marszrut pojazdów. Problemy typu TSD można podzielić na:

- Problemy *szeregowania*, których istotą jest określenie kolejności zadań z zadanego zbioru w sposób zapewniający przestrzeganie właściwej kolejności tych zadań.
- Problemy *harmogramowania*, będące problemami szeregowania z dodatkowymi ograniczeniami na wielkość resursów potrzebnych do wykonania zadań.

3. Problemy typu SO (*Stan Optymalny*) polegają na znalezieniu takiego stanu dopuszczalnego, który zapewni optimum (maksimum lub minimum) jakiegoś *wskaźnika jakości*, najczęściej o charakterze ekonomicznym (zysk, strata). Wymienione problemy mają większą liczbę stanów dopuszczalnych, dzięki czemu jest możliwe wyznaczenie stanu optymalnego. Takimi problemami są np. problemy kompletowania załogi minimalizujące całkowitą liczbę zatrudnionych. Problemy te można również podzielić na problemy *konfigurowania*, *przyporządkowania* i *rozkładowania* z dodatkowym wymogiem optymalizacji odpowiedniego *wskaźnika jakości*.

4. Problemy typu TSO (*Trajektoria Stanów Optymalnych*) polegają na znalezieniu ciągu stanów dopuszczalnych, od zadanego *stanu początkowego* do zadanego *stanu końcowe*go, przy zachowaniu wszystkich ograniczeń dla przejść pomiędzy sąsiadującymi stanami, i dodatkowej optymalizacji wskaźnika jakości. Dla problemów tych istnieje zawsze większa liczba dopuszczalnych trajektorii, dzięki czemu jest możliwe wyznaczenie trajektorii optymalnej. Klasycznymi przykładami problemów typu TSO są problemy linii zadań i problemy marszrutyzacji pojazdów. Problemy te można również podzielić na problemy *szeregowania*, *rozkładowania* i *harmogramowania* z dodatkowym wymogiem optymalizacji odpowiedniego *wskaźnika jakości*.

Dużo przemawia za tym, że wymienione cztery kategorie problemów wyczerpują wszystkie znane aplikacje *Prolog*u i *CLP*. Autor nie zetknął się do tej pory z problemem rozwiązywanym za pomocą *Prolog*u lub *CLP*, którego nie można by sprowadzić do jednej z wymienionych czterech kategorii. Dlatego też problemy z tych kategorii będą występować we wszystkich rozdziałach książki.

Rozdział 2

Na początku był Prolog

Pierwszym językiem programowania, zawierającym w zalążku podstawowe mechanizmy działania języków *CLP*, takie jak poszukiwanie z nawrotami i propagacja ograniczeń, był *Prolog*[1]. Ze względu na prostotę i przejrzystość tych mechanizmów w Prologu, zaczniemy od ich przedstawienia korzystając z *Prolog*u zaimplementowanego w ECL^iPS^e CPS.

2.1 Elementy Prologu

Nazwa *Prolog* jest akronimem słów "PROgramming in LOGic"[2]. Istotą *Prolog*u jest bardzo inspirująca idea, zgodnie z którą program nie składa się z *instrukcji* (jak w przypadku programowania *imperatywnego* (*proceduralnego*)) ani z *funkcji* (jak w przypadku programowania *funkcjonalnego*), lecz z reguł, których wnioskami są *implikacje predykatów*. Idea ta leży u podstaw *deklaratywnego* charakteru *Prolog*u: za pomocą implikacji i predykatów nie można formułować poleceń (tzn. nie można tworzyć algorytmów), lecz można opisywać (deklarować) relacje pomiędzy różnymi elementami rozważanego problemu. Innymi słowy, *Prolog* jest bardzo dobrym narzędziem do *opisu* wiedzy o rozważanym problemie. Aby

[1] Twórcami Prologu były grupy informatyków skupione wokół Alain'a Colmerauer'a w Marsylii, Francja, oraz Roberta Kowalskiego (zob. [Kowalski-89]) w Edinburgu, Zjednoczone Królestwo, w latach 1971-1974.

[2] Słowo "prolog" pochodzenia greckiego oznacza wstęp do większej całości, np. książki czy sztuki teatralne. Mamy tu do czynienie z ciekawym zbiegiem okoliczności: język Prolog okazał się wstępem do bardzo obszernej grupy języków CLP.

użyteczność *Prolog*u była pełna, potrzebny jest jeszcze system, który wnioskuje z zawartej w programie prologowym wiedzy. System ten (zwany w dalszym ciągu *systemem wnioskującym*) ma charakter uniwersalny i stanowi część kompilatora (lub interpretera) języka *Prolog*. Oznacza to, że system wnioskujący jest oddzielony (fizycznie i logicznie) od wiedzy, na temat której wnioskuje. Oznacza to również, że w *Prolog*u programujemy *deklaratywnie*: *opis problemu* jest *modelem problemu* jest *programem modelowania problemu* jest *rozwiązaniem problemu*. Pod "opisem problemu" rozumie się *odpowiedni* (zrozumiały dla *Prolog*u) opis problemu. Upraszczając można powiedzieć, że *sztuka Prologu* polega na tworzeniu odpowiednich (*zrozumiałych* dla *Prolog*u i ponadto *efektywnych obliczeniowo*) opisów problemów.

2.1.1 Dziedzina wnioskowania

Dziedziną wnioskowania *Prolog*u i *CLP* jest dziedzina *termów* zwana również dziedziną *Herbranda*. *Term* to dowolny ciąg znaków pozbawiony znaczenia semantycznego. Wyróżnia się następujące *typy termów*:

- *atom* będący dowolnym ciągiem znaków rozpoczynającym się małą literą lub rozpoczynającym się dowolną literą, lecz znajdującym się pomiędzy podwójnymi lub pojedynczymi górnymi cudzysłowami[3]. *Atomy* są nienumerycznymi (tzn. *logicznymi* lub *symbolicznymi*) *stałymi*. Np. "pada deszcz" jest stałą logiczną, gdyż w danej sytuacji jest ona lub nie jest prawdą, natomiast "Antoni Niederliński" jest stałą symboliczną, gdyż nie można jej przyporządkować wartości logicznej. Stosowanie cudzysłowów umożliwia odróżnianie atomów rozpoczynających się dużymi literami od zmiennych;

- *zmienna*, będąca dowolnym ciągiem znaków rozpoczynającym się dużą literą lub podkreślnikiem, np. `X, A, Jan, Kto, Co, _kto`, `_co`. Zmienne w *Prolog*u i *CLP* spełniają rolę niewiadomych, podobnie jak w logice i algebrze. Tym różnią się od zmiennych w programach proceduralnych, gdzie są nazwami miejsc pobytu znanych lecz zmieniających się wielkości. Korzystanie ze zmiennych wymaga zadeklarowania dziedzin, z których mogą

[3]Pojedyncze (proste) cudzysłowy nie będą w książce stosowane. Wynika to stąd, że są one w plikach tekstowych (np. PDF-owych) konwertowane na cudzysłowy leksykograficzne (zakrzywione), które są niezrozumiałe dla ECL^iPS^ea, co skutkuje tym, ze skanu PDF-owego pliku programu z takimi cudzysłowami nie można uruchomić.

pochodzić wartości tych zmiennych. Jeżeli zmiennej została przyporządkowana jakaś wartość z jej dziedziny, mówimy że zmienna ta jest *uziemiona*. W przeciwnym przypadku uważa się ją za *zmienną wolną*. Mody zmiennych są obszerniej omawiane w rozdziale 2.1.3. Zdecydowana większość zmiennych w programach prologowych i clp-owych to *zmienne decyzyjne*, tzn. zmienne potrzebne dla sformułowania ograniczeń problemu. W programach mogą również występować *zmienne niedecyzyjne*, tzn. zmienne nie służące do opisu ograniczeń problemu, lecz np. służące do programowania komunikatu generowanego przez program lub do zaprogramowania *akumulatora*, patrz rozdział 2.1.9;

- *liczba* będąca *stałą całkowitą* (np. 9 lub 123) albo *stałą zmiennoprzecinkową* w notacji inżynierskiej (np. 3.14 lub 2.79), wyłącznie z kropką dziesiętną[4];

- *predykat* (po polsku: orzeczenie), będący *relacją* (po polsku: związkiem, zależnością) pomiędzy zmiennymi, np.

  ```
  lubi(Kto, Co).
  ```

  Przykładowy predykat stanowi więc sformalizowaną prefiksową postać potocznego infiksowego zdania

  ```
  "Kto lubi Co",
  ```

  mającą tę przewagę nad potocznym zdaniem, że umożliwia bezproblemowy dostęp do argumentów `Kto` i `Co`, i może łatwo akomodować większą liczbę zmiennych. Ponieważ `Kto` i `Co` są zmiennymi, predykat nie ma wartości logicznej: nie jest ani prawdziwy, ani fałszywy, lecz wieloznaczny. Dopiero *uziemienie* zmiennych `Kto` i `Co`, tzn. zastąpienie ich odpowiednimi stałymi (np. `Kto` = `"Andrzej Kowalski"`, `Co` = `prolog`) sprawia, że predykat zostaje *uziemiony*, tzn. powstaje z niego *stwierdzenie*

  ```
  lubi("Andrzej Kowalski",prolog),
  ```

  które może być prawdą (mówimy wówczas, że predykat jest *spełniony*), lub może nie być prawdą (mówimy wówczas, że predykat jest *niespełniony*, a uziemienie, które do tego doprowadziło, nazywa się *porażką*).

  Należy podkreślić, że argumenty predykatu są *krotką*, tzn. ich kolejność jest istotna i musi być zachowana dla każdego przypadku stosowania tego predykatu w danym programie. Jest to konieczne, gdyż *Prolog* i *CLP* identyfikują zmienne nie po ich nazwie, lecz po ich pozycji w krotce predykatowej. Tak np. predykaty:

  ```
  lubi("Andrzej Kowalski",prolog),
  ```

[4] Przecinki są w *Prolog*u i *CLP* z reguły rezerwowane dla oznaczania operacji koniunkcji.

```
lubi(prolog,"Andrzej Kowalski"),
```

nie powinny występować w tym samym programie, aczkolwiek w (błędnej) intencji programisty mogą oznaczać to samo. Uporządkowanie argumentów "dziedziczy" predykat po relacji, z kolei relacja "dziedziczy" uporządkowanie argumentów po iloczynie kartezjańskim.

W tym miejscu należy wspomnieć o istnieniu *zmiennej anonimowej*, będącej zmienną, której nie potrzeba uziemiać. Jest ona stosowana dla zachowania zadeklarowanej arności predykatów w przypadkach, gdy wartość tej zmiennej jest dla dalszych wnioskowań nieistotna. Np. w przypadku gdy interesuje nas tylko, czy ktoś w jakimś zbiorze studentów w ogóle lubi *Prolog*, możemy posłużyć się zapytaniem ze zmienną anonimową:

```
lubi(_,prolog),
```

na które otrzymamy odpowiedź "yes" lub "no".

Predykaty mogą się *zagnieżdżać*, tzn. mogą służyć jako argumenty innych predykatów, np.:

```
lubi(student_2_go_roku(Kto),przedmiot(Co)).
```

Szczególnym przypadkiem predykatu są *funkcje*[5]; wszystkie funkcje są predykatami, lecz nie wszystkie predykaty są funkcjami. Dlatego w dalszym ciągu w zasadzie wszędzie stosowane będzie pojęcie predykatu, z wyjątkiem przypadku funkcji tradycyjnie zapisywanych w notacji infiksowej (np. X1 + X2, gdzie "+" jest standardowym operatorem dodawania). Tego typu infiksowa notacja jest akceptowana zarówno przez *Prolog* jak i *CLP*.

W dalszym ciągu rozróżnia się:

1. *Predykaty standardowe* (ang. *built-in predicates*), zdefiniowane i zaprojektowane przez twórców *Prolog*u lub języków *CLP* i udostępnione użytkownikom języka. Predykaty standardowe dzieli się na *predykaty elementarne*, definiujące podstawowe relacje i zawarte w bibliote-

[5]Dla zbioru n zmiennych o określonych dziedzinach, funkcja przyporządkowuje unikalną wartość jednej zmiennej (zmiennej wyjściowej) z jej dziedziny każdej (n-1)-krotce pozostałych zmiennych (zmiennych wejściowych) z ich dziedzin.

kach `ic` oraz `branch_and_bound`, o zmiennych zawartych najwyżej w jednej *liście*, oraz *predykaty globalne*, definiujące złożone relacje z bibliotek `ic_global`, `ic_cumulative`, `ic_edge_finder`, `ic_edge_finder3`, których *argumenty* mogą być zawarte w szeregu *listach*.

2. *Predykaty prywatne*, zdefiniowane i zaprojektowane przez twórcę programu prologowego lub clp-owego, o nazwie różnej od nazw predykatów standardowych. Predykaty prywatne mogą zawierać listy.

- *struktura* lub *rekord* (ang. *structure*, skrót *struct*) to złożony typy danych, grupujący powiązane ze sobą termy różnego typu (zwane *argumentami struktury*) w jednym pojęciu, opatrzonym *nazwą* wyglądającą jak atom:

```
nazwa_struktury(arg_1,...,arg_n) .
```

  Argumenty struktury są również *krotką*, tzn. ich kolejność jest istotna i musi być zachowana dla każdego przypadku stosowania tej struktury. Struktury pozwalają w przejrzysty sposób opisywać złożone obiekty. Przykładem struktury może być informacja o programie:

$$\text{program}(ECL^iPS^e, \text{CLP}, \text{"domena publiczna"}).$$

  Struktury umożliwiają przedstawianie i przetwarzanie zagnieżdżających się relacyjnych baz danych. Aczkolwiek struktury pozornie podobne są do predykatów, występuje między nimi zasadnicza różnica: struktury nigdy nie zawierają zmiennych, co znaczy że są zawsze uważane za prawdziwe.

- *lista termów*, rozpoczynająca się nawiasem kwadratowym lewostronnym i kończąca się nawiasem kwadratowym prawostronnym, a zawierająca elementy listy przedzielone przecinkami. W szczególnym przypadku lista może być pusta. Lista termów (niepusta) może wyglądać następująco:

```
[a, b, "CDE", 5, F]
```

  a lista pusta jest zapisywana za pomocą dwóch nawiasów kwadratowych jako `[]`. Jeżeli wszystkie elementy listy są *uziemione*, mówi się iż lista jest *uziemiona*. Właściwościom list poświęcimy więcej uwagi przy okazji dyskusji rekurencji, zob. rozdział 2.1.8.

2.1.2 Programy prologowe i CLP-owe

Program *prolog*owy (i *CLP*-owy) jest deklaracją ograniczeń problemu. Ograniczenia mają postać *klauzul* (po polsku: zdań) zakończonych kropką, będących albo *faktami* albo *regułami*:

1. *Fakty* są strukturami lub predykatami z wszystkimi argumentami uziemionymi, uważanymi za *prawdę*, np.:

   ```
   lubi("Jan",prolog).
   ```

   Brak zmiennych wolnych w fakcie sprawia, że przedstawia wiedzę o charakterze szczegółowym, związanym z konkretną sytuacją.

2. *Reguły* są zdaniami warunkowymi zapisywanymi za pomocą *implikacji prologowych* o budowie:

   ```
   wniosek(_) :-
               warunek_1(_),
               warunek_2(_),
               ...,
               warunek_n(_).
   ```

   Zarówno `wniosek` jak i `warunki` nie są uziemione. Wniosek "`wniosek(_)`" jest prawdą, jeżeli wszystkie warunki reguły (a więc "`warunek_1(_)`", "`warunek_2(_)`",...," `warunek_n(_)`") zostały uziemione i są prawdą. Przecinek jest w regule funktorem *koniunkcji*, czytanym "i". Symbol `:-` jest symbolem strzałki $\leftarrow$ *implikacji prologowej*, czytanym "jeżeli". Powyższą regułę czytamy więc następująco: `wniosek(_)` jest prawdą, jeżeli `warunek_1(_)` i `warunek_2(_)` i...`warunek_n(_)` są prawdą. Obecność zmiennych wolnych we wniosku i warunkach czyni reguły *ogólnymi*, spełnialnymi dla różnych stanów z przestrzeni stanów. Wniosek reguły bywa czasem nazywany *głową* reguły, a koniunkcja warunków - ciałem reguły . Stosowana przy zapisie reguły indentacja ma poprawić czytelność reguły,

   Należy podkreślić, że implikacja prologowa różni się od znanej z logiki implikacji (zob. tablica 2.1) tym, że jeżeli którykolwiek z warunków reguły nie jest prawdą, wniosek uważa się również za nie będący prawdą, zob. tablica 2.2. Założenie to, zwane często *założeniem zamkniętego świata*, ma na celu uniknięcie niejednoznaczności wartości logicznej wniosku.

charakterystycznej dla implikacji logiki. Założenie to jest równoznaczne z założeniem, że w programie prologowym (lub clp-owym) zapisano *całą* wiedzę o przedmiocie wnioskowania.

`Warunek`	`Wniosek`	`Wniosek <- Warunek`
prawda	prawda	prawda
nieprawda	nieprawda	prawda
nieprawda	prawda	prawda

Tablica 2.1: Definicja implikacji w logice

`Warunek`	`Wniosek`	`Wniosek :- Warunek`
prawda	prawda	prawda
nieprawda	nieprawda	prawda

Tablica 2.2: Definicja implikacji w Prologu i CLP

Należy w tym miejscu podkreślić pewne osobliwości nazewnictwa prologowego (i clp-owego). Jest rzeczą oczywistą, że samo słowo `lubi` w predykacie `lubi(Kto, Co)` nic dla *Prolog*u (i *CLP*) nie znaczy. Nic dla *Prolog*u (i *CLP*) nie znaczą również nazwy zmiennych `Kto` i `Co`. Równie dobrze moglibyśmy napisać zamiast

```
lubi(Kto, Co)
```

na przykład

```
bla_bla(Blu_blu, Ble_ble),
```

byle konsekwentnie stosować nazwę predykatu `bla_bla` w całym programie. Z tym, że wtedy moglibyśmy mieć trudności w zrozumieniu, o co w programie chodzi. Dlatego nazwy prologowych predykatów powinny w miarę możliwości dobrze odzwierciedlać ich sens. Co zaś się tyczy nazw zmiennych, to obowiązują one tylko wewnątrz reguły, nie zaś na zewnątrz reguły. Konkretnie:

- w obrębie reguły ta sama nazwa oznacza tą samą zmienną ;
- na zewnątrz reguły *Prolog* (i *CLP*) poznają zmienne nie po nazwie, lecz po ich pozycji w predykatach: jeżeli np. ten sam predykat występuje w kilku regułach, to jego druga (w kolejności) zmienna jest tą samą zmienną, bez względu na to, jak została nazwana. Ma to zarówno konsekwencje pozytywne jak i negatywne:

- ważną pozytywną konsekwencję praktyczną jest to, że "obcy" program prologowy (i clp-owy) możemy "włożyć" do naszego programu nie troszcząc się o uzgodnienie nazw zmiennych, lecz tworząc wyłącznie (prosty) mechanizm jego wywoływania;
- negatywną konsekwencją jest to, że stosując różne (w dodatku jeszcze nieadekwatne) nazwy tych samych zmiennych w różnych regułach i stosując nieadekwatne nazwy predykatów możemy program prologowy (i clp-owy) bardzo skutecznie zachachmęcić i bardzo utrudnić (lub wręcz uniemożliwić) jego zrozumienie.

2.1.3 Mody zmiennych

Termin *mod zmiennej*, pochodzący od łacińskiego *modus* oznaczającego m. in. *sposób*, określa rolę zmiennej będącej argumentem standardowego predykatu, w sytuacji wywołania tego predykatu w programie. Zmienna może być:

- *zmienną wejściową* zwaną często krótko *wejściem*, jeżeli jej wartość będzie określona na zewnątrz predykatu, w którym występuje, np. przez inny predykat lub przez listę, jako atom lub liczba;
- *zmienną wyjściową* zwaną często krótko *wyjściem*, jeżeli jej wartość będzie określona przez predykat, w którym występuje.

W celu uniknięcia błędnego stosowania standardowych predykatów, ich mody są zadeklarowane w dokumentacji ECL^iPS^e za pomocą nazw i symboli:

- *zmienne wejściowe*, które (na mocy deklaracji zawartej w programie) powinny być (w momencie ich wywołania przez program) związane z jakimś predykatem (niekoniecznie uziemionym), z jakąś listą (niekoniecznie uziemioną), z jakimś atomem lub liczbą, są w dokumentacji nazywane *ukonkretnionymi* i opatrzone (w definicji standardowego predykatu) prefiksem `+`, np. `+X`;
- *zmienne wejściowe*, które (na mocy deklaracji zawartej w programie) powinny być (w momencie ich wywołania przez program) związane z jakimś uziemionym predykatem, uziemioną listą, atomem lub liczbą, są w dokumentacji nazywane *uziemionymi* i opatrzone (w definicji standardowego predykatu) prefiksem `++`, np. `++X`;
- *zmienne wyjściowe* są opatrzone (w definicji standardowego predykatu) prefiksem `-`, np. `-X`;

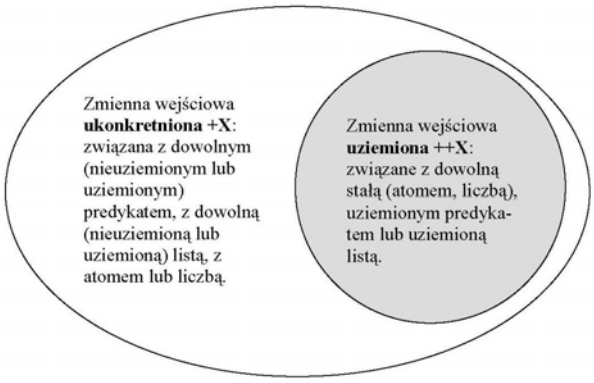
- *zmienne wejściowo-wyjściowe*, mogące być albo wejściami, albo wyjściami (to jedna z osobliwości *Prologu* i *CLP*) są opatrzone (w definicji standardowego predykatu) prefiksem `?`, np. `?X`.

Definicje te zestawiono w tablicy 2.3.

Zmienna ukonkretniona `(+X)`	Zmienna uziemiona `(++X)`	Zmienna wolna `(-X)`	Zmienna dowolnego modu `(?X)`
Wejście związane z dowolnym predykatem, z dowolną listą , z atomem lub liczbą.	Wejście związane z uziemionym predykatem, z uziemioną listą z atomem lub liczbą.	Wyjście nie związane z czymkolwiek.	Wejście lub wyjście.

Tablica 2.3: Mody zmiennych

Lepszemu uwypukleniu różnicy pomiędzy zmiennymi wejściowymi ukonkretnionymi a zmiennymi wejściowymi uziemionymi służy rysunek 2.1. Jak widać, każda *zmienna uziemiona* jest *zmienną ukonkretnioną*, natomiast nie każda *zmienna ukonkretniona* jest *uziemiona*.



Rysunek 2.1: Klasyfikacja zmiennych wejściowych

2.1.4 Operacje

Podstawowe operacje arytmetyczne stosowane w *Prolog*u i *CLP* są przedstawione w tablicy 2.4. Można je stosować w postaci *infiksowej* lub *prefiksowej*, zob. [ECLiPSe Documentation-12]. Tamże można znaleźć zestawienie wszystkich operacji arytmetycznych.

Symbol	Operacja
+	dodawanie
-	odejmowanie
*	mnożenie
/	dzielenie rzeczywiste
//	dzielenie całkowite
mod	modulo
^	potęgowanie

Tablica 2.4: Standardowe operacje arytmetyczne

Standardową kolejność wykonywania operacji (siłę wiązania, precedensy) przedstawia tablica 2.5[6].

Operacja	Siła wiązania	Wartość precedensu
wyrażenia w nawiasach	silna	niska
potęgi i pierwiastki	$\wedge$	$\vert$
mnożenie i dzielenie	$\vert$	$\vee$
dodawania i odejmowania	słaba	wysoka

Tablica 2.5: Standardowa kolejność wykonywania operacji

Strzałki w tablicy 2.5 oznaczają kierunki wzrostu (malenia) siły wiązania (wartości precedensu). Oznaczają one, że jeżeli liczba, symbol lub wyrażenie związane jakimś symbolem operacji jest poniżej pozycji jakiegoś drugiego operatora, lecz powyżej pozycji jakiegoś trzeciego operatora, to operator znajdujący się w tablicy wyżej powinien być najpierw stosowany: tak jak w standardowym *Edinburgh Prolog*u, mniejsza wartość precedensu oznacza, że operator wiąże mocniej (najmocniej dla wartości precedensu równej 1, najsłabiej dla wartości

[6]Ciąg dalszy można pominąć przy pierwszym czytaniu.

precedensu równej 1200)[7]. Użytkownik *Prolog*u lub *CLP* może modyfikować syntaks dynamicznie deklarując nowe operatory o określonym precedensie. Do tego służy predykat standardowy `op/3` o strukturze:

```
op(+Precedens, +Asocjatywność, ++Nazwa)
```

gdzie `Precedens` jest liczbą całkowitą z zakresu od `1` do `1200`, `Nazwa` jest operatorem o wskazanym precedensie, a `Asocjatywność` jest argumentem deklarującym różne klasy operatorów. Wprowadzając oznaczenia:

f - operator o zadeklarowanym precedensie,

x - argument o precedensie ściśle mniejszym aniżeli precedens operatora

y - argument o precedensie mniejszym lub równym precedensowi operatora,

i zakładając w dalszym ciągu, że:
- argumenty w nawiasach lub argumenty bez operatorów mają precedens `0`,
- argumenty z operatorami mają precedens równy precedensowi operatora,

można zestawić możliwe klasy operatorów i ich asocjatywność jak to pokazano w tablicy 2.6.

Klasa operatorów	Asocjatywność
prefiks	fx, fy (unarny) lub fxx, fxy (binarny)
infiks	xfx, xfy, yfx
postfiks	xf, yf

Tablica 2.6: Klasy operatorów i ich asocjatywność

Wymienione pojęcia ilustruje następujący prosty przykład:
rozpatrzmy wyrażenie

`u - v - w`,

gdzie operator "`-`" ma precedens 500. Chcemy rozpatrywane wyrażenie interpretować jako:

`(u - v) - w`,

nie zaś jako:

`u - (v - w)`.

Aby uzyskać żądaną interpretację, operator "`-`" powinien mieć asocjatywność `yfx`. Bardziej zaawansowany przykład można znaleźć w rozdziale 5.8.3.

[7]Terminologia ta jest niezbyt fortunna, gdyż większa wartość precedensu w Prologu (i w CLP) jest tożsama z niższą wartością precedensu w języku potocznym.

2.1.5 Propagacja ograniczeń

Propagacja ograniczenia jest procesem zapoczątkowanym uziemieniem jakiejś zmiennej ograniczenia na wartości z jej dziedziny. Jej celem jest "powiadomienie" o dokonanym uziemieniu wszystkich elementów programu tego potrzebujących i dokonaniu zmian wynikających z rzeczonego uziemienia. W Prologu propagacja ograniczenia obejmuje dwa etapy:

1. Propagacja wartości zmiennej.
2. Unifikacja.

Propagacja wartości zmiennej polega na przekazaniu wartości uziemionej zmiennej wszystkim instancjom tej zmiennej w ciele reguły, w którym nastąpiło to uziemienie, oraz instancjom tej zmiennej w ciałach innych reguł.
Unifikacja jest procesem dopasowywania wartości innych instancji uziemionej zmiennej do wartości uziemionej zmiennej w celu uzyskania równości dwóch termów. Zasady unifikacji można sformułować następująco:

- w dziedzinie Herbranda unifikowalne są termy *syntaktycznie równoważne*. Dwa termy są *syntaktycznie równoważne*, jeżeli:
  - są tego samego typu i mają tą samą budowę, np.
    "lubi(A,B)" i "lubi(X,Y)",
    albo "[X,Y,Z]" i "[P,_,R]",
    lub jeżeli:
  - jeden z unifikowanych termów jest *zmienną wolną*;
- *zmienne wolne* można unifikować z czymkolwiek, również z innymi *zmiennymi wolnymi*. Jest to w pełni zgodne z wzmiankowaną już właściwością, iż nazwy zmiennych "mają znaczenie lokalne", tylko w regułach, w których występują;
- różne atomy nie są unifikowalne;
- różne liczby nie są unifikowalne.

Unifikację powoduje predykat standardowy `=/2`. Np. unifikacja:

```
lubi(Kto, Co) = lubi("Jan",prolog)
```

jest wykonalna i daje

```
Kto = "Jan"
Co = prolog
```

W *Prolog*u (i w *CLP*) nieprawdą jest więc `1 + 1 = 2`, bo mamy tu do czynienie

z termami syntaktycznie nierównoważnymi. Możemy zaś posłużyć się słowem kluczowym `is`, zapisując odpowiednią zależność w postaci `1 + 1 is 2`.

Oczywiście, nie wszystkie termy syntaktycznie równoważne są unifikowalne. Np.:

`lubi(Kto, pascal) = lubi(`Jan",prolog)

jest nieunifikowalne (nie jest prawdą), gdyż różne stałe `pascal` i `prolog` nie są unifikowalne. To samo jest słuszne dla:

`lubi("Piotr", prolog) = lubi("Jan",prolog),`

gdyż stałe `"Piotr"` i `"Jan"` nie są unifikowalne. Stosowanie symbolu równości (`=`) jest dopuszczalne tylko pomiędzy termami unifikowalnymi, np.:

```
X = Y
X = 5
```

Wynik unifikacji może być dwojaki:

1. Jeżeli unifikacja kończy się sukcesem, następna zmienna wolna zostaje uziemiona.

2. Jeżeli unifikacja kończy się porażką, ostatnia uziemiona zmienna zostaje odziemiona, co inicjuje *nawrót*.

2.1.6 Poszukiwania w drzewach, których nie ma

Dla dalszych rozważań pożyteczne będą pojęcia *całkowitego stanu*, *przestrzeni stanu*, *dopuszczalnego stanu* i *częściowego stanu*:

- *Całkowity stan*, w dalszym ciągu po prostu nazywany *stanem*, to dowolne uziemienie wszystkich *zmiennych decyzyjnych*.

- *Przestrzeń stanu* oznacza wszystkie możliwe uziemienia wszystkich zmiennych decyzyjnych.

- *Dopuszczalny stan* to taki stan, który spełnia wszystkie ograniczenia problemu.

- *Częściowy stan* to dowolne uziemienie części zmiennych decyzyjnych.

Wymienione pojęcia nie są zbyt popularne w środowiskach badaczy i twórców *Prolog*u i *CLP*. Autor książki, ze względu na swe wykształcenie i długoletnią pracę w obszarze teorii i techniki sterowania, odczuwał od dawna brak owych pojęć w *Prolog*u i *CLP*. Pisanie tej książki jest wspaniałą okazją by ten brak

uzupełnić.

Celem programu prologowego lub clp-owego jest *spełnienie* pewnego *zapytania*, tzn. uczynienie tego zapytania (którym jest zawsze wniosek jakiejś reguły) prawdą na drodze wszystkich możliwych uziemień wszystkich zmiennych, od których zapytanie zależy. Dokonuje tego *system wnioskujący* będący częścią kompilator *Prolog*u lub *CLP*. Istota jego działania polega na poszukiwaniu w przestrzeni stanu takiego stanu dopuszczalnego, który uczyni zapytanie prawdą. W szczególności system wnioskujący działa na zasadzie:

- dynamicznego konstruowania *drzewa poszukiwań* odpowiadającego programowi prologowemu (lub clp-owemu), z równoczesnym *przeszukiwaniem* tego drzewa za pomocą *algorytmu poszukiwań w głąb* (zwanego również algorytmem poszukiwań w głąb z nawrotami lub czasem krótko algorytmem nawrotów). Istotą poszukiwań jest uziemianie zmiennych na wartościach z ich dziedzin;

- *propagacji ograniczeń*, której istotą jest uwzględnienie wpływu dokonanego uziemienia zmiennej na predykaty problemu. Jeżeli ostatnie uziemienie doprowadza do porażki jakiegokolwiek predykatu, następuje *nawrót*, polegający na tym, że odpowiednia zmienna odpowiedzialne za *porażkę* zostaje *odziemiona* (staje się ponownie zmienną wolną), a poszukiwania są kontynuowane powyżej tego punktu w drzewie poszukiwań, w którym dokonano uprzedniego uziemienia. Poszukiwania te mogą doprowadzić do ponownego uziemienia (*reuziemienia*) zmiennej. Jeżeli ostatnie uziemienie nie skutkuje porażką, kontynuuje się poszukiwania uziemiając kolejną zmienną.

Ilustruje to następujący prosty program (`2_1_poszukiwania.pl`) w języku *Prolog*:

```
/*1*/  a(X,Y) :-
/*2*/       b(X),
/*3*/       c(X,Y).
/*4*/  b(1).
/*5*/  b(&).
/*6*/  c(&,"A").
```

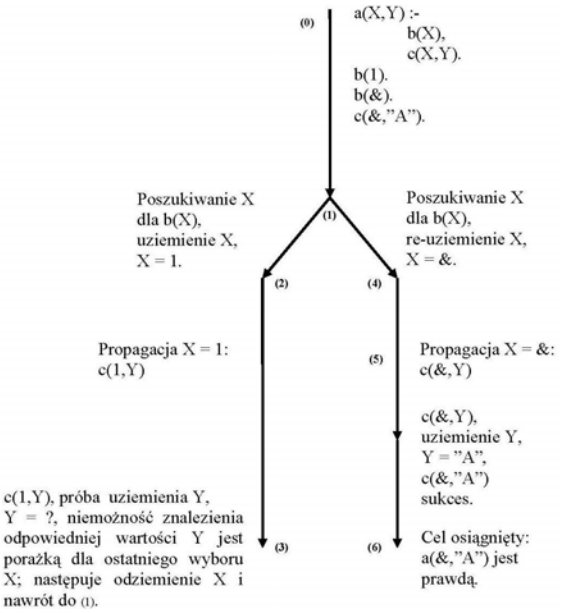
Program ten zawiera jedną regułę (linie 1, 2 i 3) oraz trzy fakty (linie 4, 5 i 6). Dziedziny zmiennych `X` i `Y` zostały w tym programie (jak we wszystkich programach prologowych) zadeklarowane *implicite*, w liniach 4, 5 i 6: dziedziną zmiennej `X` jest `(1,&)`, dziedziną zmiennej `Y` jest `"A"`. Zapytaniem niech będzie `a(X,Y)`. A więc celem programu jest znalezienie takich wartości zmiennych X i Y, by `a(X,Y)` było prawdą. Reguła (linie 1, 2 i 3) stwierdza, że `a(X,Y)` będzie prawdą, jeżeli znajdzie się takie `X`, dla którego `b(X)` będzie prawdą, i takie `Y` które łącznie z uziemioną już zmienną `X` sprawi, że `c(X,Y)` będzie prawdą. Zgodnie z liniami 2 i 3 nastąpi to, gdy uziemi się zmienne `X` i `Y` tak, by `b(X)` i `c(X,Y)` było prawdą. Idąc z góry programu w dół kompilator szuka najpierw takiej wartości `X`, dla której `b(X)` jest prawdą. W linii 4 stwierdza się, że wartością tą jest `X = 1`, tzn. zmienna `X` zostaje uziemiona na wartości `1`. Uziemienie to zostaje propagowane dla wartości `X` z linii 3. Przechodząc do linii 3 i uwzględniając propagację dla `X`, szuka się z kolei takiej wartości `Y`, by `c(1,Y)` było prawdą. Niestety, odpowiedniego faktu brak w programie; a więc poprzednie uziemienie dokonane dla `X` jest *porażką*. Dlatego `X` zostaje odziemione, wykonuje się nawrót do linii 2 i szuka innej wartości `X` takiej, by `b(X)` było prawdą. W linii 5 stwierdza się, że wartością tą jest `X = &`, tzn. zmienna `X` zostaje *reuziemiona* na wartości `&`. Przechodząc ponownie do linii 3 i pamiętając ostatnie uziemienie dla `X`, szuka się z kolei takiej wartości `Y`, by `c(@,Y)` było prawdą. Z linii 6 wynika, że wartością tą jest `"A"`. Zmienna `Y` zostaje więc *reuziemiona* jako `Y = "A"`, czyli `a(X,Y)` jest prawdą dla `X = &` i `Y = "A"`, co kończy program.

Można uważać, że opisane poszukiwania odbywają się one zgodnie z drzewem pokazanym na rysunku 2.2.

Ze względów oczywistych można więc rozważane poszukiwania nazywać *poszukiwaniami w głąb* owego drzewa. Stosowany zaś sposób generowania nawrotów nazywa się *standardowym*. Stąd pełna nazwa rozważanych poszukiwań: *poszukiwania w głąb ze standardowymi nawrotami*.

Drzewo z rysunku 2.2 nosi nazwę *drzewa poszukiwań*. Jest ono definiowane przez wszystkie punkty przestrzeni stanu programu w języku *Prolog* (lub w *CLP*). Stanowi graficzny obraz przebiegu poszukiwań. Punkt (1) drzewa poszukiwań jest przykładem *punktu wyboru*, gdyż w nim dokonuje się wyboru kolejnych możliwych wartości zmiennej `X` i do tego punktu dokonywany jest nawrót w przypadku porażki. Z przedstawionego przykładu wynika, że drzewo poszukiwań jest tworzone *dynamicznie*, gałąź po gałęzi, w miarę wykonywania kolejnych kroków programu.

Można również zauważyć, że nie ma potrzeby (a praktycznie bardzo często - możliwości) przechowywania całego tworzonego drzewa poszukiwań w pamięci



Rysunek 2.2: Drzewo poszukiwań w głąb ze standardowymi nawrotami dla prostego programu prologowego

operacyjnej komputera: należy przechowywać tylko analizowaną gałąź oraz te punkty wyboru, które odpowiadają wartościom zmiennych jeszcze nie testowanych. Właściwość ta leży u podstaw zalet *Prolog*u i *CLP*: drzewa poszukiwań dla realistycznych problemów mają iście kosmiczne rozmiary i ewentualna potrzeba ich zapisywania "w całości" przed rozpoczęciem poszukiwań byłaby trudnością nie do pokonania. To właśnie mamy na myśli pisząc o drzewach poszukiwań, *których nie ma*.

Przykład ilustruje również pewną ogólną prawidłowość, zgodnie z którą:

- *uziemienie zmiennej* jest elementem *poszukiwań w głąb* drzewa poszukiwań i następuje w sytuacji dotarcia do predykatu, zawierającego nieuziemione zmienne;
- *odziemienie zmiennej* jest wynikiem *porażki* poniesionej dla propagacji następującej po uziemieniu: ponieważ dla wzmiankowanego uziemienia nie

jest możliwa żadna propagacja (tzn. nie może być spełnione żadne ograniczenie), to zmienna zostaje odziemiona i następuje nawrót (w górę drzewa poszukiwań), do tego najbliższego punktu wyboru, dla którego istnieją jeszcze nie testowane wartości odziemionej zmiennej.

Podkreśla się, że w *Prolog*u i *CLP* zmienna może zmienić swą wartość jedynie w wyniku nawrotu do punktu wyboru (lub poza punkt wyboru), w którym została jej przyporządkowana ta wartość.

2.1.7 Pożyteczne porażki

Już wiemy, że nawrót jest inicjowany w wyniku porażki uziemionego predykatu. Teraz dowiemy się, że w pewnych sytuacjach warto doprowadzić do porażki, i do tego służy specjalny standardowy predykat `fail/0`, który jest zawsze stałą logiczną nieprawdziwą. Predykat ten umożliwia generowanie nawrotów dzięki którym zostaną wyznaczone wielokrotne rozwiązania programu prologowego, jeżeli one istnieją. Przedstawia to program `2_2_fail.pl`:

```
/*1*/  top:-
/*2a*/      twoi_przyjaciele_1.
/*2b*/      %   twoi_przyjaciele_2.
/*2c*/      %   twoi_przyjaciele_3.

/*3*/  przyjaciel("Marek").
/*4*/  przyjaciel("Jacek").
/*5*/  przyjaciel("Andrzej").

% Dla 'twoi_przyjaciele_1' nie ma nawrotów.
% Program wyznacza jedno rozwiązanie (pierwsze z góry):
/*6*/  twoi_przyjaciele_1:-
/*7*/  przyjaciel(Kto),
/*8*/  write("Przyjaciel:  "),write(Kto),  nl.

% Dla 'twoi_przyjaciele_2', 'fail' generuje nawroty,
% dzięki którym wszyscy przyjaciele zostają wymienieni,
% ale program kończy się porażką 'No':
/*9*/  twoi_przyjaciele_2:-
/*10*/       przyjaciel(Kto),
/*11*/        write("Przyjaciel:  "),write(Kto),nl,
/*12*/       fail.

% Dla 'twoi_przyjaciele_3', 'fail' generuje nawroty,
% dzięki którym wszyscy przyjaciele zostają wymienieni.
% Gdy już nie można dokonać nawrotu, program przechodzi
% do drugiej reguły z wnioskiem 'twoi_przyjaciele_3', co
```

```
% kończy program sukcesem  'Yes':
/*13*/ twoi_przyjaciele_3:-
/*14*/       przyjaciel(Kto),
/*15*/       write("Przyjaciel:  "),write(Kto), nl,
/*16*/       fail.

/*17*/ twoi_przyjaciele_3:-
/*18*/       write("To wszyscy twoi przyjaciele."),nl.
```

Komunikat dla `twoi_przyjaciele_1`:

```
Przyjaciel:  Marek.
Yes.
```

Klikając dwukrotnie w przycisk "more" w Głównym Menu z rysunku 1.5 w celu wymuszenia pozostałych rozwiązań, pojawia się komunikat:

```
Przyjaciel:  Jacek
Yes.
Przyjaciel:  Andrzej
Yes.
```

Komunikat dla `twoi_przyjaciele_2`:

```
Przyjaciel:  Marek
Przyjaciel:  Jacek
Przyjaciel:  Andrzej
No.   % Niestety, ten program kończy się porażką.
```

Komunikat dla `twoi_przyjaciele_3`:

```
Przyjaciel:  Marek
Przyjaciel:  Jacek
Przyjaciel:  Andrzej
To wszyscy twoi przyjaciele.
Yes.  % Hurra, ten program kończy się sukcesem dzięki liniom 17 i 18.
```

W tym miejscu należy zauważyć, że predykat standardowy `fail/0` jest jawnie proceduralnym operatorem, którego potrzeba demonstruje niemożność pisania czysto deklaratywnych programów, pozbawionych proceduralnych "protez". Dobrze jest również zapamiętać, że `fail/0` można zastąpić dowolnym zawsze nieprawdziwym predykatem (np. `1 is 2`) lub predykatem, który aktualnie nie jest spełniony. Ta ostatnia możliwość - jak zobaczymy w dalszym ciągu - jest

stosowana w programach prologowych i clp-owych na masową skalę właśnie dla generacji nawrotów.

2.1.8 Rekurencje

Rekurencyjna definicja predykatu składa się z reguły i faktu, lub inaczej, z właściwej rekurencji i warunku początkowego (końcowego):

- dla *właściwej rekurencji* definiowany predykat stanowi głowę reguły, której ciało zawiera tenże predykat z odmienną strukturą argumentów;
- *warunek początkowy (końcowy)* jest zapisany najczęściej za pomocą tego samego predykatu w postaci uziemionej.

Zwięzłość, deklaratywność i moc *Prolog*u (i *CLP*) ma m.in. swe źródło w powszechnym stosowaniu definicji rekurencyjnych. Przykładem może być definicja elementu listy. Listy są w *Prolog*u najczęściej stosowanym sposobem przedstawiania danych. Listę prologową zapisuje się w następujący sposób:

```
Lista prologowa = [element_1, element_2,...,element_n]
```

i uważa się ją za twór o następującej strukturze:

```
Lista prologowa = [Głowa_listy|Ogon_listy]
```

gdzie `Głowa_listy` jest pierwszym elementem listy, zaś `Ogon_listy` jest listą pozostałą po usunięciu `Głowy_listy`. Ponieważ `Ogon_listy` jest listą, w takim razie można go zdekomponować na pierwszy element (`Głowa_ogona_listy`) i pozostałą listę (`Ogon_ogona_listy`), itd., aż do uzyskania ogona w postaci listy pustej, zapisywane w postaci `[]`. Spróbujmy zdefiniować pojęcie elementu listy. Można to zrobić definiując predykat:

```
element_listy(Element_listy,Lista)
```

za pomocą faktu (1) i reguły (2):

```
/*1*/      element_listy(G,[G|_]).
/*2*/      element_listy(E,[_|O]) :-
/*3*/                element_listy(E,O).
```

Fakt `(1)` stwierdza, że głowa listy jest elementem listy, niezależnie od tego, czym jest ogon: podkreślnik `_` oznacza *zmienną anonimową*, tzn. zmienną, której wartość jest dla nas nieinteresująca. Reguła `(2)` - `(3)` stwierdza, że elementem listy jest element ogona tej listy. Popatrzmy na funkcjonowanie tej definicji, ilustrowane programem `2_3_lista.pl`:

```
/*1*/  top:-
/*2*/       element_listy(E,[1,2,3,4]),
/*3*/       writeln(E),
/*4*/       fail.
/*5*/  top:-
/*6*/       writeln("To wszystkie elementy listy.").

/*7*/  element_listy(G,[G|_]).
/*8*/       element_listy(E,[_|O]) :-
/*9*/  element_listy(E,O).
```

Program generuje następujący komunikat:

```
1
2
3
4
To wszystkie elementy listy.
```

Zapytaniem programu jest (i tak będzie zawsze!) `top`. *Prolog* stara się uczynić `top` prawdą na drodze uziemiania zmiennych. W tym celu musi uziemić predykat z linii `2`. Aby tego dokonać wywołuje definicję z linii `/*7*/`, zmienna `G` zostaje uziemiona na wartości `1` i wypisana, a ze względu na pozostałą część definicji (linie `/*8*/` i `/*9*/`) powstaje punkt wyboru dla predykatu `element_listy/2`, do którego generuje nawrót predykat `fail/0`. W punkcie tym korzysta się z rekurencyjnej części definicji (linie `/*8*/` i `/*9*/`), wyznaczając kolejny element listy jako pierwszy element ogona `O = [2,3,4]`, itd. Kolejne usuwania głów listy dokonywane w liniach `/*8*/`, `/*9*/` i `/*7*/` doprowadza do listy pustej, co kończy rekurencję.
W przypadku platformy ECL^iPS^e użytkownik nie musi definiować predykatu `element_listy/2`, gdyż jest on dany w postaci predykatu standardowego `member/2`.

2.1.9 Operacje na listach

W *Prolog*u i w *CLP* możliwe są tylko dwie rekurencyjne operacje na listach:

1. Usuwanie kolejnych głów listy i przekazywanie ich do jakiegoś ograniczenia, aż do uzyskania listy pustej. Ilustruje to następujący przykład:

```
predykat_rekurencyjny([G|O],....):-
        % Teraz usuwa się i przetwarza głowę listy:
        przetwarza_się_głowę(G),
        ........................
        predykat_rekurencyjny(O,....).


        % Usuwanie głów doprowadza do listy pustej:
predykat_rekurencyjny([],....).
```

   Rekurencja z usuwaniem głów rozpoczyna się listą pełną, której głowy są kolejno usuwane i stosowane do jakiś poszukiwań, aż do uzyskania listy pustej. Rekurencja ta zachodzi pomiędzy listą (w głowie reguły) a ogonem tej listy (w rekurowanym predykacie w ciele reguły).

2. Rekurencja z dokładaniem głów: uzyskane z jakiegoś ograniczenia uziemione zmienne są wprowadzane jako kolejne głowy do listy, aż listy uzyska wymagana postać końcową. Ilustruje to następujący przykład:

```
predykat_rekurencyjny(Lista,....):-
        % Teraz wyznacza się i dokłada głowę listy:
        wyznacza_się_głowę(G),
        ........................
        predykat_rekurencyjny([G|Lista],....).


        % Dokładanie głów doprowadza do listy o wyróżnionej postaci:
predykat_rekurencyjny(Wyróżniona_postać_listy,,....).
```

   Rekurencja z dokładaniem głów rozpoczyna się listą pustą lub częściowo zapełnioną, do której są kolejno dokładane głowy uzyskane w wyniku jakiegoś poszukiwania, aż do uzyskania listy o wyróżnionej postaci. Rekurencja ta zachodzi pomiędzy ogonem listy (w głowie reguły) a uzupełnioną o głowę listą (w rekurowanym predykacie w ciele reguły).

Należy pamiętać, że z listy można usuwać (i do listy można dokładać) tylko głowy. Wszystkie inne operacje na listach można wykonać korzystając z wymienionych dwóch.

Obydwie operacje ilustruje program `2_4_odwracanie.pl`, dokonujący odwracania kolejności elementów w liście za pomocą dwóch prywatnych predykatów:

1. `odwracanie(Lista_poczatkowa, Lista_odwrotna)`

2. `odwracanie(Lista_poczatkowa, Lista_odwrotna,`
   `Akumulator_listy_odwroconej)`

Predykaty te mają jednakowe nazwy, lecz różnią się *arnością*, dzięki czemu kompilator potrafi je odróżnić. *Akumulatorem* nazywa się pierwotnie pustą listę, do której dodawane są kolejne *głowy*. Program `2_4_odwracanie.pl` jest postaci:

```
/*1*/      top:-
/*2*/               odwracanie([a,b,c,d],Lista_odwrotna),
/*3*/               write("Lista odwrotna = "),write(Lista_odwrotna).

/*4*/      odwracanie(Lista_poczatkowa,Lista_odwrotna):-
/*5*/               odwracanie(Lista_poczatkowa,Lista_odwrotna,[]).
/*6*/      odwracanie([],B,B).

/*7*/      odwracanie([H|T],Lista_odwrotna,A):-
/*8*/               odwracanie(T,Lista_odwrotna,[H|A]).
```

Program generuje komunikat:

```
Lista odwrotna = [d, c, b, a]
```

Program (w liniach `/*7*/` i `/*8*/`) usuwa kolejne głowy `H` z `Listy_poczatkowej` (zapisanej jako `[H|T]`) i kolejno dokłada te głowy (jako głowy) do (pierwotnie pustej) listy-akumulatora o nazwie `A`), co daje listę `[H|A]`. Kiedy `Lista_poczatkowa` stanie się listą pustą (tzn. gdy wszystkie jej elementy zostały przeniesione w odwrotnej kolejności do listy-akumulatora (co dokonało się w linii `/*6*/`), wtedy `Lista_odwrotna` (drugi argument predykatu z linii `/*6*/`) jest unifikowana z listą-akumulatorem (trzeci argument predykatu z linii `/*6*/`), co daje rozwiązanie. Wzmiankowana unifikacja jest dokonywana przez stosowanie tej samej nazwy zmiennej (`B`) dla `Listy_odwrotnej` i `listy-akumulatora` (linia `/*6*/`), co jest pięknym przykładem korzystania z "liberalizmu nazewniczego" *Prolog*u.

W przypadku platformy ECL^iPS^e użytkownik nie musi definiować predykatu `odwracanie/2`, gdyż istnieje predykat standardowy `reverse/2`, działający dokładnie tak jak `odwracanie`.

Idea akumulatora wymaga kilku komentarzy. W *Prolog*u i *CLP* *akumulatorami* są *zmienne niedecyzyjne*[8] umożliwiające pisanie tzw. reguł z *rekurencją ogonową*, tzn. takich reguł, w których głowa reguły wywołuje siebie na samym końcu ciała reguły, zob. prosty przykład dany liniami `/*7*/` i `/*8*/`. Rekurencja ogonowa jest najbardziej oszczędną (z punktu widzenia użycia przestrzeni na stosie), gdyż nie wymaga ona stosu.

2.1.10 Generowanie list

Praktycznie wszystko, co można zrobić za pomocą *Prolog*u i *CLP*, wymaga stosowania list. Listy najczęściej są generowane ze zbiorów danych, często za pomocą predykatu standardowego *findall/3* o następującej strukturze:

```
findall(?Term, +Cel, -Lista)
```

gdzie `Lista` jest listą wszystkich tych wartości zmiennej `Term` które spełniają `Cel`. Rozpatrzmy następujący przykład:

Firma *Backyard Used Car* rozpoczyna atrakcyjną wyprzedaż następujących używanych, lecz niezbyt starych i dobrze utrzymanych samochodów produkcji znanej firmy *Clunker Motors Company*: *Clunker SUV*, *Clunker Great Tour*, *Clunkerlac*, *Clunker Family*, *Clunkerdes*,*Clunker Electric* i *Clunker Green*. Szczegóły przedstawia tabela 2.7.

Należy wyznaczyć średnią cenę wszystkich samochodów i średni przebieg samochodów kosztujących mniej niż 1900, o kolorze różnym od czerwonego i nie starszy niż rocznik 2006. W tym celu definiuje się strukturę:

```
oferta(Model, Przebieg, Rok, Cena, Kolor),
```

służącą do zadeklarowania relacyjnej bazy danych o samochodach, koniecznej dla rozwiązania przykładu. Program `2_5_graty.pl` przedstawia zastosowanie predykatu standardowego *findall/3* dla wyznaczenia wymienionych danych:

[8] Niedecyzyjne w tym sensie, że nie odpowiadają żadnej z pierwotnych zmiennych opisujących problem.

Model	Przebieg	Rocznik	Cena w JP	Kolor
Clunker SUV	29000	2008	1500	Czarny
Clunker Great Tour	60000	2009	1900	Zielony
Clunkerlac	47000	2007	1200	Bialy
Clunker Family	38000	2009	2200	Granatowy
Clunkerdes	46000	2008	3100	Srebrny
Clunker Electric	75000	2005	1100	Czerwony
Clunker Green	52000	2006	1300	Srebrny

Tablica 2.7: Dane używanych samochodów

```
/*1*/  top:-
/*2*/      srednia_cena_wszystkich_samochodow,
/*3*/      sredni_przebieg_wybranych_samochodow.

/*4*/  srednia_cena_wszystkich_samochodow:-
/*5*/      findall(Cena,oferta(_,_,_,Cena,_),Lista),
/*6*/      writeln("Lista cen wszystkich samochodow":Lista),
/*7*/      length(Lista, N),
/*8*/      Sum is sum(Lista),
/*9*/      Srednia_Cena is Sum/N,
/*10*/     writeln("Srednia cena":Srednia_Cena),nl.

/*11*/ sredni_przebieg_wybranych_samochodow:-
/*12*/     findall(Przebieg,wybrane_samochody(Przebieg),Lista),
/*13*/     writeln("Lista przebiegow samochodow kosztujacych mniej niz 1900, "),
/*14*/     writeln("o kolorze nie-czerwonym i młodszych niz rocznik 2006":Lista),
/*15*/     length(Lista, N),
/*16*/     Sum is sum(Lista),
/*17*/     Sredni_Przebieg is Sum/N,
/*18*/     writeln("Sredni przebieg samochodow kosztujacych mniej niz 1900,"),
/*19*/     writeln("o kolorze nie-czerwonym i młodszych niz rocznik 2006 ":
                    Sredni_Przebieg).

/*20*/ wybrane_samochody(Przebieg):-
/*21*/     oferta(_,Przebieg,Rocznik,Cena,Kolor),
/*22*/     Rocznik > 2006,
/*23*/     Cena < 1900,
/*24*/     Kolor \== "Czerwony".

/*25*/ oferta("Clunker SUV", 29000, 2008, 1500, "Czarny").
/*26*/ oferta("Clunker Great Tour", 60000, 2009, 1900, "Zielony").
/*27*/ oferta("Clunkerlac", 47000, 2007, 1200, "Bialy").
```

```
/*28*/ oferta("Clunker Family", 38000 , 2009, 2200, "Niebieski").
/*29*/ oferta("Clunkerdes", 46000, 2008, 3100, "Srebrny").
/*30*/ oferta("Clunker Electric", 75000, 2005, 1100, "Czerwony").
/*31*/ oferta("Clunker Green", 52000, 2006, 1300, "Srebrny").
```

Odpowiedź programu jest następująca::

```
Lista cen wszystkich samochodow : [1500, 1900, 1200, 2200, 3100, 1100, 1300]
Srednia cena samochodow : 1757.14285714286

Lista przebiegow samochodow kosztujacych mniej niz 1900 i
o kolorze nie-czerwonym i nie starszych niz rocznik 2006 : [29000, 47000]
Sredni przebieg samochodow kosztujacych mniej niz 1900,
o kolorze nie-czerwonym i nie starszych niz rocznik 2006  : 38000.0
```

Obecność zmiennych anonimowych w linii `5` wynika stąd, że spośród wszystkich pięciu argumentów struktury `oferta/5` chcemy zrobić listę wartości tylko jednego z nich, a mianowicie *Ceny*.

Czytelnicy znający języki bazo-danowe z pewnością zauważą, ze *Prolog* jest również eleganckim językiem dla formułowania zapytań do baz danych, o możliwościach odpowiadających silnemu podzbiorowi *SQL*.

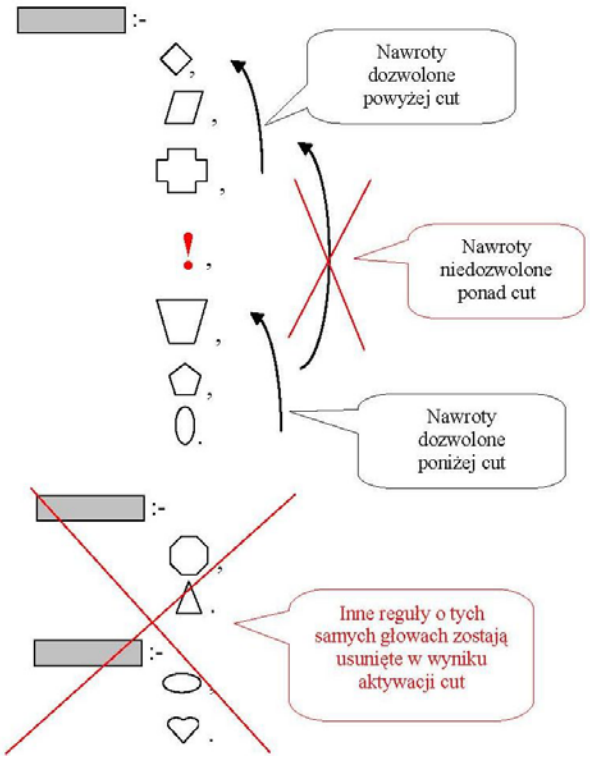
2.1.11 Sterowanie nawrotami za pomocą "cut"

Nawroty w *Prolog*u (i *CLP*) są "automatyczne", tzn. występują zawsze w wyniku porażki, o ile istnieją punkty wyboru. Jest to zarówno korzystne, jak i niekorzystne:

- korzyścią bezsprzeczną jest to, że programista prologowy (i clp-owy) nie musi programować nawrotów w swoich programach;
- niekorzystnym może się okazać to, że w wyniku nawrotów mogą pojawić się nie interesujące nas rozwiązania cząstkowe, co przedłuża czas działania programu.

Można temu ostatniemu zaradzić korzystając z predykatu standardowego `!/0` noszącego angielską nazwę *cut* (po polsku *wytnij*), czytaj *kat*. Właściwości tego predykatu przedstawiono na rysunku 2.3, gdzie czarne strzałki oznaczają możliwe nawroty.

W szczególności:



Rysunek 2.3: Właściwości predykatu cut (!/0)

- predykat *cut* jest zawsze spełniony;
- wywołanie predykatu *cut* uniemożliwia jednak dokonania nawrotu, w obrębie danej reguły, poza miejsce w którym predykat *cut* się znajduje;
- wywołanie predykatu *cut* uniemożliwia spełnienie następnych reguł dla tego samego wniosku co reguła, w której został wywołany *cut*.

Program `2_6_zabawa_z_cut.pl` ilustruje wszystkie właściwości predykatu `cut/0`:

```
/*1*/  top:-
/*2*/       a.
```

```
/*3*/  a :-
/*4*/      b, write("   Kompilator Prologu nie wywolal 'cut'.  "),nl,
/*5*/      write("   Dzieki temu, poniewaz pierwszej reguly dla 'b'"),nl,
/*6*/      write("   nie udalo sie spelnic, 'a' jest prawdziwe dzieki "),nl,
/*7*/      write("   spelnieniu drugiej reguly dla 'b'."),nl,nl.

/*8*/  a :-
/*9*/      write("Jestem tu 4!"),nl,
/*10*/     write("   Poniewaz kompilator Prologu wywolal 'cut' w"),nl,
/*11*/     write("   pierwszej regule dla 'b', nie mozna juz przejsc do "),nl,
/*12*/     write("   drugiej reguly dla 'b'. Pierwsza "),nl,
/*13*/     write("   regula dla 'a' nie zostanie wiec spelniona. "),nl,
/*14*/     write("   Zostanie jednak spelniona druga regula dla 'a',"),nl,
/*15*/     write("   gdyż przejście do niej jest dopuszczalne."),nl,nl.

/*16*/ b :-
/*17*/     c(X),
/*18*/     d(X),
/*19*/     !,
/*20*/     e(Y),
/*21*/     f(Y),
/*22*/     g(X).

/*23*/ b:-
/*24*/     write(" Jestem tu 0!"),nl.

/*25*/ c(1).

/*26*/ c(2):-
/*27*/     write("Jestem tu 1!"),nl,
/*28*/     write("   Nawrot jest mozliwy przed 'cut'."),nl,nl.
/*29*/%    Na przemian usuń i wprowadź zdanie dla c(2),
/*30*/%     analizując wyniki.

/*31*/ c(3).

/*32*/ d(2).

/*33*/ e(1).

/*34*/ e(2) :-
/*35*/     write("Jestem tu 2!"),nl,
/*36*/     write("   Nawrot jest mozliwy za 'cut'."),nl,nl.

/*37*/ f(2) :-
/*38*/     write("Jestem tu 3!"),nl,
/*39*/     write("   Niestety, nawrot nie jest mozliwy "),nl,
/*40*/     write("   zza miejsca umieszczenia 'cut' w pierwszej "),nl,
/*41*/     write("   regule dla 'b' przed miejsce umieszczenia"),nl,
```

```
/*42*/     write("   'cut' w tejze regule."),nl,nl.

/*43*/ g(3).
```

Komunikaty w sytuacji, gdy linie 26, 27 i 28 zostały odkomentowane, są następujące:

```
Jestem tu 1!
   Nawrot jest mozliwy przed 'cut'.

Jestem tu 2!
   Nawrot jest mozliwy za 'cut'.

Jestem tu 3!
   Niestety, nawrot nie jest mozliwy
   zza miejsca umieszczenia 'cut' w pierwszej
   reguly dla 'b' przed miejsce umieszczenia
   'cut' w tejze reguly.

Jestem tu 4!
   Poniewaz kompilator Prologu wywolal 'cut' w
   pierwszej regule dla 'b', nie mozna juz przejsc do
   drugiej reguly dla 'b'. Pierwsza
   regula dla 'a' nie zostanie wiec spelniona.
   Zostanie jednak spelniona druga regula dla 'a',
   gdyż przejście do niej jest dopuszczalne.
```

Komunikaty w sytuacji, gdy linie 26, 27 i 28 zostały zakomentowane, są następujące:

```
Jestem tu 0!
   Kompilator Prologu nie wywolal 'cut'.
   Dzieki temu, poniewaz pierwszej reguly dla 'b'
   nie udalo sie spelnic, 'a' jest prawdziwe dzieki
   spelnieniu drugiej reguly dla 'b'.
```

A więc `cut/0` jest drugim (obok `fail/0`) *explicite* proceduralnym operatorem, potrzebnym by częściowo deklaratywne programy zechciały efektywnie funkcjonować.

2.1.12 Ułomność logiki prologowej (i clp-owej też!)

Czytelnik zapewne już zauważył, że w *Prolog*u jest stosunkowo mało logiki, biorąc pod uwagę ogromny zasób wiedzy dyscypliny matematycznej o nazwie *logika*. To samo okaże się być prawdą dla *CLP*. Mało tego, to niewiele logiki w *Prolog*u (i w *CLP*) jest nieco ułomne (tzn. niezgodne z matematycznymi definicjami), co jest spowodowane tym, że programy prologowe i clp-owe (póki co) są realizowane przez pojedyncze procesory, przetwarzające klauzule programów kolejno w czasie, z góry programu w dół, z lewej strony ciała reguły w prawą stronę ciała reguły.

Dla ukonkretnienia tego stwierdzenia popatrzmy na ciało dowolnej reguły: warunki tego ciała zostały uprzednio uznane za tworzące *koniunkcję*. Jak nas uczy logika, koniunkcja jest *komutatywna*: zmiana kolejności koniungowanych zmiennych nie wpływa na wartość logiczną koniunkcji. Niestety, właściwość komutatywności bywa naruszana dla reguł prologowych (i clp-owych). Program `2_5_graty.pl` stanowi dobry przykład dla zilustrowania takiego naruszenia. Rozpatrzmy zamianę miejscami linii `5` i linii `8`. Program nadal kompiluje się, lecz po uruchomieniu generuje komunikat:

```
instantiation fault in (0, _347, _355).
```

Powód tego jest oczywisty: predykaty w ciele reguły są testowane (przez jeden procesor) w kolejności ich występowania, od lewej strony w prawo, lub z góry w dół. Dlatego nie można (w liniach `7` i `8`) korzystać z danych dla wciąż jeszcze nieuziemionej zmiennej `Lista`: zmienna ta nie została uziemiona w wyniku zamiany linii `5` z linią `8`. Stąd wniosek, że w Prologu (i w *CLP*) obowiązkiem programisty jest zadbanie o "właściwą kolejność" predykatów występujących w ciałach reguł.

2.2 Problemy konfigurowania

2.2.1 Konfigurowanie systemu 3-elementowego

Rozpatrzmy następujący przykład konfigurowania systemu składającego się z trzech elementów, elementu typu `A`, typu `B` i typu `C`, dla którego:

- elementami typu `A` są `A1`, `A2`, `A3`;
- elementami typu `B` są `B1`, `B2`, `B3`, `B4`;
- elementami typu `C` są `C1`, `C2`.

Elementy te mają różne ceny:

- cena `A1` to 1900; cena `A2` to 750; cena `A3` to 900;

- cena `B1` to 300; cena `B2` to 500; cena `B3` to 450; cena `B4` to 600;
- cena `C1` to 700; cena `C2` to 850.

Pewne elementy nie są kompatybilne z innymi elementami:

- `C1` nie jest kompatybilne z `A2`;
- `B2` nie jest kompatybilne z `C2`;
- `C2` nie jest kompatybilne z `B3`;
- `B4` nie jest kompatybilne z `A2`;
- `B3` nie jest kompatybilne z `A1`;
- `A3` nie jest kompatybilne z `B3`;

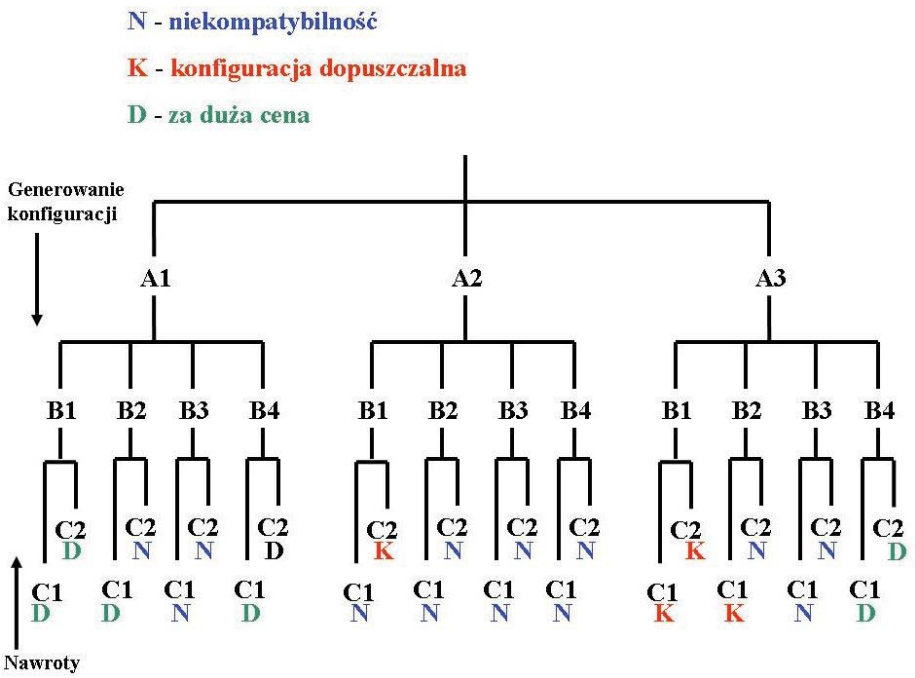
Należy (1) wyznaczyć system składający się z trzech kompatybilnych elementów `A`, `B` i `C` i cenie nie większej niż cena progowa równa 2100, oraz (2) wyznaczyć system składający się z trzech kompatybilnych elementów `A`, `B` i `C` o najmniejszej cenie. Tak sformułowane zadanie jest *generyczne*, tzn. odpowiada bardzo dużej liczbie konkretnych konfigurowań (np. doboru dań obiadu, doboru elementów stroju ślubnego, doboru elementów systemu komputerowego).

2.2.2 Metoda przeglądu zupełnego

Najbardziej prymitywnym sposobem rozwiązania zadania konfigurowania jest *metoda przeglądu zupełnego*: generujemy kolejne konfiguracje (`A`, `B` i `C`) po czym sprawdzamy, czy zawierają one elementy kompatybilne i czy cena systemu nie przekracza ceny progowej. Wymaga to generowania wszystkich konfiguracji. Aczkolwiek dla przeglądu zupełnego nie jest to potrzebne, dla potrzeb dalszych modyfikacji zakłada się, że uziemianie zmiennych następuje kolejno, np. najpierw `A`, potem `B` i w końcu `C`. Wówczas przebieg przeglądu zupełnego można przedstawić korzystając z *drzewa poszukiwań* z rysunku 2.4.

Przegląd zupełny zaczynamy (całkowicie arbitralnie) od zmiennej `A`, której przyporządkowujemy wartość `A1`. Ponieważ nie jest to jedyny sposób uziemiania zmiennej `A` (może ona przyjąć również wartości `A2` i `A3`),wierzchołek drzewa rozwiązań, w którym uziemiamy `A`, staje się *punktem wyboru*. W następnej kolejności uziemiamy zmienną `B`, której przyporządkowujemy wartość `B1` tworząc zarazem kolejny punkt wyboru. Na końcu uziemiamy zmienną `C` na wartości `C1`, tworząc jeszcze jeden punkt wyboru. Dopiero wówczas przystępujemy do testowania kompatybilności i ceny wybranych elementów, stwierdzając, że łączna cena wybranych elementów przekracza cenę progową 2100. Dlatego dokonujemy *nawrotu*, którego istota polega na:

- odziemieniu zmiennej `C`, w wyniku czego nie jest ona już przyporządkowana wartości `C1`;



Rysunek 2.4: Drzewo poszukiwań metodą przeglądu zupełnego dla konfigurowania

- przejściu (w górę drzewa) do najbliższego punktu wyboru. Jest nim punkt wyboru wartości zmiennej `C`;
- reuziemieniu zmiennej `C` na wartości `C2` dotąd nie używanej. Ponieważ tym samym zostały wyczerpane możliwości wyboru w punkcie wyboru wartości `C`, punkt ten znika.

W ten sposób uzyskaliśmy konfigurację (`A1`, `B1`, `C2`), której cena również przekracza cenę progową. Dlatego:

- odziemiamy ponownie zmienną `C`;
- przechodzimy (w górę drzewa) do najbliższego punktu wyboru. Jest nim punkt wyboru wartości zmiennej `B`;

- odziemiamy zmienną B;
- reuziemiamy ją na wartości B2;
- przechodzimy do uziemiania zmiennej C na wartości C1, tworząc nowy punkt wyboru dla C.

W ten sposób uzyskaliśmy konfigurację (A1, B2, C1), której cena również przekracza cenę progową. Dlatego wracamy do punktu wyboru dla C, generując konfigurację (A1, B2, C2), zawierającą niekompatybilne elementy B2 i C2. Czytelnik na podstawie rysunku 2.4 z łatwością prześledzi dalsze poszukiwania.

Wymieniony system można skonfigurować na 24 sposoby, przy czym 13 konfiguracji zawiera niekompatybilne elementy, a 7 konfiguracji przekracza założoną cenę 2100. Następujące cztery konfiguracje są konfiguracjami dopuszczalnymi:

- Konfiguracja(A2, B1, C2) w cenie 1900;
- Konfiguracja(A3, B1, C1) w cenie 1900;
- Konfiguracja(A3, B1, C2) w cenie 2050;
- Konfiguracja(A3, B2, C1) w cenie 2100.

Istotą zaprezentowanego postępowania było kolejne uziemianie, odziemianie i reuziemianie zmiennych aż do wyczerpania istniejących możliwości. Cenę progową umieszczamy w dynamicznej bazie danych (asertujemy)za pomocą uziemionego predykatu `cena_dopuszczalna(Cena_dopuszczalna)`. Przedstawiony przegląd zupełny realizuje program (`2_7_konf_pz.pl`), w którym cenę progową umieszczono w dynamicznej bazie danych (*asertowano*)za pomocą uziemionego predykatu `cena_dopuszczalna(Cena_dopuszczalna)`:

```
/*1*/   top:-
/*2*/       assert(cena_dopuszczalna(2100)),
/*3*/       cena_dopuszczalna(Cena_dopuszczalna),
/*4*/       konfiguracja(Cena_dopuszczalna).

/*5*/   konfiguracja(Cena_dopuszczalna):-
/*6*/       member(A,["A1","A2","A3"]),
/*7*/       member(B,["B1","B2","B3","B4"]),
/*8*/       member(C,["C1","C2"]),
/*9*/       not(niekompatybilnosc([A,B])),
/*10*/      not(niekompatybilnosc([A,C])),
/*11*/      not(niekompatybilnosc([B,C])),
/*12*/      cena(A,Cena_A),
```

```
/*13*/      cena(B,Cena_B),
/*14*/      cena(C,Cena_C),
/*15*/      Cena_calkowita is Cena_A+Cena_B+Cena_C,
/*16*/      Cena_dopuszczalna>=Cena_calkowita,
/*17*/      write("Konfiguracja("),write(A),write(","),
/*18*/      write(B),write(","),write(C),write(")"),
/*19*/      write(" w cenie "),write(Cena_calkowita),
/*20*/      nl,fail.


/*21*/  konfiguracja(Cena_dopuszczalna):-
/*22*/      write("To wszystkie konfiguracje w "),
/*23*/      write("cenie nie przekraczajacej"),
/*24*/      write(Cena_dopuszczalna),write(".").


/*25*/  cena("A1",1900). cena("A2",750). cena("A3",900).
/*26*/  cena("B1",300). cena("B2",500). cena("B3",450).
/*27*/  cena("B4",600). cena("C1",700). cena("C2",850).


/*28*/  niekompatybilne("C1","A2").
/*29*/  niekompatybilne("B2","C2").
/*30*/  niekompatybilne("C2","B3").
/*31*/  niekompatybilne("B4","A2").
/*32*/  niekompatybilne("B3","A1").
/*33*/  niekompatybilne("A3","B3").


/*34*/  niekompatybilnosc([X,Y]):- niekompatybilne(X,Y),!.
/*35*/  niekompatybilnosc([X,Y]):- niekompatybilne(Y,X),!.
```

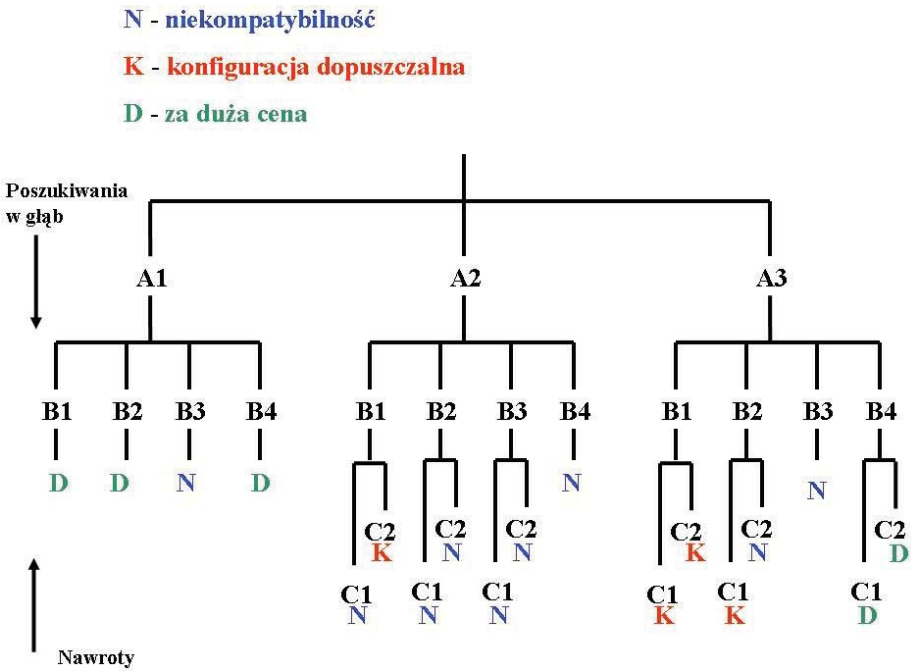
Program generuje komunikat:

```
Konfiguracja(A2,B1,C2) w cenie 1900
Konfiguracja(A3,B1,C1) w cenie 1900
Konfiguracja(A3,B1,C2) w cenie 2050
Konfiguracja(A3,B2,C1) w cenie 2100
To wszystkie konfiguracje w cenie nie przekraczajacej 2100.
```

Należy zwrócić uwagę na *deklaratywność* programu: jedynym miejscem, gdzie rezygnuje się (siłą rzeczy) z "opisu świata" i każe się programowi coś zrobić, to predykaty pisania *write/1*.

2.2.3 Metoda poszukiwań w głąb ze standardowymi nawrotami

Przegląd zupełny jest mało efektywny z tego powodu, że ograniczenia kompatybilności i ceny są testowane dopiero po wygenerowaniu pełnej konfiguracji. Jest oczywistym że można to zrobić wcześniej, po każdorazowym uziemieniu dowolnej zmiennej. Zostało to pokazane na rysunku 2.5.



Rysunek 2.5: Drzewo poszukiwań metodą poszukiwań w głąb ze standardowymi nawrotami dla konfigurowania

Np. jeżeli po wyborze `A1` wybierzemy `B1`, cena konfiguracji cząstkowej (`A1`, `B1`) już przekracza cenę progową, a więc uzupełnianie tej konfiguracji nie ma sensu, wykonujemy nawrót do punktu wyboru dla `B`. Jeżeli zaś po wyborze `A1` wybierzemy `B3`, konfiguracja cząstkowa (`A1`, `B3`) ma elementy niekompatybilne, a więc uzupełnianie tej konfiguracji również nie ma sensu i ponownie wykonujemy

nawrót do punktu wyboru dla B. Jeżeli natomiast po wyborze A2 i B1 wszystkie ograniczenia są spełnione, to wybór C2 skutkuje wyznaczeniem konfiguracji spełniającej wszystkie ograniczenia.

Rozwiązanie metodą poszukiwań w głąb ze standardowymi nawrotami sprowadza się więc do naprzemiennego wykonania dwóch działań, którymi są:

1. *Poszukiwania*, polegające na uziemieniu kolejnej wolnej zmiennej decyzyjnej. W wyniku tego jest generowana jedna z gałęzi drzewa rozwiązań.
2. *Propagacja*, polegająca na uwzględnieniu dokonanego wyboru wartości ostatnio uziemionej zmiennej na spełnienie ograniczeń zadania. Propagacja następuje każdorazowo po uziemieniu kolejnej wolnej zmiennej decyzyjnej. Wynik propagacji może być dwojaki:
   (a) jeżeli wszystkie ograniczenia są spełnione, to rozpoczyna się kolejne poszukiwanie w głąb drzewa rozwiązań;
   (b) jeżeli jakieś ograniczenie nie jest spełnione, to wykonywany jest nawrót do najbliższego punktu wyboru drzewa rozwiązań.

Istotą poszukiwań jest więc inkrementalne rozszerzenie częściowego (spełniającego ograniczenia) rozwiązania do etapu uzyskania rozwiązania całkowitego. Rozwiązanie metodą poszukiwań i propagacji daje program 2_8_konf_pp.pl:

```
/*1*/   top:-
/*2*/       assert(cena_dopuszczalna(2100)),
/*3*/       cena_dopuszczalna(Cena_dopuszczalna),
/*4*/       konfiguracja(Cena_dopuszczalna).

/*5*/   konfiguracja(Cena_dopuszczalna):-
/*6*/       member(A,["A1","A2","A3"]),
/*7*/       cena(A,Cena_A),
/*8*/       Cena_A =< Cena_dopuszczalna,
/*9*/       member(B,["B1","B2","B3","B4"]),
/*10*/      not(niekompatybilnosc([A,B])),
/*11*/      cena(B,Cena_B),
/*12*/      Cena_AB is Cena_A+Cena_B,
/*13*/      Cena_AB =< Cena_dopuszczalna,
/*14*/      member(C,["C1","C2"]),
/*15*/      not(niekompatybilnosc([A,C])),
/*16*/      not(niekompatybilnosc([B,C])),
/*17*/      cena(C,Cena_C),
/*18*/      Cena_calkowita is Cena_A+Cena_B+Cena_C,
/*16*/      Cena_calkowita =< Cena_dopuszczalna,
/*17*/      write("Konfiguracja("),write(A),write(","),
```

```
/*18*/      write(B),write(","),write(C),write(")"),
/*19*/      write(" w cenie "),write(Cena_calkowita),
/*20*/      nl,fail.

/*21*/  konfiguracja(Cena_dopuszczalna):-
/*22*/      write("To wszystkie konfiguracje w "),
/*23*/      write("cenie nie przekraczajacej "),
/*24*/      write(Cena_dopuszczalna),write(").

/*25*/  cena("A1",1900). cena("A2",750). cena("A3",900).
/*26*/  cena("B1",300). cena("B2",500). cena("B3",450).
/*27*/  cena("B4",600). cena("C1",700). cena("C2",850).

/*28*/  niekompatybilne("C1","A2").
/*29*/  niekompatybilne("B2","C2").
/*30*/  niekompatybilne("C2","B3").
/*31*/  niekompatybilne("B4","A2").
/*32*/  niekompatybilne("B3","A1").
/*33*/  niekompatybilne("A3","B3").

/*34*/  niekompatybilnosc([X,Y]):- niekompatybilne(X,Y),!.
/*35*/  niekompatybilnosc([X,Y]):- niekompatybilne(Y,X),!.
```

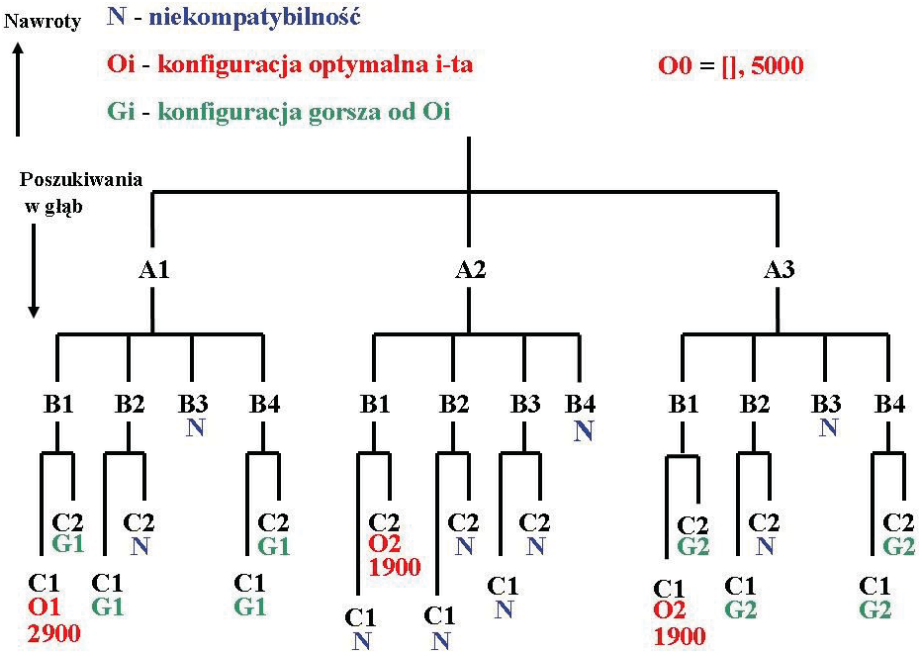
2.3 Problemy konfigurowania optymalnego

2.3.1 Konfigurowanie systemu 3-elementowego

Cenę pełnej konfiguracji można uważać za wskaźnik jakości problemu optymalizacyjnego polegającego na wyznaczeniu konfiguracji o najmniejszej cenie. Rozwiązanie tego problemu można uzyskać metodą *gałęzi i ograniczeń* (ang. *branch-and-bound*). W dalszym ciągu okaże się, że "ograniczenia" metody *gałęzi i ograniczeń* są - podobnie jak ograniczenia *CLP* - ograniczeniami aktywnymi, ale o bardziej wyspecjalizowanej funkcjonalności. Metoda *gałęzi i ograniczeń* może korzystać również z mechanizmu poszukiwań w głąb ze standardowymi nawrotami. Robi to jednak nieco odmiennie, jak to pokazano na rysunku 2.6.

Odpowiedni program przedstawia `2_9_konf_opt.pl`:

```
/*1*/   top:-
   % Na wstępie asertujemy pustą ale bardzo kosztowną konfiguracje,
   % którą będziemy sukcesywnie aktualizować, aż stanie się optymalną:
/*2*/       assert(konfiguracja_najtansza([],5000)),
```



Rysunek 2.6: Drzewo poszukiwań metodą *gałęzi i ograniczeń* dla konfigurowania

```
/*3*/      fail.

/*4*/   top:-
/*5*/      member(A,["A1","A2","A3"]),
/*6*/      member(B,["B1","B2","B3","B4"]),
/*7*/      not(niekompatybilnosc([A,B])),
/*8*/      cena(A,Cena_A),
/*9*/      cena(B,Cena_B),
/*10*/     konfiguracja_najtansza(_,Cena_dotychczas_najmniejsza),
/*11*/     Cena_AB is Cena_A+Cena_B,
/*12*/     Cena_AB<Cena_dotychczas_najmniejsza,
/*13*/     member(C,["C1","C2"]),
/*14*/     not(niekompatybilnosc([A,C])),
/*15*/     not(niekompatybilnosc([B,C])),
/*16*/     cena(C,Cena_C),
/*17*/     Cena is Cena_AB+Cena_C,
/*18*/     aktualizuj_najtansza([A,B,C],Cena),
```

```
/*19*/      fail.

/*20*/   top:-
/*21*/      write("Najtansza jest: "),nl,
/*22*/      konfiguracja_najtansza([A,B,C],Cena),
/*23*/      write("Konfiguracja("),write(A),write(","),
/*24*/      write(B),write(", "),write(C),write(")"),
/*25*/      write(" w cenie "),write(Cena), nl,
/*26*/      fail.
/*27*/   top:-
/*28*/      write("To juz wszystkie najtansze konfiguracje!").

/*29*/   aktualizuj_najtansza([A,B,C],Cena):-
/*30*/      konfiguracja_najtansza(_,Cena_dotychczas_najmniejsza),
/*31*/      Cena_dotychczas_najmniejsza > Cena,
/*32*/      retract_all(konfiguracja_najtansza(_,_)),
/*33*/      assert(konfiguracja_najtansza([A,B,C],Cena)),!.

/*34*/   aktualizuj_najtansza([A,B,C],Cena):-
/*35*/      konfiguracja_najtansza(_,Cena_dotychczas_najmniejsza),
/*36*/      Cena_dotychczas_najmniejsza = Cena,
/*37*/      assert(konfiguracja_najtansza([A,B,C],Cena)),!.

/*38*/   aktualizuj_najtansza(_,Cena):-
/*39*/      konfiguracja_najtansza(_,Cena_dotychczas_najmniejsza),
/*40*/      Cena_dotychczas_najmniejsza<Cena,!.

/*41*/  cena("A1",1900). cena("A2",750). cena("A3",900).
/*42*/  cena("B1",300). cena("B2",500). cena("B3",450).
/*43*/  cena("B4",600). cena("C1",700). cena("C2",850).

/*44*/  niekompatybilne("C1","A2").
/*45*/  niekompatybilne("B2","C2").
/*46*/  niekompatybilne("C2","B3").
/*47*/  niekompatybilne("B4","A2").
/*48*/  niekompatybilne("B3","A1").
/*49*/  niekompatybilne("A3","B3").

/*50*/  niekompatybilnosc([X,Y]):- niekompatybilne(X,Y),!.
/*51*/  niekompatybilnosc([X,Y]):- niekompatybilne(Y,X),!.
```

Program generuje komunikat:

```
Najtansza jest:
Konfiguracja(A2, B1, C2) w cenie 1900
Konfiguracja(A3, B1, C1) w cenie 1900
```

```
To juz wszystkie najtansze konfiguracje!
```

Z powyższego widać, że dla rozpatrywanego przykładu metoda *gałęzi i ograniczeń* polega na:

- znalezieniu (lub zadeklarowaniu) dolnego ograniczenia wartości wskaźnika jakości (*ograniczenie* z nazwy metody). W rozpatrywanym przykładzie tym ograniczeniem było

  ```
  konfiguracja_najtansza([],5000);
  ```

- odrzucaniu wszystkich tych gałęzi drzewa rozwiązań, które dają wartości gorsze od aktualnego dolnego ograniczenia (przejdź na inną *gałąź* z nazwy metody) lub które naruszają ograniczenia problemu (*poszukiwania w głąb ze standardowymi nawrotami*);

- aktualizacji dolnego ograniczenia (*ograniczenie* z nazwy metody) w przypadku znalezienia rozwiązania lepszego od stosowanego dolnego ograniczenia.

Opisana najprostsza możliwa wersja metody *gałęzi i ograniczeń* będzie w dalszym ciągu nazywana wersją standardową.

2.4 Problemy przyporządkowania

2.4.1 Golfiści

Bardzo dobrymi ćwiczeniami umiejętności modelowania w *Prolog*u i w językach *CLP* są ćwiczenia polegające na rozwiązywaniu różnego rodzaju łamigłówek, dla których zazwyczaj dana jest tylko część faktów, ale są to fakty determinujące wartości pozostałych, nieznanych faktów. Ważność tej grupy ćwiczeń potwierdza duża liczba witryn poświęconych rozwiązywaniu łamigłówek w *Prolog*u lub w językach *CLP*, patrz np. [Dough-10]. Ważniejszą zapewne okolicznością jest to, że łamigłówki w bardzo uproszczonej formie przedstawiają problemy występujące w nieporównanie spotęgowanej skali w takich realistycznych problemach, jak np. układanie rozkładu zajęć studenckich i marszrutyzacja pojazdów. Owe "realistyczne problemy" nie są niestety problemami, od których można zacząć naukę *Prolog*u i języków *CLP*. Rozpatrzmy następujący przykład[9]:

[9]Temat przykładu został zaczerpnięty z książki [Friedman-Hill-03].

Czterech golfistów (Fred, Jurek, Robert, Tomek) stoi przy dołku, w rzędzie, od lewej do prawej. Każdy z nich ma spodnie innego koloru:
(1)Jeden ma spodnie czerwone.
(2)Golfista na prawo od Freda ma spodnie niebieskie.
(3)Jurek jest na miejscu drugim.
(4)Robert ma spodnie białe.
(5)Tomek nie jest na miejscu drugim ani na czwartym i nie ma spodni pomarańczowych.
W jakiej kolejności stoją golfiści i jakie mają spodnie?

Modelowania problemu zaczyna się od zdefiniowania kilku niezbędnych "prywatnych" predykatów. Mogą nimi być np.:

- `warunki(Numer_miejsca_golfisty,Imię_golfisty, Kolor_spodni_golfisty)`
- `wszystkie_pozycje_rozne(Numer_miejsca_1,Numer_miejsca_2, Numer_miejsca_3,Numer_miejsca_4)`
- `wszystkie_kolory_rozne(Kolor_1, Kolor_2, Kolor_3,Kolor_4)`
- `warunek_1(Numer_miejsca_1, Kolor_spodni_golfisty_na_miejscu_1, Numer_miejsca_2, Kolor_spodni_golfisty_na_miejscu_2, Numer_miejsca_3, Kolor_spodni_golfisty_na_miejscu_3, Numer_miejsca_4, Kolor_spodni_golfisty_na_miejscu_4)`
- `warunek_2(Numer_miejsca_1, Kolor_spodni_golfisty_na_miejscu_1, Numer_miejsca_2, Kolor_spodni_golfisty_na_miejscu_2, Numer_miejsca_3, Kolor_spodni_golfisty_na_miejscu_3, Numer_miejsca_4, Kolor_spodni_golfisty_na_miejscu_4)`
- `kolory(nazwa koloru)`
- `pozycja(numer_miejsca)`.

Odpowiedni program (`2_10_golfisci.pl`) jest następujący:

```
/*1*/   top:-

   % Fred stoi gdzieś tam i ma jakieś tam spodnie:
/*2*/       warunki(P1,"Fred",C1),
```

```
   % Jurek jest na miejscu drugim:
/*3*/       warunki(P2, "Jurek",C2),
/*4*/       P2 is 2,

   % Robert ma spodnie białe:
/*5*/       warunki(P4, "Robert", C4),
/*6*/       C4 = biale,

   % Tomek nie jest na miejscu drugim ani na czwartym
   % i nie ma spodni pomarańczowych:
/*7*/       warunki(P3,"Tomek",  C3),
/*8*/       C3 \==  pomaranczowe,
/*9*/       P3 =\= 2,
/*10*/      P3 =\= 4,

/*11*/      wszystkie_pozycje_rozne(P1, P2, P3, P4),
/*12*/      wszystkie_kolory_rozne(C1, C2, C3, C4),

   % Ktoś ma spodnie czerwone - patrz definicja warunek_1/8:
/*13*/      warunek_1(P1, C1, P2, C2, P3, C3, P4, C4),

   % Golfista na prawo od Freda ma spodnie niebieskie -
   % patrz definicja warunek_2/8:
/*14*/      warunek_2(P1, C1, P2, C2, P3, C3, P4, C4),

/*15*/      write("Fred jest na pozycji "),write(P1),
                    write(" i ma spodnie "),write(C1), nl,
/*16*/      write("Jurek jest na pozycji "), write(P2),
                    write(" i ma spodnie "),write(C2), nl,
/*17*/      write("Tomek jest na pozycji "), write(P3),
                    write(" i ma spodnie "),write(C3), nl,
/*18*/      write("Robert jest na pozycji "), write(P4),
                    write(" i ma spodnie "),write(C4),nl,nl,fail.

/*19*/   top:-
/*20*/      write("To wszystko!"),nl.

/*21*/   warunki(Numer,_,Kolor) :-
/*22*/      pozycja(Numer),
/*23*/      kolor(Kolor).

/*24*/   warunek_1(_,C1,_,_,_,_,_,_) :-
/*25*/      C1 = czerwone,!.
/*26*/   warunek_1(_,_,_,C2,_,_,_,_) :-
/*27*/      C2 = czerwone,!.
/*28*/   warunek_1(_,_,_,_,_,C3,_,_) :-
/*29*/      C3 = czerwone,!.
/*30*/   warunek_1(_,_,_,_,_,_,_,C4) :-
```

```
/*31*/      C4 = czerwone,!.

/*32*/   warunek_2(P1,_,P2,C2,_,_,_,_) :-
/*33*/      P2 is P1 + 1,
/*34*/      C2 = niebieskie,!.
/*35*/   warunek_2(P1,_,_,_,P3,C3,_,_) :-
/*36*/      P3 is P1 + 1,
/*37*/      C3 = niebieskie,!.
/*38*/   warunek_2(P1,_,_,_,_,_,P4,C4) :-
/*39*/      P4 is P1 + 1,
/*40*/      C4 = niebieskie,!.

/*41*/   wszystkie_pozycje_rozne(X1, X2, X3, X4) :-
/*42*/      X1 =\= X2, X1 =\= X3, X1 =\= X4,
/*43*/      X2 =\= X3, X2 =\= X4, X3 =\= X4.

/*44*/   wszystkie_kolory_rozne(X1, X2, X3, X4) :-
/*45*/      X1 \== X2, X1 \== X3, X1 \== X4,
/*46*/      X2 \== X3, X2 \== X4, X3 \== X4.

/*47*/      kolor(pomaranczowe).
/*48*/      kolor(niebieskie).
/*49*/      kolor(czerwone).
/*50*/      kolor(biale).

/*51*/      pozycja(1).
/*52*/      pozycja(2).
/*53*/      pozycja(3).
/*54*/      pozycja(4).
```

Rozwiązanie generowane przez program jest następujące:

```
Fred jest na pozycji 1 i ma spodnie pomaranczowe
Jurek jest na pozycji 2 i ma spodnie niebieskie
Tomek jest na pozycji 3 i ma spodnie czerwone
Robert jest na pozycji 4 i ma spodnie białe.
```

2.4.2 Trzy kule

Porównywanie atrybutów dla bytów opisanych innymi różnymi atrybutami stwarza często trudności. W poniższym przykładzie porównuje się rozmiary kul opisanych różnymi atrybutami: jedna kula jest opisana numerem, druga - kolorem.

Przykład jest następujący:

Trzy kule są różnych rozmiarów (mały, duży, średni), są w trzech różnych kolorach (czarny, szary, biały) i są opatrzone trzema różnymi numerami (1,2,3). Wiadomo, że:
(1) Duża kula ma jaśniejszy kolor od kuli średniej;
(2) Mała kula ma numer 2;
(3) Numer kuli czarnej jest większy niż numer na kuli białej;
(4) Rozmiar kuli o numerze 3 jest mniejszy niż kuli szarej.
Jakie numery i jakie kolory są na poszczególnych kulach? W tym przypadku wystarczą nam trzy predykaty "prywatne":

- `kula(Kolor_kuli, Rozmiar_kuli, Numer_kuli)`,
- `mniejszy_rozmiar(Mniejszy_rozmiar, Większy_rozmiar)`,
- `jasniej(Jaśniejszy_kolor, Ciemniejszy_kolor)`.

Odpowiedni program (`2_11_trzy_kule.pl`) ma postać:

```
/*1*/   top:-
        %(1) Duża kula ma jaśniejszy kolor niż kula średnia.
/*2*/      kula(Kolor_duzej,duzy,Numer_duzej),
/*3*/      kula(Kolor_sredniej,sredni,Numer_sredniej),
/*4*/      jasniej(Kolor_duzej,Kolor_sredniej),

        %(2) Mała kula ma numer 2.
/*5*/      kula(Kolor_malej,maly,2),

        %(3) Numer  kuli czarnej jest większy niż numer kuli białej.
/*6*/      kula(czarny,_,Numer_czarnej),
/*7*/      kula(bialy,_,Numer_bialej),
/*8*/      Numer_czarnej > Numer_bialej,

        %(4) Rozmiar kuli o numerze 3 jest mniejszy niż rozmiar kuli szarej.
/*9*/      kula(_,Rozmiar_3, 3),
/*10*/     kula(szary,Rozmiar_szarej,Numer_szarej),
/*11*/     mniejszy_rozmiar(Rozmiar_3,Rozmiar_szarej),

        %numery kul są różne.
/*12*/     Numer_szarej =\= Numer_bialej,
/*13*/     Numer_szarej =\= Numer_czarnej,
/*14*/     Numer_bialej =\= Numer_czarnej,
/*15*/     2 =\= Numer_duzej,
/*16*/     2 =\= Numer_sredniej,
```

```
/*17*/      Numer_duzej=\=Numer_sredniej,

        %kolory kul są różne.
/*18*/      Kolor_duzej\==Kolor_sredniej,
/*19*/      Kolor_duzej\==Kolor_malej,
/*20*/      Kolor_malej\==Kolor_sredniej,

/*21*/   writeln("Kolor duzej kuli": Kolor_duzej),
/*22*/   writeln("Numer duzej kuli": Numer_duzej),nl,
/*23*/   writeln("Kolor sredniej kuli": Kolor_sredniej),
/*24*/   writeln("Numer sredniej kuli": Numer_sredniej),nl,
/*25*/   writeln("Kolor malej kuli": Kolor_malej),
/*26*/   writeln("Numer malej kuli": 2),nl.

/*27*/  kula(Kolor,Rozmiar,Numer):-
/*28*/      member(Kolor,[czarny,szary, bialy]),
/*29*/      member(Rozmiar,[maly, duzy, sredni]),
/*30*/      member(Numer,[1,2,3]).

/*31*/  mniejszy_rozmiar(maly, duzy).
/*32*/  mniejszy_rozmiar(maly, sredni).
/*33*/  mniejszy_rozmiar(sredni, duzy).

/*34*/  jasniej(bialy,szary).
/*35*/  jasniej(bialy,czarny).
/*36*/  jasniej(szary,czarny).
```

Rozwiązanie ma postać:

```
Kolor duzej kuli : szary
Numer duzej kuli : 1

Kolor sredniej kuli : czarny
Numer sredniej kuli : 3

Kolor malej kuli : bialy
Numer malej kuli : 2
```

2.4.3 Kto zabił?

Powszechną trudnością przy modelowaniu problemów decyzyjnych za pomocą ECL^iPS^e *Prolog* (jak i innych *Prolog*ów oraz języków *CLP*) jest trudność tworzenia predykatów opisujących problem. Widać to dobrze na poniższym przykładzie, dla którego wybór predykatów "prywatnych" wcale nie jest oczywisty,

i co może najważniejsze - nie jest jedynym wyborem prowadzącym do rozwiązania.

Należy rozwiązać następującą zagadkę kryminalną:
Arek, Bolek i Czarek tworzą zamknięty krąg podejrzanych o zamordowanie Marka. Podejrzani zostali przesłuchani i zeznali co następuje:
1)Arek stwierdził, że jest niewinny, i że przyjacielem Marka był Bolek, a Czarek nienawidził Marka.
2)Bolek zeznał, że ma alibi na czas popełnienia zbrodni, a Marka nie znał.
3)Czarek stwierdził, że jest niewinny, i że tuż przed zbrodnią rozmawiał z Bolkiem i Markiem oraz rozmawiał z Arkiem i Markiem.
Zbrodnię popełnił ten z nich, którego zeznania są niezgodne z zeznaniami pozostałych dwóch podejrzanych. Program wykrywający mordercę korzysta więc ze znanej zasady Sherlock Holmes'a: połącz fakty w logicznie zgodną całość, a uzyskasz rozwiązanie. Program ten (`2_12_kto_zabil.pl`) jest następujący:

```
/*1*/   top:-
/*2*/       morderca.

/*3*/   zeznania_Arka([niewinny("Arek"),przyjaciel("Bolek","Marek"),
                     nienawidzi("Czarek","Marek")]).
/*4*/   zeznania_Bolka([alibi("Bolek"),nie_zna("Bolek","Marek")]).
/*5*/   zeznania_Czarka([niewinny("Czarek"),razem("Czarek","Marek"),
                     razem("Bolek","Marek"),razem("Arek","Marek")]).

/*6*/   morderca:-
/*7*/      zeznania_Arka(Lista_a),
/*8*/      zeznania_Bolka(Lista_b),
/*9*/      zeznania_Czarka(Lista_c),
/*10*/     zeznania_zgodne(Lista_b,Lista_c),
/*11*/     zeznania_niezgodne(Lista_a,Lista_b),
/*12*/     zeznania_niezgodne(Lista_a,Lista_c),
/*13*/     write("Morderca to Arek".),nl,!.

/*14*/   morderca:-
/*15*/     zeznania_Arka(Lista_a),
/*16*/     zeznania_Bolka(Lista_b),
/*17*/     zeznania_Czarka(Lista_c),
/*18*/     zeznania_zgodne(Lista_a,Lista_c),
/*19*/     zeznania_niezgodne(Lista_a,Lista_b),
/*20*/     zeznania_niezgodne(Lista_b,Lista_c),
/*21*/     write("Morderca to Bolek".),nl,!.

/*22*/  morderca:-
/*23*/     zeznania_Arka(Lista_a),
```

```
/*24*/     zeznania_Bolka(Lista_b),
/*25*/     zeznania_Czarka(Lista_c),
/*26*/     zeznania_zgodne(Lista_a,Lista_b),
/*27*/     zeznania_niezgodne(Lista_a,Lista_c),
/*28*/     zeznania_niezgodne(Lista_b,Lista_c),
/*29*/     write("Morderca to Czarek".),nl,!.

/*30*/  zeznania_zgodne(Zeznanie_1,Zeznanie_2):-
/*31*/     not(zeznania_niezgodne(Zeznanie_1,Zeznanie_2)).

/*32*/  zeznania_niezgodne(Zeznanie_1,Zeznanie_2):-
/*33*/     iloczyn_kartezjański(Zeznanie_1,Zeznanie_2,Iloczyn_kart),
/*34*/     testuj_sprzecznosc_par(Iloczyn_kart).

/*35*/  iloczyn_kartezjański([], _, []).
/*36*/  iloczyn_kartezjański([H|T], L, M) :-
/*37*/     generuj_pary(H,L,M1),
/*38*/     iloczyn_kartezjański(T, L, M2),
/*39*/     append(M1, M2, M).

/*40*/  generuj_pary(_, [], []).
/*41*/  generuj_pary(A, [B|L], [[A,B]|N] ) :-
/*42*/     generuj_pary(A, L, N).

/*43*/  testuj_sprzecznosc_par([[H1,H2]|T]):-
/*44*/     not(sprzecznosc_par(H1,H2)),
/*45*/     testuj_sprzecznosc_par(T).

/*46*/  testuj_sprzecznosc_par([[H1,H2]|_]):-
/*47*/     sprzecznosc_par(H1,H2),
/*48*/     !.

/*49*/  sprzecznosc_par(P1,P2):-
/*50*/     parami_sprzeczne([P1,P2]).

/*51*/  sprzecznosc_par(P1,P2):-
/*52*/     parami_sprzeczne([P2,P1]).

/*53*/  parami_sprzeczne([nienawidzi("Bolek","Marek"),
                 przyjaciel("Bolek","Marek")]).
/*54*/  parami_sprzeczne([przyjaciel("Bolek","Marek"),
                 nie_zna("Bolek","Marek")]).
/*55*/  parami_sprzeczne([razem("Bolek","Marek"),
                 nie_zna("Bolek","Marek")]).
/*55*/  parami_sprzeczne([przyjaciel("Czarek","Marek"),
                 nienawidzi("Czarek","Marek")]).
```

```
/*56*/  parami_sprzeczne([niewinny("Arek"),winny("Arek")]).
/*57*/  parami_sprzeczne([niewinny("Czarek"),winny("Czarek")]).
/*58*/  parami_sprzeczne([alibi("Bolek"),razem("Bolek","Marek")]).
/*59*/  parami_sprzeczne([alibi("Bolek"),winny("Bolek")]).
/*60*/  parami_sprzeczne([alibi("Czarek"),razem("Czarek","Marek")]).
```

Program generuje rozwiązanie: `Morderca to Bolek.`

2.4.4 Hetmani - definiowanie zmiennych

Problem rozmieszczania `N` hetmanów na szachownicy o wymiarach $N \times N$ tak, by żaden nie atakował innego, jest tradycyjnym *benchmarkiem* sztucznej inteligencji[10]. Każde zadanie prologowe lub clp-owe wymaga dokonania wyboru sposobu definiowania zmiennych. W poprzednich zadaniach wybór ten nie był problemem. W przypadku rozmieszczania hetmanów - może być problemem. Okazuje się, że najbardziej efektywnym sposobem kodowania zmiennych jest kodowanie ustawienia hetmanów (na standardowej szachownicy 8×8) w postaci listy:

```
[X1,X2,...,Xi,...,X8]
```

gdzie `Xi` jest numerem wiersza, w którym hetman jest w `i`-tej kolumnie. Definicja ta sama w sobie spełnia dwa ograniczenia:

1. Dwaj hetmani nigdy nie pojawią się w tej samej kolumnie gdyż każda pozycja w liście jest unikalna.

2. Dwaj hetmani nigdy nie pojawią się w tym samym wierszu jeżeli 8-krotka `X1,X2,...,X8` będzie uziemiana dla *permutacji* o 8-krotki `1,2,3,4,5,6,7,8`.

2.4.5 Hetmani - przegląd zupełny

Rozwiązanie metodą przeglądu zupełnego polega na generowaniu kolejnych permutacji listy `[1,2,3,4,5,6,7,8]` i sprawdzaniu, czy permutacje te odpowiadają ustawieniom bezpiecznym. Permutacje te spełniają zarazem wzmiankowane dwa ograniczenia:

[10] Wymyślne programy dla rozwiązywania problemu n hetmanów radzą sobie z tysiącami hetmanów na odpowiednio dużych szachownicach.

1. Każdy hetman jest w innym wierszu gdyż numery wierszy są różne.

2. Każdy hetman jest w innej kolumnie gdyż jest na innej pozycji w permutacji.

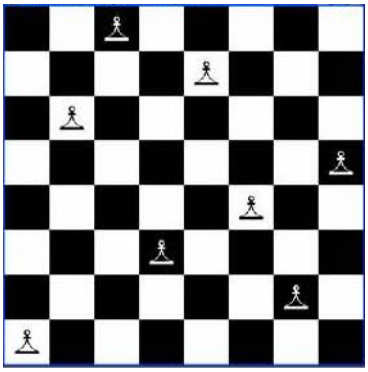
Odpowiedni program przedstawia przykład `2_13_hetmani_pz.pl`:

```
/*1*/   top:-
/*2*/       wszystkie_rozwiazania.

/*3*/   osiem_hetmanow([X1,X2,X3,X4,X5,X6,X7,X8]):-
/*4*/      permutacja([X1,X2,X3,X4,X5,X6,X7,X8],[1,2,3,4,5,6,7,8]),
/*5*/      bezpieczny([X1,X2,X3,X4,X5,X6,X7,X8]).

/*6*/   permutacja([],[]).
/*7*/   permutacja([X|Xs],Ls):-
/*8*/      usun(X,Ls,Rs),
/*9*/      permutacja(Xs,Rs).

/*10*/  usun(X,[X|Xs],Xs).
/*11*/  usun(X,[Y|Ys],[Y|Rs]):-
/*12*/     usun(X,Ys,Rs).

/*13*/  bezpieczny([]).
/*14*/  bezpieczny([X|Xs]):-
/*15*/     nie_atakuje(X,Xs),
/*16*/     bezpieczny(Xs).

/*17*/  nie_atakuje(X,Xs):-
/*18*/     nie_atakuje(X,Xs,1).

/*19*/  nie_atakuje(_,[],_).
/*20*/  nie_atakuje(X,[Y|Ys],Nb):-
/*21*/     X=\=Y-Nb,
/*22*/     X=\=Y+Nb,
/*23*/     Nb1 is Nb+1,
/*24*/     nie_atakuje(X,Ys,Nb1).

/*25*/  wszystkie_rozwiazania:-
/*26*/     osiem_hetmanow(X),
/*27*/     write(X),nl,
/*28*/     fail.

/*29*/  wszystkie_rozwiazania:-
/*30*/     write("To wszystko!").
```

Rozwiązań dopuszczalnych jest 92, dlatego przedstawimy tylko dwa pierwsze i dwa ostatnie:

```
[1, 5, 8, 6, 3, 7, 2, 4]
[1, 6, 8, 3, 7, 4, 2, 5]
.........................
[8, 3, 1, 6, 2, 5, 7, 4]
[8, 4, 1, 3, 6, 2, 7, 5]
To wszystko!
```

Przedostatnie rozwiązanie przedstawiono na szachownicy z rysunku 2.7

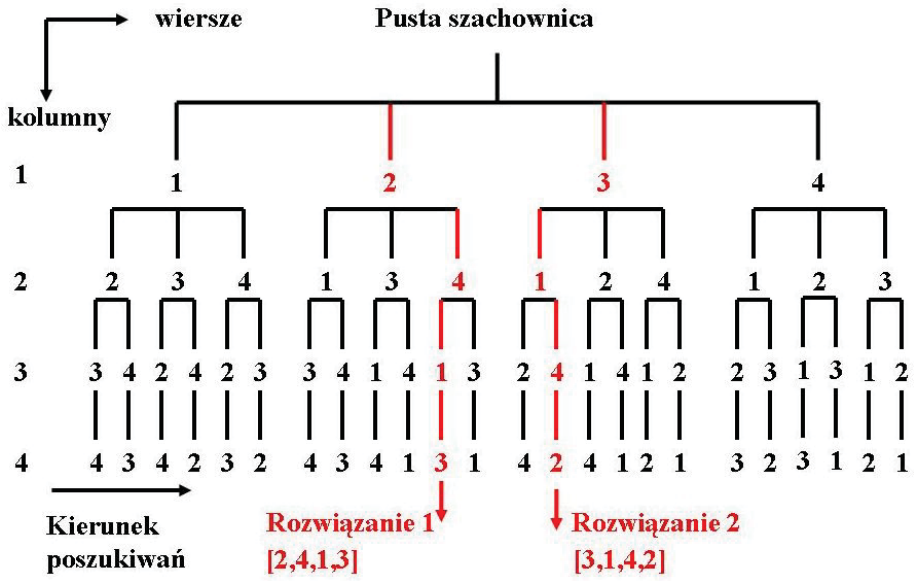


Rysunek 2.7: Przedostatnie bezpieczne ustawienie 8 hetmanów

Drzewo poszukiwań dla prostszego przypadku czterech hetmanów i przeglądu zupełnego przedstawiono na rysunku 2.8.

2.4.6 Hetmani - poszukiwania w głąb, standardowe nawroty

Przegląd zupełny ma oczywistą wadę, na którą zwrócono już uwagę w rozdziale 2.2.3. Załóżmy np., że już ustawienie pierwszych dwóch hetmanów jest ustawieniem niebezpiecznym. Zamiast czym prędzej z tego ustawienia się wycofać, kontynuujemy jednak (zupełnie bezsensownie) ustawianie dalszych hetmanów, by dopiero po ustawieniu ostatniego z nich testować bezpieczeństwo konfiguracji.



Rysunek 2.8: Drzewo poszukiwań metodą przeglądu zupełnego dla czterech hetmanów

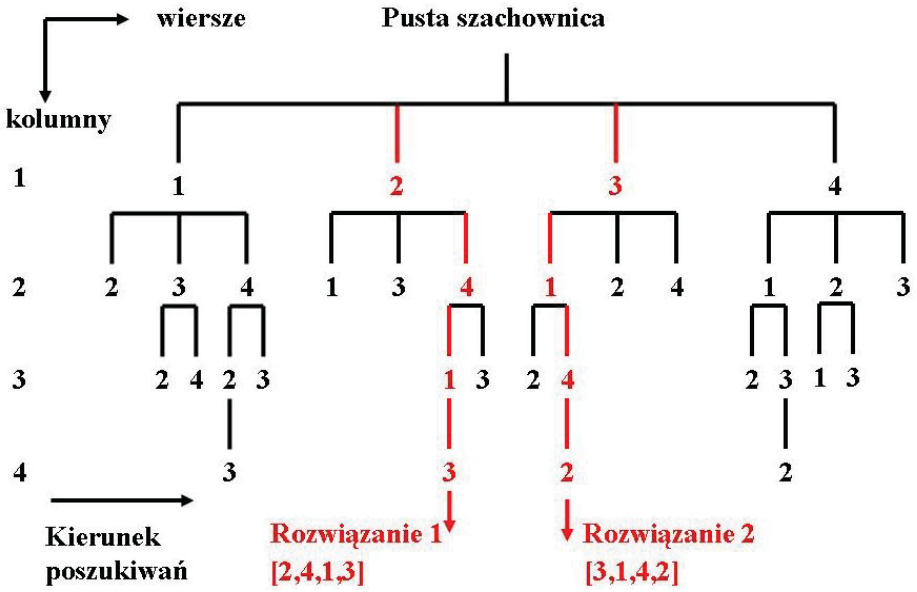
Można to zrobić znacznie lepiej. Obecnie do listy `[X1,X2,..Xi]` przedstawiającej bezpieczne ustawienia `i` hetmanów dodaje się nowego hetmana, po czym sprawdza się, czy obecne ustawienie jest nadal bezpieczne. Jeżeli tak, to dostawia się następnego hetmana. Jeżeli zaś ustawienie to nie jest bezpieczne, to wykonuje się nawrót do takiego najbliższego poprzedniego ustawienia, dla którego istnieje inny wybór dostawianego hetmana. Rozwiązanie to przedstawia przykład `2_14_hetmani_p.pl`. Jest ono znacznie bardziej efektywne aniżeli rozwiązanie z przykładu `2_13_hetmani_pz.pl`:

```
/*1*/   top:-
/*2*/       wszystkie_rozwiazania.

/*3*/   osiem_hetmanow(X):-
/*4*/      hetmani(X,[],[1,2,3,4,5,6,7,8]).
 % hetmani(Lista_wierszy_nieobsadzonych,
 %   Lista_wierszy_obsadzonych,Lista_wierszy_do_obsadzenia)
```

```
/*5*/   hetmani([],_,[]).
/*6*/   hetmani([X|Xs],Ulokowani,Lista_do_obsadzenia):-
/*7*/      usun(X,Lista_do_obsadzenia,Nowa_lista_do_obsadzenia),
/*8*/      nie_atakuje(X,Ulokowani),
 % hetman w kol.1, wiersz X, nie atakuje hetmanów już ulokowanych.
/*9*/     hetmani(Xs,[X|Ulokowani],Nowa_lista_do_obsadzenia).
 % dlatego dołączamy go do ulokowanych.


 % Zasadnicza idea: dołączamy hetmana tylko wtedy,
 % gdy nie atakuje żadnego z hetmanów już ulokowanych.

/*10*/  usun(X,[X|Xs],Xs).
/*11*/  usun(X,[Y|Ys],[Y|Rs]):-
/*12*/     usun(X,Ys,Rs).

/*13*/  nie_atakuje(X,Ulokowani):-
/*14*/     nie_atakuje(X,Ulokowani,1).

/*15*/  nie_atakuje(_,[],_).
/*16*/  nie_atakuje(X,[Y|Ys],Nb):-
/*17*/     X=\=Y-Nb,
/*18*/     X=\=Y+Nb,
/*19*/     Nb1 is Nb+1,
/*20*/     nie_atakuje(X,Ys,Nb1).

/*21*/  wszystkie_rozwiazania:-
/*22*/     osiem_hetmanow(X),
/*23*/     write(X),nl,
/*24*/     fail.
/*25*/  wszystkie_rozwiazania:-
/*26*/     write("To wszystko!").
```

W liniach 6, 7, 8 i 9 mamy do czynienia ze znanym nam już usuwaniem głów z jednej listy (`Listy_wierszy_nieobsadzonych`) i dokładaniem ich do innej listy (`Listy_wierszy_obsadzonych`, początkowo pustej), przy czym numery wierszy są pobierane z "rezerwuaru" zwanego `Listą_wierszy_do_obsadzenia` za pomocą predykatu `usun/3`, usuwającego kolejno z `Listy_wierszy_do_obsadzenia` takie numery wierszy, które zapewniają nieatakowanie.

Drzewo poszukiwań i przebieg poszukiwań i dla prostszego przypadku czterech hetmanów i metody poszukiwań w głąb ze standardowymi nawrotami przedstawiono na rysunkach 2.9, 2.10 i 2.11.



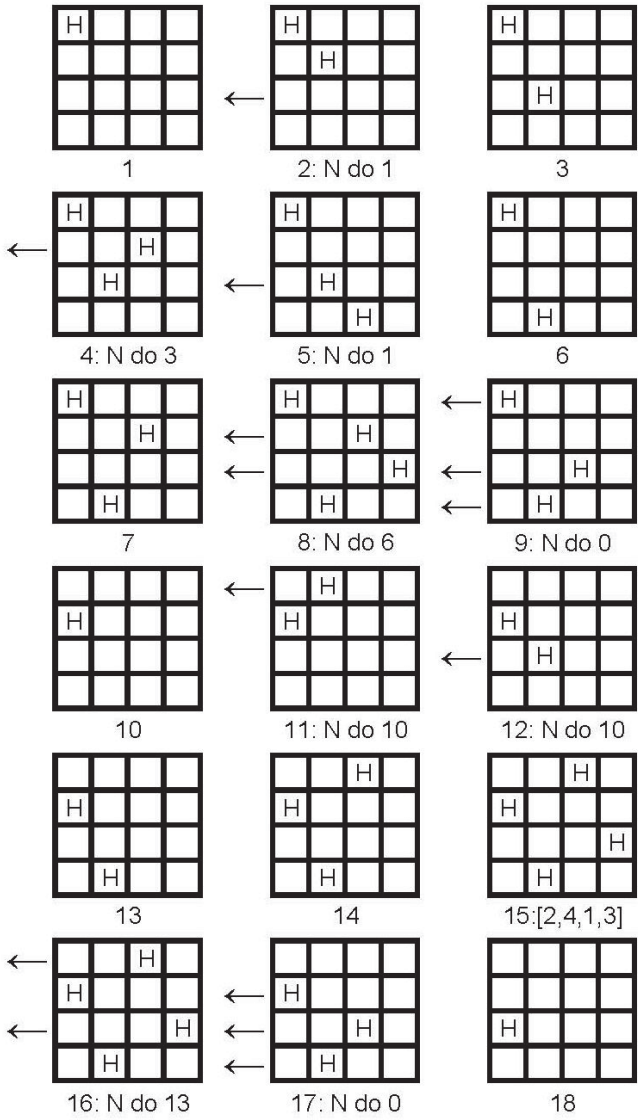
Rysunek 2.9: Drzewo poszukiwań w głąb ze standardowymi nawrotami dla czterech hetmanów

2.4.7 Egzamin - poszukiwania w głąb, standardowe nawroty

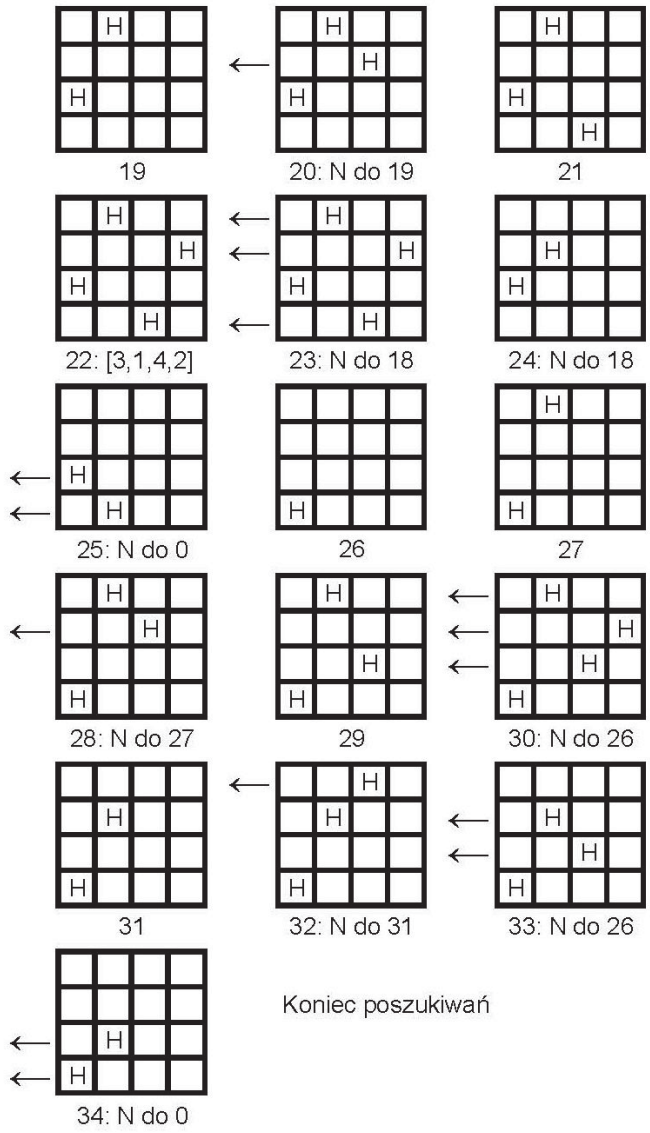
Bardzo często łamigłówki są dominowane przez "wiedzę negatywną" tzn. wiedzę o tym, czego nie wolno. Wiedza taka nie stwarza specjalnych problemów platformie ECL^iPS^e *Prolog*, co ilustruje poniższy przykład:

Sala egzaminacyjna ma 17 miejsc M1,...,M17 w układzie pokazanym na rysunku 2.12.

W sali tej będzie zdawało egzamin 17 studentów. Są cztery różne zestawy zadań: 1, 2, 3 i 4. Jak rozdzielić te zestawy pomiędzy miejsca, by na sąsiadujących miejscach (wzdłuż sali, wszerz sali i skośnie) nie było jednakowych zestawów? Czy jest możliwe uzyskanie takiego rozdziału zestawów w przypadku, gdy są tylko trzy różne zestawy zadań?
Odpowiedni program `2_15_egzamin.pl` ma postać:



Rysunek 2.10: Przebieg poszukiwań w głąb ze standardowymi nawrotami dla czterech hetmanów, część 1



Rysunek 2.11: Przebieg poszukiwań w głąb ze standardowymi nawrotami dla czterech hetmanów, część 2

Rysunek 2.12: Rozkład miejsc w sali egzaminacyjnej

```
/*0*/ top:-
/*1*/ L=[1,2,3,4],
/*2*/  member(M1,L),      /*3*/  member(M2,L),
/*4*/  member(M3,L),      /*5*/  member(M4,L),
/*6*/  member(M5,L),      /*7*/  member(M6,L),
/*8*/  member(M7,L),      /*9*/  member(M8,L),
/*10*/ member(M9,L),      /*11*/ member(M10,L),
/*12*/ member(M11,L),     /*13*/ member(M12,L),
/*14*/ member(M13,L),     /*15*/ member(M14,L),
/*16*/ member(M15,L),     /*17*/ member(M16,L),
/*18*/ member(M17,L),

/*19*/      M1 =\= M2,        /*20*/      M1 =\= M5,
/*21*/      M1 =\= M6,        /*22*/      M1 =\= M7,
/*23*/      M2 =\= M6,        /*24*/      M2 =\= M7,
/*25*/      M2 =\= M3,        /*26*/      M2 =\= M8,
/*27*/      M3 =\= M7,        /*28*/      M3 =\= M8,
/*29*/      M3 =\= M9,        /*30*/      M3 =\= M4,
/*31*/      M4 =\= M8,        /*32*/      M4 =\= M9,
/*33*/      M5 =\= M6,        /*34*/      M5 =\= M10,
/*35*/      M5 =\= M11,       /*36*/      M6 =\= M10,
/*37*/      M6 =\= M11,       /*38*/      M6 =\= M7,
/*39*/      M6 =\= M12,       /*40*/      M7 =\= M11,
/*41*/      M7 =\= M12,       /*42*/      M7 =\= M8,
/*43*/      M7 =\= M13,       /*44*/      M8 =\= M12,
/*45*/      M8 =\= M13,       /*46*/      M8 =\= M14,
/*47*/      M8 =\= M9,        /*48*/      M9 =\= M13,
/*49*/      M9 =\= M14,       /*50*/      M10 =\= M11,
/*51*/      M11 =\= M15,      /*52*/      M11 =\= M12,
```

```
/*53*/      M12 =\= M15,      /*54*/      M12 =\= M16,
/*55*/      M12 =\= M13,      /*56*/      M13 =\= M15,
/*57*/      M13 =\= M16,      /*58*/      M13 =\= M17,
/*59*/      M13 =\= M14,      /*60*/      M14 =\= M16,
/*61*/      M14 =\= M17,      /*62*/      M15 =\= M16,
/*63*/      M16 =\= M17,
/*64*/      write("   "),write(M1),write(", "),write(M2),
              write(", "),write(M3),write(", "),write(M4),nl,
/*65*/      write(M5),write(", "),write(M6),write(", "),write(M7),
              write(", "),write(M8),write(", "),write(M9),nl,
/*66*/      write(M10),write(", "),write(M11),write(", "),write(M12),
              write(", "),write(M13),write(","),write(M14),nl,
/*67*/      write("      "),write(M15),write(", "),write(M16),
              write(", "),write(M17), nl.
```

Jednym z wielu możliwych rozwiązań jest:

```
   1, 2, 1, 2
2, 3, 4, 3, 4
4, 1, 2, 1, 2
      3, 4, 3
```

Rozwiązanie to uzyskano po stosunkowo długim czasie (203 sekundy) na 2.0 GHz notebooku pod systemem operacyjnym Windows XP. W rozdziale 3.7.4 zostanie przedstawiony znacznie bardziej efektywny sposób rozwiązywania tego problemu w ramach platformy ECL^iPS^e CLP

Łatwo wykazać, że w przypadku trzech różnych zestawów zadań problem nie ma rozwiązania. Wystarczy w tym celu w liniach `2`,...,`18` zmienić listy na `[1,2,3]`. Rozpatrywane zadanie jest bowiem szczególnym przypadkiem aplikacji *twierdzenia o kolorowaniu krawędzi grafu płaskiego*[11], zgodnie z którym minimalna liczba kolorów potrzebna do pokolorowania krawędzi takiego grafu tak, by wszystkie sąsiadujące krawędzie grafu miały zawsze różne kolory, jest równa `4`.

[11] To bardzo ciekawe twierdzenie zostało - jako pierwsze - udowodnione wyłącznie metodami komputerowymi, a mianowicie za pomocą przeglądu zupełnego, patrz [Lines-95]

2.4.8 Błędne koła w Prologu

Rozpatrzmy znany paradoks[12] Bertranda Russela o fryzjerze: pewnemu fryzjerowi w małym miasteczku kazano, by golił wszystkich, którzy sami się nie golą. Czy fryzjer może sam siebie ogolić?

Łatwo stwierdzić, że cokolwiek fryzjer zrobi, naruszy nałożony nań warunek:

- jeżeli sam się ogoli, to jako golący się samemu mieszkaniec miasteczka nie powinien być golony przez fryzjera, a więc przez siebie;
- jeżeli sam się nie ogoli, to jako niegolący się samemu mieszkaniec miasteczka powinien być ogolony przez fryzjera, a więc przez siebie.

Mamy więc do czynienia z czymś, co się nazywa zwykle *błędnym kołem*. Jest ono spowodowane tym, że fryzjer jest również elementem zbioru *mieszkańcy miasteczka*: gdyby fryzjer przyjeżdżał z sąsiedniego miasteczka, mógłby się spokojnie albo golić samemu, albo korzystać z usług kolegi z tego innego miasteczka.

Gdybyśmy spróbowali "rozwiązać" paradoks fryzjera za pomocą programu prologowego, doprowadzi to do przepełnienia stosu, na który *Prolog* składa wyniki poszukiwań. Ilustruje to program `2_16_fryzjer.pl`:

```
/*1*/   top:-
/*2*/         goli(fryzjer,fryzjer).

/*3*/   goli(fryzjer,X):-
/*4*/         not(goli(X,X)).
```

Jego wykonanie generuje komunikat:

```
** Overflow of the local/control stack!
** You can use the "-l kBytes" (LOCALSIZE) option to have a larger
stack.
Peak sizes were: local stack 40384 kbytes, control stack 90688
kbytes*/
```

Przepełnienie stosu wynika stąd, że aby prawdą było `goli(fryzjer,fryzjer)`, *Prolog* usiłuje dowieść, że prawdą jest `not(goli(fryzjer,fryzjer))`, ale w tym celu należy wykazać, że nieprawdą jest `goli(fryzjer,fryzjer)`, a więc

[12] Paradoks jest zdaniem zawierającym efektowne lub zaskakujące sformułowanie, z którego wynika wniosek sprzeczny z powszechną opinią lub sprzeczny wewnętrznie. W rozpatrywanym przykładzie mamy do czynienia z ostatnim przypadkiem.

znów wraca do wniosku reguły `/*3*/`, itd. itd., a przy każdym powrocie zdanie `goli(fryzjer,fryzjer)` jest przechowywane na stosie, co doprowadzi koniec końców do jego przepełnienia. Zawarta w komunikacie rada o zwiększeniu rozmiarów stosu jest - dla rozpatrywanej sytuacji - całkowicie nieadekwatna.

Wypada jeszcze podkreślić, że reguła `/*3*/` nie jest rekurencją prologową, która zachodzi zawsze pomiędzy predykatem określonym dla listy (wniosek reguły rekurencyjnej), a tymże predykatem określonym dla ogona tej listy (warunek reguły rekurencyjnej).

Ponieważ *Prolog* (jak i poprawne ludzkie wnioskowanie) jest bezsilny wobec błędnych kół, należy ich unikać, podobnie jak unika się dzielenia przez zero w operacjach arytmetycznych. Pociechą może być to, że w poprawnie i kompletnie sformułowanych rzeczywistych problemach (np.: 1.Fryzjer goli sam siebie. 2.Pozostali mieszkańcy, nie golący się sami, są goleni przez fryzjera.), w przeciwieństwie do intelektualnych igraszek, nie spotyka się błędnych kół.

2.4.9 Jak zostać swym własnym dziadkiem?

Nicklaus Wirth w cenionym i popularnym podręczniku [Wirth-00] przedstawił następującą historię człowieka opisującego niedole swego życia:

Ożeniłem się z wdową, która miała dorosłą córkę. Mój ojciec, który odwiedzał nas często, zakochał się w mojej przybranej córce i ożenił się z nią. Tak więc mój ojciec stał się moim przybranym synem, a moja przybrana córka stała się moją matką (macochą).

Później moja żona urodziła mi syna, który stał się przybranym bratem mojego ojca i jednocześnie moim wujem.

Żona mojego ojca, czyli moja przybrana córka, miała także syna. W ten sposób uzyskałem brata i jednocześnie wnuka.

Moja żona jest moją babką, ponieważ jest matką mojej matki. Tak więc jestem mężem mojej żony i jednocześnie jej przybranym wnukiem.

Znajomi twierdzą, że jestem swoim własnym dziadkiem.

Czy to wszystko jest prawdą? Dla dowodu zakłada się, że *przybrana* relacja rodzinna jest tożsama z *normalną* relacją rodzinną, np. *przybrana córka = córka*, *przybrany brat = brat*, itp. Przyjmuje się następującą kolejność argumentów dla stosowanych predykatów:

```
ojciec(Ojciec,Syn), matka(Matka,Syn/Córka),
dziadek(Dziadek,Wnuk), babcia(Babcia,Wnuk),
brat(Ojciec,Brat_1,Brat_2),  wuj(Ojciec,Wuj,Bratanek).
```

Odpowiednie program (`2_17_dziadek.pl`) ma postać:

```
/*1*/  top:-
%              syn mój i żony jest przybranym bratem mojego ojca:
/*2*/       brat(_,moj_ojciec,syn_moj_i_zony),
%              syn mój i żony jest moim wujem:
/*3*/       wuj(syn_moj_i_zony,moj_ojciec,ja),
%              syn mojej przybranej córki jest moim bratem:
/*4*/       brat(_,syn_mojej_przybranej_corki,ja),
%              syn mojej przybranej córki jest moim wnukiem:
/*5*/       dziadek(ja,syn_mojej_przybranej_corki),
%              moja żona jest moją babką;
%              jestem więc przybranym wnukiem mojej żony:
/*6*/       babcia(moja_zona,ja),
%              jestem swoim własnym dziadkiem:
/*7*/       dziadek(ja,ja),nl,
/*8*/       write("Wszystko O.K.").

/*9*/  dziadek(Dziadek,Wnuk):-
/*10*/       ojciec(Dziadek,Syn_Dziadka),
/*11*/       ojciec(Syn_Dziadka,Wnuk).

/*12*/  babcia(Babcia,Wnuk):-
/*13*/       matka(Babcia,Corka_Babci),
/*14*/       matka(Corka_Babci,Wnuk).

%              jestem synem swego ojca:
/*15*/       ojciec(moj_ojciec,ja).
%              mój ojciec jest moim  przybranym synem:
/*16*/       ojciec(ja,moj_ojciec).
%              jestem ojcem syna mojej żony:
/*17*/       ojciec(ja,syn_moj_i_zony).
%              moja przybrana córka miała także syna:
/*18*/       ojciec(moj_ojciec,syn_mojej_przybranej_corki).

%              moja przybrana córka jest moją matką:
/*19*/       matka(moja_przybrana_corka,ja).
%              moja żona jest matką mojej przybranej córki:
/*20*/       matka(moja_zona,moja_przybrana_corka).

/*21*/  brat(Ojciec,Brat_1,Brat_2):-
/*22*/       ojciec(Ojciec,Brat_1),
/*23*/       ojciec(Ojciec,Brat_2).
```

```
/*24*/  wuj(Ojciec,Wuj,Bratanek):-
/*25*/        brat(_,Ojciec,Wuj),
/*26*/        ojciec(Wuj,Bratanek).
```

Po jego uruchomieniu otrzymuje się komunikat `Wszystko O.K.`, co znaczy, że wszystkie relacje wymienione jako warunki wniosku `top` są prawdziwe.

2.4.10 Stosowanie warunkowych predykatów

Podstawowy warunkowy predykat standardowy:

```
+Condition -> +Then ; +Else.
```

ma następujące właściwości:

- Na początku jest testowany warunek `Condition`. Jeżeli jest prawdą, wszystkie inne klauzule dla `Condition` zostają pominięte i predykat `Then` jest wywołany. Predykat `Else` nie jest w tym przypadku wywoływany niezależnie od wyniku wywoływania `Then`.
- Jeżeli `Condition` nie jest prawdą, wywoływany zostaje predykat `Else`. W tym przypadku predykat `Then` nie jest wywoływany.

W programach *prologowych* zamiast `Else` często występuje `True`, co umożliwia pominięcie predykatu warunkowego w przypadku niespełnienia warunku `Condition`.

Stosowanie warunkowego predykatu może uprościć pewne programy *prologowe*, co ilustruje następujący przykład:

Studenci Arek, Barbara i Czarek postanowili uczestniczyć na swym uniwersytecie na pewne wykłady dodatkowe. Każda z wymienionych osób wybrała inny wykład, odbywający się innego dnia, o różnych godzinach, w różnych salach, a mianowicie:

1) Arek wybrał wykład profesora Przybylskiego.
2) Wtorkowy wykład nie rozpoczyna się o godz. 14.00.
3) Wykład z "Inżynierii wiedzy" nie rozpoczyna się o godz. 17.30.
4) Czwartkowy wykład rozpoczyna się o godz. 15:45.
5) Krzysztof będzie uczęszczał na wykład z "Modeli ekonometrycznych".
6) Barbara chciałaby uczęszczać na wtorkowy wykład.
7) Wykład "Sztuczna inteligencja" odbywa się w sali D3.

8) Środowy wykład nie odbywa się w sali 104.
9) Profesor Kowalski nie jest wykładowcą "Modeli ekonometrycznych".
10)Profesor Jankowski nie ma swego wykładu w sali K2.

Program `2_25_wyklady.pl` wyznacza - dla każdego z wymienionych studentów - nazwę dodatkowego wykładu, wykładowcę, dzień i godzinę wykładu oraz miejsce wykładu:

```
/*1*/  top:-
/*2*/        studenci(Imie_1,Imie_2,Imie_3),
/*3*/        wyklady(Wyklad_1,Wyklad_2,Wyklad_3),
/*4*/        profesorowie(Profesor_1,Profesor_2,Profesor_3),
/*5*/        sale(Sala_1,Sala_2,Sala_3),
/*6*/        dni(Dzien_1,Dzien_2,Dzien_3),
/*7*/        godziny(Godzina_1,Godzina_2,Godzina_3),

/*8*/        ograniczenia(Imie_1,Wyklad_1,Profesor_1,Sala_1,Dzien_1,Godzina_1),
/*9*/        ograniczenia(Imie_2,Wyklad_2,Profesor_2,Sala_2,Dzien_2,Godzina_2),
/*10*/       ograniczenia(Imie_3,Wyklad_3,Profesor_3,Sala_3,Dzien_3,Godzina_3),

/*11*/       write(Imie_1),write(" preferuje wyklad "),write(Wyklad_1),
             write(" profesora "),write(Profesor_1), write(" w sali "),
             write(Sala_1), write(" on "), write(Dzien_1), write(" o godzinie"),
             write(Godzina_1), nl,

/*12*/       write(Imie_2),write(" preferuje wyklad "),write(Wyklad_2),
             write(" profesora "),write(Profesor_2), write(" w sali "),
             write(Sala_2), write(" on "), write(Dzien_2), write(" o godzinie"),
             write(Godzina_2),nl,

/*13*/       write(Imie_3),write(" preferuje wyklad "),write(Wyklad_3),
             write(" profesora "),write(Profesor_3),  write(" w sali "),
             write(Sala_3), write(" on "), write(Dzien_3), write(" o godzinie"),
             write(Godzina_3),nl.

/*14*/ studenci(Imie_1,Imie_2,Imie_3):-
/*15*/       imie(Imie_1),
/*16*/       imie(Imie_2),
/*17*/       imie(Imie_3),
/*18*/       wszystkie_rozne(Imie_1,Imie_2,Imie_3).

/*19*/ wyklady(Wyklad_1,Wyklad_2,Wyklad_3):-
/*20*/       wyklad(Wyklad_1),
/*21*/       wyklad(Wyklad_2),
/*22*/       wyklad(Wyklad_3),
/*23*/       wszystkie_rozne(Wyklad_1,Wyklad_2,Wyklad_3).
```

```
/*24*/ profesorowie(Profesor_1,Profesor_2,Profesor_3):-
/*25*/       profesor(Profesor_1),
/*26*/       profesor(Profesor_2),
/*27*/       profesor(Profesor_3),
/*28*/       wszystkie_rozne(Profesor_1,Profesor_2,Profesor_3).

/*29*/ sale(Sala_1,Sala_2,Sala_3):-
/*30*/       sala(Sala_1),
/*31*/       sala(Sala_2),
/*32*/       sala(Sala_3),
/*33*/       wszystkie_rozne(Sala_1,Sala_2,Sala_3).

/*34*/ dni(Dzien_1,Dzien_2,Dzien_3):-
/*35*/       dzien(Dzien_1),
/*36*/       dzien(Dzien_2),
/*37*/       dzien(Dzien_3),
/*38*/       wszystkie_rozne(Dzien_1,Dzien_2,Dzien_3).

/*39*/ godziny(Godzina_1,Godzina_2,Godzina_3):-
/*40*/       godzina(Godzina_1),
/*41*/       godzina(Godzina_2),
/*42*/       godzina(Godzina_3),
/*43*/       wszystkie_rozne(Godzina_1,Godzina_2,Godzina_3).

/*44*/ ograniczenia(Imie,Wyklad,Profesor,Sala,Dzien,Godzina):-
  % 1) Arek wybrał wykład  profesora Przybylskiego:
/*45*/      (  (Imie == "Arek")-> Profesor = "Przybylski"
/*46*/      ; true   ),

  % 2) Wtorkowy wykład nie rozpoczyna się o godz. 14.00
/*47*/      (  (Dzien == "wtorek")-> Godzina \== "14.00"
/*48*/      ; true   ),

  % 3) Wyklad "Inżynieria wiedzy"  nie rozpoczyna się o godzinie 17.30:
/*49*/      (  ( Wyklad == "Inzynieria wiedzy")->  Godzina \== "17:30"
/*50*/      ; true   ),

  % 4) Czwartkowy wyklad rozpoczyna sie o godzinie 15.45:
/*51*/      (  ( Dzien == "czwartek")->  Godzina = "15:45"
/*52*/      ; true   ),

  % 5) Krzysztof preferuje wyklad  "Modele ekonometryczne":
/*53*/      (  ( Imie == "Krzysztof")->  Wyklad = "Modele ekonometryczne"
/*54*/      ; true   ),

  % 6) Barbara chce uczeszczać na wyklad wtorkowy:
/*55*/      (  ( Imie == "Barbara") ->  Dzien = "wtorek"
/*56*/       ; true   ),
```

```
  % 7) Wyklad "Sztuczna inteligencja" odbywa się w sali D3:
/*57*/       (  ( Wyklad == "Sztuczna inteligencja") ->  Sala = "D3"
/*59*/      ; true   ),

  % 8) Środowy wyklad nie odbywa się  w sali 104:
/*59*/      ( ( Dzien == "sroda") -> Sala \== "104"
/*60*/      ; true   ),

  % 9) Profesor Kowalski nie jest wykładowcą "Modeli ekonometrycznych":
/*61*/      ( ( Profesor == "Kowalski") ->  Wyklad \== "Modele ekonometryczne"
/*62*/      ; true   ),

  % 10) Profesor Jankowski nie ma swego wykladu w sali K2.:
/*63*/      ( ( Profesor == "Jankowski") -> Sala \== "K2"
/*64*/      ; true   ).

/*65*/ wszystkie_rozne(Zmienna_1,Zmienna_2,Zmienna_3):-
/*66*/      Zmienna_1 \== Zmienna_2,
/*67*/      Zmienna_1 \== Zmienna_3,
/*68*/      Zmienna_2 \== Zmienna_3.

/*69*/ imie("Arek").
/*70*/ imie("Barbara").
/*71*/ imie("Krzysztof").

/*72*/ wyklad("Inzynieria wiedzy").
/*73*/ wyklad("Modele ekonometryczne").
/*74*/ wyklad("Sztuczna inteligencja").

/*75*/ profesor("Przybylskiego").
/*76*/ profesor("Kowalskiego").
/*77*/ profesor("Jankowskiego").

/*78*/ sala("D3").
/*79*/ sala("104").
/*80*/ sala("K2").

/*81*/ dzien("wtorkowy").
/*82*/ dzien("srodowy").
/*83*/ dzien("czwartkowy").

/*84*/ godzina("14.00").
/*85*/ godzina("17.30").
/*86*/ godzina("15.45").
```

Rozwiązaniem jest:

```
Andrzej preferuje wyklad srodowy 'Inzynieria wiedzy'
          profesora Przybylskiego w sali K2 o godzinie 14.00
Barbara preferuje wyklad wtorkowy 'Sztuczna inteligencja'
         profesora Kowalskiego w sali D3 o godzinie 17.30
Krzysztof preferuje wyklad czwartkowy 'Modele ekonometryczne'
          profesora Jankowskiego w sali 104 o godzinie 15.45
```

2.5 Problemy szeregowania

2.5.1 Chłop-wilk-koza-kapusta

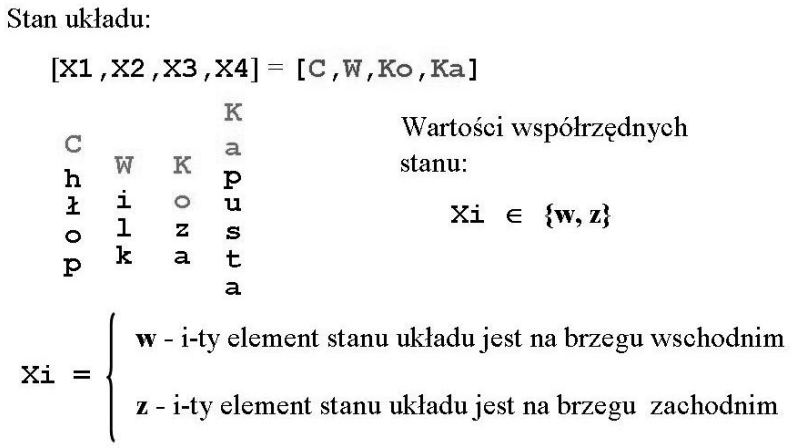
Popularna łamigłówka "Chłop-wilk-koza-kapusta" jest bardziej zaawansowanym przykładem poszukiwania trajektorii w przestrzeni stanu. Chłop z kozą, wilkiem i kapustą chce przeprawić się łódką z brzegu zachodniego rzeki na brzeg wschodni. W łódce są tylko dwa miejsca i tylko chłop umie wiosłować. Przeprawa jest narażona na dwa niebezpieczeństwa:

- Jeżeli po jednej stronie rzeki będzie tylko koza i kapusta, kapusta może zostać zjedzona przez kozę.
- Jeżeli po jednej stronie rzeki będzie tylko wilk i koza, koza może zostać zjedzona przez wilka.

Kto, w jakiej kolejności i w jakim kierunku ma przeprawić się przez rzekę, by wszyscy znaleźli się zdrowi i cali na brzegu wschodnim[13]?
Dla rozwiązania tego przykładu również dobrze jest posłużyć się pojęciem *stanu układu*, w którym skumulowana jest cała informacja potrzebna na danym etapie przeprawy dla wykonania poprawnego następnego ruchu. Stan układu "chłop-wilk-koza-kapusta" jest opisany przez podanie miejsca pobytu chłopa, wilka, kozy i kapusty, zob. rysunek 2.13. Przy przeprawach żaden stan nie może wystąpić dwukrotnie. Rozwiązanie wyznacza program `2_18_cwkk.pl`:

```
/*1*/  top:-
/*2*/       przepraw_wszystkich(stan(w,w,w,w),stan(z,z,z,z)),
/*3*/       writeln("Koniec ").
```

[13]Problem ten jest przypisywany osobie Alcuina z Yorku (730 - 804), anglosaskiemu uczonemu, duchownemu, poecie, matematykowi i nauczycielowi z Yorku w Northumbrii. Jest on autorem podręcznika *Propositiones ad Acuendos Iuvenes* (po polsku: *Problemy rozwijające młodzież*) zawierającego 53 łamigłówki, niektóre typu "przeprawy przez rzekę".



Rysunek 2.13: Stan układu chłop-wilk-koza-kapusta

```
/*4*/  przepraw_wszystkich(SP,SK):-
/*5*/       przeprawa_dozwolona(SP,SK,[SP],D),nl,
/*6*/       pisz_przeprawa_dozwolona(D),
/*7*/       fail.
/*8*/  przepraw_wszystkich(_,_):-
/*9*/       writeln("Wszyscy zostali przeprawieni bezpiecznie.").

/*10*/ przeprawa_dozwolona(SP,SK,D,D1):-
/*11*/      przeprawa(SP,S1),
/*12*/      not(niebezpieczny(S1)),
/*13*/      not(member(S1,D)),
/*14*/      przeprawa_dozwolona(S1,SK,[S1|D],D1).
                % Stan końcowy osiągnięty:
/*15*/ przeprawa_dozwolona(SK,SK,D,D):-!.

           % Chłop i wilk przeprawiają się z brzegu na brzeg,
            % koza i kapusta nie zmieniają miejsca pobytu:
/*16*/ przeprawa(stan(X,X,Ko,Ka),stan(Y,Y,Ko,Ka)):-
/*17*/      przeciwne_brzegi(X,Y).

           % Chłop i koza  przeprawiają się z brzegu na brzeg,
            % wilk i kapusta nie zmieniają miejsca pobytu:
/*18*/ przeprawa(stan(X,W,X,Ka),stan(Y,W,Y,Ka)):-
/*19*/      przeciwne_brzegi(X,Y).
```

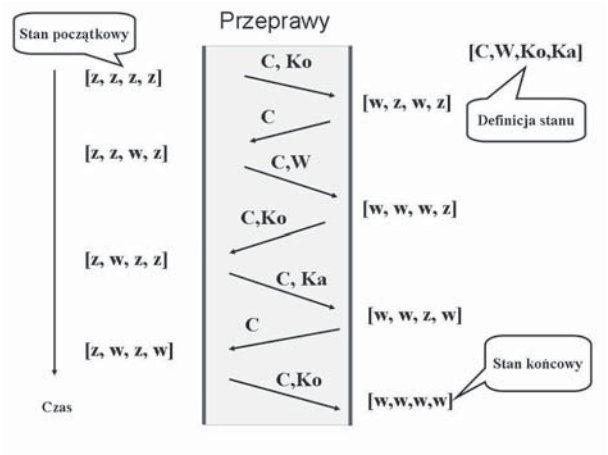
```
           % Chłop i kapusta przeprawiają się z brzegu na brzeg,
            % wilk i koza nie zmieniają miejsca pobytu:
/*20*/ przeprawa(stan(X,W,Ko,X),stan(Y,W,Ko,Y)):-
/*21*/      przeciwne_brzegi(X,Y).

           % Tylko chłop przeprawia się z brzegu na brzeg,
            % wilk, koza i kapusta nie zmieniają miejsca pobytu:
/*22*/ przeprawa(stan(X,W,Ko,Ka),stan(Y,W,Ko,Ka)):-
/*23*/      przeciwne_brzegi(X,Y).

               %  Wilk może zjeść kozę:
/*24*/ niebezpieczny( stan(Y,X,X,_) ):-
/*25*/      przeciwne_brzegi(Y,X).

               %  Koza może zjeść kapustę:
/*26*/ niebezpieczny( stan(Y,_,X,X) ):-
/*27*/      przeciwne_brzegi(Y,X).

/*28*/      przeciwne_brzegi(w,z).
/*29*/      przeciwne_brzegi(z,w):- !.

/*30*/ pisz_przeprawa_dozwolona([H1,H2|T]) :- !,
/*31*/      pisz_przeprawe(H1,H2),
/*32*/      pisz_przeprawa_dozwolona([H2|T]).
/*33*/      pisz_przeprawa_dozwolona([]).

/*34*/ pisz_przeprawe( stan(X,W,G,C), stan(Y,W,G,C) ):- !,
/*35*/      write("Chlop przeprawia sie z "),write(X),
                write(" na "),write(Y),nl.
/*36*/ pisz_przeprawe( stan(X,X,G,C), stan(Y,Y,G,C) ):- !,
/*37*/      write("Chlop bierze wilka z "),write(X),
                write(" na "),write(Y),nl.
/*38*/ pisz_przeprawe( stan(X,W,X,C), stan(Y,W,Y,C) ) :- !,
/*39*/      write("Chlop bierze koze z "),write(X),
                write(" na "),write(Y),nl.
/*40*/ pisz_przeprawe( stan(X,W,G,X), stan(Y,W,G,Y) ) :- !,
/*41*/      write("Chlop bierze kapuste z "),write(X),
                write(" na "),write(Y),nl.
```

Zadanie ma dwa rozwiązania, jedno z których pokazano na rysunku 2.14:

```
Chlop bierze koze z z na w          Chlop bierze koze z z na w
Chlop przeprawia sie z w na z       Chlop przeprawia sie z w na z
Chlop bierze kapuste z z na w       Chlop bierze wilka z z na w
Chlop bierze koze z w na z          Chlop bierze koze z w na z
```

```
Chlop bierze wilka z z na w
Chlop przeprawia sie z w na z
Chlop bierze koze z z na w
Wszyscy zostali przeprawieni
bezpiecznie.
Koniec
```

```
Chlop bierze kapuste z z na w
Chlop przeprawia sie z w na z
Chlop bierze koze z z na w
Wszyscy zostali przeprawieni
bezpiecznie.
```



Rysunek 2.14: Możliwe kolejne przeprawy chłopa, wilka, kozy i kapusty

Podstawowa rekurencja programu (dla predykatu `przeprawa_dozwolona/4`, linie 10,...14) jest rekurencją z dokładaniem głów. Ponownie okazuje się, że rekurencja taka jest charakterystyczna dla problemów wyznaczania trajektorii w przestrzeni stanu.

2.5.2 Misjonarze i kanibale

Łamigłówka "Misjonarze i kanibale" jest kolejną trudniejszą wersją wyznaczania trajektorii w przestrzeni stanu. Treść jej jest następująca:
Trzech misjonarzy i trzech kanibali musi przeprawić się przez rzekę z brzegu lewego na brzeg prawy, korzystając z łódki, która pomieści tylko dwóch pasażerów. Jeżeli misjonarze na którymś z brzegów będą w mniejszości, to zostaną zjedzeni przez kanibalów. Należy wyznaczyć taki sposób zorganizowania prze-

prawy na drugi brzeg, który to uniemożliwi[14]. Stan układu (zob. rysunek 2.15) jest tym razem opisany przez podanie
`Liczby misjonarzy na lewym brzegu`, `Liczby kanibali na lewym brzegu`
i `Lokalizacji łódki`. Przy przeprawach żaden stan nie może wystąpić dwukrotnie. Zmienna `Lokalizacja łódki` może przyjmować dwie wartości:
`lbl = łódka na brzegu lewym ; lbp = łódka na brzegu prawym.`

Stan układu:

[X1, X2, X3]

X1 – liczba misjonarzy na lewym brzegu

X2 – liczba kanibali na lewym brzegu

X3 – lokalizacja łódki

Wartości współrzędnych stanu:

$X1 \in \{1, 2, 3\}$

$X2 \in \{1, 2, 3\}$

$X3 \in \{lbl, lbp\}$,

lbl(p) łódka na brzegu lewym (prawym)

Rysunek 2.15: Stan układu misjonarze-kanibale

Odpowiedni program (`2_19_mik.pl`) korzysta z predykatu prywatnego:
```
przeprawa_dozwolona(Stan_początkowy,Stan_końcowy,
          Akumulator_drogi,Droga_odwrotna,
               Droga_poprawna,Lokalizacja_lodki)
```
i ma postać:

```
/*1*/  top:-
/*2*/       przepraw_wszystkich(stan(3,3,lbl), stan(0,0,lbp),lbl).

/*3*/  przepraw_wszystkich(SP,SK,Lokalizacja_lodki):-
/*4*/     przeprawa_dozwolona(SP,SK,[SP],_,Droga_poprawna,Lokalizacja_lodki),
```

[14]Ostatnio popularność zyskuje bardziej "politycznie poprawna" wersja tej łamigłówki, zgodnie z którą nie należy dopuścić do tego, by na którymś z brzegów kanibale byli w mniejszości; misjonarze mogą bowiem wtedy nawrócić kanibali.

```
/*5*/     nl, pisz_przeprawa_dozwolona(Droga_poprawna),
/*6*/     write("Przeprawe zakonczono."),nl,nl.

/*7*/  przeprawa_dozwolona(SP,SK,Aku,Droga,Droga_poprawna,Lokalizacja_przed):-
/*8*/     przeprawa(SP,S1,Lokalizacja_przed),
/*9*/     sprawdz(S1,Aku),
/*10*/    zmien_lokalizacje_lodki(Lokalizacja_przed,Lokalizacja_po),
/*11*/    przeprawa_dozwolona(S1,SK,[S1|Aku],Droga,Droga_poprawna,Lokalizacja_po),!.

      % Stan końcowy osiągnięty:
/*12*/ przeprawa_dozwolona(SK,SK,Droga,Droga,Droga_poprawna,_):-
/*13*/    reverse(Droga,Droga_poprawna),
/*14*/    write("Droga = "),write(Droga_poprawna),nl,!.

/*15*/ przeprawa(stan(X,K,lbl),stan(Y,K,lbp),lbl):-
/*16*/      Y is X-1.   % jeden misjonarz przepływa na prawy brzeg

/*17*/ przeprawa(stan(X,K,lbl),stan(Y,K,lbp),lbl):-
/*18*/      Y is X-2.   % dwóch misjonarzy przepływa na prawy brzeg

/*19*/ przeprawa(stan(M,X,lbl),stan(M,Y,lbp),lbl):-
/*20*/      Y is X-1.   % jeden kanibal przepływa na prawy brzeg

/*21*/ przeprawa(stan(M,X,lbl),stan(M,Y,lbp),lbl):-
/*22*/      Y is X-2.     % dwóch kanibali przepływa na prawy brzeg

/*23*/ przeprawa(stan(X,X1,lbl),stan(Y,Y1,lbp),lbl):-
/*24*/      Y is X-1, Y1 is X1-1.
            % misjonarz i kanibal przepływają na prawy brzeg

/*25*/ przeprawa(stan(X,K,lbp),stan(Y,K,lbl),lbp):-
/*26*/      Y is X+2.   % dwóch misjonarzy przepływa na lewy brzeg

/*27*/ przeprawa(stan(X,K,lbp),stan(Y,K,lbl),lbp):-
/*28*/      Y is X+1.   % jeden  misjonarz przepływa na lewy brzeg

/*29*/ przeprawa(stan(M,X,lbp),stan(M,Y,lbl),lbp):-
/*30*/      Y is X+1.   % jeden kanibal przepływa na lewy brzeg

/*31*/ przeprawa(stan(M,X,lbp),stan(M,Y,lbl),lbp):-
/*32*/      Y is X+2.   % dwóch kanibali przepływa na lewy brzeg

/*33*/ przeprawa(stan(X,X1,lbp),stan(Y,Y1,lbl),lbp):-
/*34*/      Y is X+1, Y1 is X1+1.
              % misjonarz i kanibal przepływają na lewy brzeg

/*35*/ sprawdz(Nowy_stan,Droga):-
/*36*/      not(niebezpieczny(Nowy_stan)),
/*37*/      not(niedozwolony(Nowy_stan)),
```

```
/*38*/      not(member(Nowy_stan,Droga)).

/*39*/ zmien_lokalizacje_lodki(lbp,lbl).
/*40*/ zmien_lokalizacje_lodki(lbl,lbp).

/*41*/ niebezpieczny(stan(M,K,_)):-  M>0, M<K.
/*42*/ niebezpieczny(stan(M,K,_)):-  M<3, K<M.
           % Jeżeli taka będzie sytuacja na jednym brzegu,
           % to na drugim brzegu kanibale będą mieć przewagę

/*43*/ niedozwolony(stan(M,_,_) ):-  M<0.
/*44*/ niedozwolony(stan(_,K,_)):-  K<0.
/*45*/ niedozwolony(stan(M,_,_)):- M>3.
/*46*/ niedozwolony(stan(_,K,_)):-  K>3.

/*47*/ pisz_przeprawa_dozwolona([H1,H2|T]):- !,
/*48*/      pisz_przeprawe(H1,H2),
/*49*/      pisz_przeprawa_dozwolona([H2|T]).
/*50*/ pisz_przeprawa_dozwolona([_]).

/*51*/ pisz_przeprawe(stan(X,K,_),stan(Y,K,_)):- Y is X+1,!,
/*52*/    write("Misjonarz przeplynął z prawego brzegu na lewy."),nl.

/*53*/ pisz_przeprawe( stan(M,X,_),stan(M,Y,_)):- Y is X+1,!,
/*54*/    write("Kanibal przeplynął z prawego brzegu na lewy."),nl.

/*55*/ pisz_przeprawe(stan(X,K,_),stan(Y,K,_)):- Y is X+2,!,
/*56*/    write("Dwoch misjonarzy przeplynelo z prawego brzegu na lewy."),nl.

/*57*/ pisz_przeprawe(stan(M,X,_),stan(M,Y,_)):- Y is X+2,!,
/*58*/    write("Dwoch kanibali przeplynelo z lewego brzegu  na prawy."),nl.

/*59*/ pisz_przeprawe(stan(X,X1,_),stan(Y,Y1,_)):-
/*60*/    Y is X+1,Y1 is X1+1,!,
/*61*/    write("Misjonarz i kanibal przeplyneli z prawego brzegu na lewy."),nl.

/*62*/ pisz_przeprawe(stan(X,K,_),stan(Y,K,_)):- Y is X-1,!,
/*63*/    write("Misjonarz przeplynal z lewego brzegu na prawy."),nl\sqrt{}.

/*64*/ pisz_przeprawe(stan(M,X,_),stan(M,Y,_)):- Y is X-1,!,
/*65*/    write("Kanibal przeplynal z lewego brzegu na prawy."),nl.

/*66*/ pisz_przeprawe(stan(X,K,_),stan(Y,K,_)):-Y is X-2,!,
/*67*/    write("Dwoch misjonarzy przeplynelo z lewego brzegu na prawy."),nl.

/*68*/ pisz_przeprawe(stan(M,X,_),stan(M,Y,_)):-Y is X-2,!,
/*69*/    write("Dwoch kanibali przeplynelo z lewego brzegu na prawy."),nl.
```

```
/*70*/ pisz_przeprawe(stan(X,X1,_),stan(Y,Y1,_)) :-
/*71*/      Y is X-1, Y1 is X1-1,!,
/*72*/    write("Misjonarz i kanibal przeplyneli z lewego brzegu na prawy."),nl.
```

Program generuje cztery rozwiązania: :

```
Droga = [stan(3,3,lbl), stan(3,1,lbp), stan(3,2,lbl),
stan(3,0,lbp), stan(3,1,lbl), stan(1,1,lbp), stan(2,2,lbl), stan(0,2,lbp),
stan(0,3,lbl), stan(0,1,lbp), stan(1,1,lbl), stan(0,0,lbp)]
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch misjonarzy przeplynelo z lewego brzegu na prawy.
     Misjonarz i kanibal przeplyneli z prawego brzegu na lewy.
     Dwoch misjonarzy przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Misjonarz przeplynal z prawego brzegu na lewy.
     Misjonarz i kanibal przeplyneli z lewego brzegu na prawy.
Przeprawy zakonczono.

Droga = [stan(3,3,lbl), stan(3,1,lbp), stan(3,2,lbl),
stan(3,0,lbp), stan(3,1,lbl), stan(1,1,lbp), stan(2,2,lbl), stan(0,2,lbp),
stan(0,3,lbl), stan(0,1,lbp), stan(0,2,lbl), stan(0,0,lbp)]
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch misjonarzy przeplynelo z lewego brzegu na prawy.
     Misjonarz i kanibal przeplyneli z prawego brzegu na lewy.
     Dwoch misjonarzy przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
Przeprawy zakonczono.

Droga = [stan(3,3,lbl), stan(2,2,lbp), stan(3,2,lbl),
stan(3,0,lbp), stan(3,1,lbl), stan(1,1,lbp), stan(2,2,lbl), stan(0,2,lbp),
stan(0,3,lbl), stan(0,1,lbp), stan(1,1,lbl), stan(0,0,lbp)]
     Misjonarz i kanibal przeplyneli z lewego brzegu na prawy.
     Misjonarz przeplynal z prawego brzegu na lewy.
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch misjonarzy przeplynelo z lewego brzegu na prawy.
     Misjonarz i kanibal przeplyneli z prawego brzegu na lewy.
```

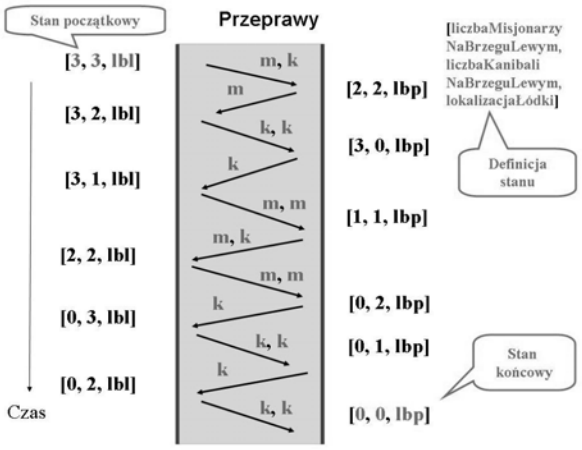
```
     Dwoch misjonarzy przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy.
     Dwoch kanibali przeplynelo z lewego brzegu na prawy.
     Misjonarz przeplynal z prawego brzegu na lewy.
     Misjonarz i kanibal przeplyneli z lewego brzegu na prawy.
Przeprawy zakonczono.

Droga = [stan(3,3,lbl), stan(2,2,lbp), stan(3,2,lbl),
stan(3,0,lbp), stan(3,1,lbl), stan(1,1,lbp), stan(2,2,lbl), stan(0,2,lbp),
stan(0,3,lbl), stan(0,1,lbp), stan(0,2,lbl), stan(0,0,lbp)]
     Misjonarz i kanibal przeplyneli z lewego brzegu na prawy.
     Misjonarz przeplynal z prawego brzegu na lewy
     Dwóch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy
     Dwóch misjonarzy przeplynelo z lewego brzegu na prawy.
     Misjonarz i kanibal przeplyneli z prawego brzegu na lewy
     Dwóch misjonarzy przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy
     Dwóch kanibali przeplynelo z lewego brzegu na prawy.
     Kanibal przeplynal z prawego brzegu na lewy
     Dwóch kanibali przeplynelo z lewego brzegu na prawy.
Przeprawe zakonczono.
```

Ostatnie rozwiązanie przedstawia rysunek 2.16. W tym przypadku, jak w poprzednich przykładach wyznaczania trajektorii w przestrzeni stanu, mamy również do czynienie z rekurencją z dokładaniem głów (linie 7,...,11).

2.5.3 Wieże z Hanoi

W szeregu przykładach korzystaliśmy z rekurencji. Szczególnie efektywnym i efektownym przykładem zastosowania rekurencji jest łamigłówka *wieże z Hanoi*. Łamigłówka ta została sformułowana przez francuskiego matematyka Edouarda Lucasa w drugiej połowie XIX wieku. Lucas zakładał istnienie wieży złożonej z 8 krążków o sukcesywnie malejącej średnicy, nadzianych na jeden z trzech prętów. Zadanie polega na przeniesieniu całej wieży na któryś z wolnych prętów, przy czym w każdym ruchu można brać tylko jeden krążek i nie wolno położyć większego krążka na mniejszy krążek. Rozpatrzmy ogólną łamigłówkę dla przypadku N krążków. Aby przenieść N krążków z pozycji początkowej na pozycję końcową, należy:

1. Przenieść $N-1$ krążków z pozycji początkowej na pozycję pośrednią z



Rysunek 2.16: Kolejne przeprawy misjonarzy i kanibali (ostatnie rozwiązanie)

wykorzystaniem pozycji końcowej. Załóżmy, że wymaga to T_{N-1} kroków.

2. Przenieść ostatni krążek z pozycji początkowej na pozycję końcową. Wykonaliśmy więc $T_{N-1}+1$ kroków.

Teraz jesteśmy w podobnej sytuacji jak przed krokiem 1 z tą różnicą, że mamy przenieść $N-1$ krążków z pozycji pośredniej na pozycję końcową z wykorzystaniem pozycji początkowej: obecność największego krążka na pozycji końcowej nie ma bowiem znaczenia dla dalszego przebiegu rozwiązania. Wymaga to również T_{N-1} kroków. Łącznie wykonaliśmy więc $T_N = 2T_{N-1}+1$ kroków. Równanie:

$$T_N = 2T_{N-1} + 1$$

ma rozwiązanie, którym jest $T_N = 2^N - 1$, co łatwo wykazać metodą indukcji matematycznej[15]. Odpowiedni program (`2_20_hanoi.pl`) ma postać:

```
/*1*/  top:-
/*2*/       write(" Podaj liczbe krazkow: "),nl,
/*3*/       read_token(Liczba, integer),
/*4*/       write(Liczba),nl,
```

[15]Dla $N = 0$ jest $T(0) = 0$. Jeżeli dla $N-1$ jest $2^{N-1}-1$, to $T(N) = 2T_{N-1}+1 = 2(2^{N-1}-1)+1 = 2^N - 1$.

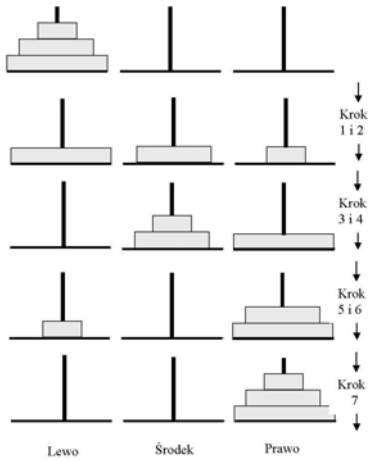
```
/*5*/       hanoi(Liczba).

/*6*/  hanoi(N) :-
/*7*/       przenies(N,"LEWO","SRODEK","PRAWO").

      % Pojedynczy krążek przenieś
      % z pozycji "LEWO" na pozycję
      % "PRAWO" bez korzystania
      % z pośredniej pozycji "SRODEK":
/*8*/  przenies(1,A,_,C) :-
/*9*/       pisz(A,C),
/*10*/      !.

      % W celu przeniesienia
      % N krążków  z pozycji "LEWO"
      % na pozycję "PRAWO",
/*11*/ przenies(N,A,B,C) :-

/*12*/      N1 is N-1,
      % przenieś N-1 krążków z pozycji
      % "LEWO" na pozycję "SRODEK"
      % z wykorzystaniem pośredniej
      % pozycji "PRAWO":
/*13*/      przenies(N1,A,C,B),

      % przenieś ostatni krążek
      % z pozycji "LEWO" na pozycję
      % "PRAWO":
/*14*/      pisz(A,C),

      % przenieś N-1 krążków z pozycji
      % "SRODEK" na pozycję "PRAWO"
      % z wykorzystaniem pośredniej
      % pozycji "LEWO":
/*15*/      przenies(N1,B,A,C).

/*16*/ pisz(Pozycja1,Pozycja2) :-
/*17*/      write("Przenies krazek z pozycji "),write(Pozycja1),
/*18*/      write(" na pozycje "),write(Pozycja2),write(".   "),nl.
```

Niecodziennym elementem tego programu jest podwójna rekurencja: `przenies(N,A,B,C)` z linii `/*11*/` jest definiowane przez `przenies(N1,A,C,B)` z linii `/*13*/` i `przenies(N1,B,A,C)` z linii `/*15*/`.

Przebieg działania programu dla 3 krążków jest następujący:

```
Podaj liczbe krazkow:  3
Przenies krazek z pozycji LEWO na pozycje PRAWO.
Przenies krazek z pozycji LEWO na pozycje SRODEK.
Przenies krazek z pozycji PRAWO na pozycje SRODEK.
Przenies krazek z pozycji LEWO na pozycje PRAWO.
Przenies krazek z pozycji SRODEK na pozycje LEWO.
Przenies krazek z pozycji SRODEK na pozycje PRAWO.
Przenies krazek z pozycji LEWO na pozycje PRAWO.
```

Przebieg przemieszczeń krążków przedstawiono na rysunku 2.18.



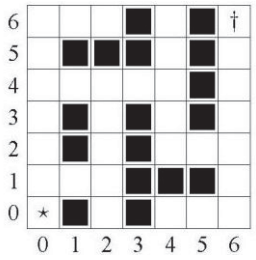
Rysunek 2.17: Rozwiązanie wież z Hanoi dla 3 krążków

Lucas uzupełnił swoją 8-krążkową łamigłówkę legendą o Wieży Brahmy, mającej 64 krążki ze złota, nadziane na 3 diamentowe pręty. U zarania czasu Brahma polecił grupie mnichów, by przełożyli je na któryś z pozostałych prętów zgodnie z podanymi już zasadami. Kiedy zrobią to, zgodnie z legendą nastąpi koniec świata. Legenda ta świadczy o dobrym wyczuciu złożoności obliczeniowej problemu przez Lucasa. Zakładając, że mnisi wykonują 1 ruch na sekundę, ułożenie wieży zajmie $2^{64} - 1 = 18446744073709551615$ (a więc blisko 18 i pół tryliona) sekund, czyli około 584 542 miliardów lat. Dla porównania: Wszechświat ma - zdaniem astronomów - około 13,7 mld lat.

2.6 Problemy szeregowania optymalnego

2.6.1 Prosty labirynt

Ważna grupą zadań prologowych jest wyznaczenie najkrótszej trajektorii w dyskretnej przestrzeni stanu, od wybranego stanu początkowego do wybranego stanu końcowego. Najprostszym przykładem tego typu zadania jest znalezienie drogi w labiryncie. Program z przykładu `2_21_labirynt.pl` znajduje najkrótszą drogę (mierzoną liczbą przebytych komórek) z punktu (0,0) do punktu (6,6) dla labiryntu pokazanego na rysunku 2.18. Dozwolone są jedynie tranzycje w pionie i w poziomie. W programie tym zastosowano również metodę *gałęzi i ograniczeń*, znaną z przykładu wyznaczania optymalnej konfiguracji, patrz 2.3.1.



Rysunek 2.18: Labirynt

Program `2_21_labirynt.pl` ma postać:

```
/*1*/  top:-
/*2*/       assert(droga_najkrotsza([[]],80)),
/*3*/       labirynt.

/*4*/       od_do([0,0],[0,1]).     /*5*/       od_do([0,1],[0,2]).
/*6*/       od_do([0,2],[0,3]).     /*7*/       od_do([0,3],[0,4]).
/*8*/       od_do([0,4],[0,5]).     /*9*/       od_do([0,5],[0,6]).
/*10*/      od_do([0,6],[1,6]).     /*11*/      od_do([1,6],[2,6]).
/*12*/      od_do([0,4],[1,4]).     /*13*/      od_do([1,4],[2,4]).
/*14*/      od_do([2,4],[3,4]).     /*15*/      od_do([3,4],[4,4]).
/*16*/      od_do([0,1],[1,1]).     /*17*/      od_do([1,1],[2,1]).
/*18*/      od_do([2,1],[2,2]).     /*19*/      od_do([2,2],[2,3]).
/*20*/      od_do([2,3],[2,4]).     /*21*/      od_do([4,4],[4,5]).
/*22*/      od_do([4,5],[4,6]).     /*23*/      od_do([4,4],[4,3]).
```

```
/*24*/      od_do([4,3],[4,2]).     /*25*/      od_do([4,2],[5,2]).
/*26*/      od_do([5,2],[6,2]).     /*27*/      od_do([6,2],[6,1]).
/*28*/      od_do([6,1],[6,0]).     /*29*/      od_do([6,0],[5,0]).
/*30*/      od_do([5,0],[4,0]).     /*31*/      od_do([6,2],[6,3]).
/*32*/      od_do([6,3],[6,4]).     /*33*/      od_do([6,4],[6,5]).
/*34*/      od_do([6,5],[6,6]).

/*35*/ tranzycja(A,B):-
/*36*/      od_do(A,B).
/*37*/ tranzycja(A,B):-
/*38*/      od_do(B,A).

/*39*/ labirynt:-
/*40*/      droga([[6,6]],Rozwiazanie_aktualne),
/*41*/      length(Rozwiazanie_aktualne,Dlugosc_aktualna),
/*42*/      aktualizuj_najkrotsza(Rozwiazanie_aktualne,
                    Dlugosc_aktualna),
/*43*/      fail.
/*44*/ labirynt:-
/*45*/      droga_najkrotsza(Rozwiazanie,Dlugosc),
/*46*/      write("Droga = "),write(Rozwiazanie),nl,
/*47*/      write("Dlugosc ="),write(Dlugosc),nl,nl,
/*48*/      fail.
/*49*/ labirynt:-
/*50*/      write("To wszystkie rozwiazania minimalizujace droge."),nl.

/*51*/ droga([Stan_obecny|Droga_przebyta],Rozwiazanie):-
/*52*/      tranzycja(Stan_obecny,Stan_następny),
/*53*/      not(member(Stan_następny,Droga_przebyta)),
/*54*/      droga([Stan_następny,Stan_obecny|Droga_przebyta],
                    Rozwiazanie).
/*55*/ droga([[0,0]|Droga_przebyta],[[0,0]|Droga_przebyta]).

/*56*/ aktualizuj_najkrotsza(Rozwiazanie_aktualne,Dlugosc_aktualna):-
/*57*/      droga_najkrotsza(_,Dlugosc),
/*58*/      Dlugosc_aktualna<Dlugosc,
/*59*/      retractall(droga_najkrotsza(_,_)),
/*60*/      assert(droga_najkrotsza(Rozwiazanie_aktualne,
                    Dlugosc_aktualna)),!.
/*61*/ aktualizuj_najkrotsza(Rozwiazanie_aktualne,
                Dlugosc_aktualna):-
/*62*/     droga_najkrotsza(_,Dlugosc),
/*63*/     Dlugosc_aktualna=Dlugosc,
/*64*/     assert(droga_najkrotsza(Rozwiazanie_aktualne,
                Dlugosc_aktualna)),!.
```

```
/*65*/ aktualizuj_najkrotsza(_,Dlugosc_aktualna):-
/*66*/     droga_najkrotsza(_,Dlugosc),
/*67*/     Dlugosc_aktualna>Dlugosc,!.
```

Rozwiązania mają postać:

```
Droga = [[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [1, 4], [2, 4],
[3, 4], [4, 4], [4, 3], [4, 2], [5, 2], [6, 2], [6, 3], [6, 4],
[6, 5], [6, 6]]
Dlugosc =17
Droga = [[0, 0], [0, 1], [1, 1], [2, 1], [2, 2], [2, 3], [2, 4],
[3, 4], [4, 4], [4, 3], [4, 2], [5, 2], [6, 2], [6, 3], [6, 4],
[6, 5], [6, 6]]
Dlugosc =17
```

To wszystkie rozwiazania minimalizujące drogę.

Pojęcie *stanu* w rozpatrywanym przykładzie jest oczywiste: stan jest dany każdorazowo przez współrzędne komórki labiryntu. W dalszym ciągu okaże się, jak bardzo ważne jest to pojęcie dla wszelkich problemów wyznaczenia trajektorii. Okaże się również, że definicja stanu może nie być czymś oczywistym.

Na uwagę zasługuje również to, że w liniach 51,...,54 mamy do czynienie z rekurencją z dokładaniem głów.

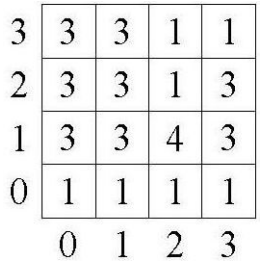
2.6.2 Pole minowe

Bardziej złożoną odmianą zadania "labirynt" jest zadanie "pole minowe", wymagające znalezienia najbezpieczniejszej drogi przez pole minowe (patrz rysunek 2.19) z punktu (0,0) do punktu (3,3).

Stan pola minowego jest - jak poprzednio - określony każdorazowo przez współrzędne komórki. W komórkach pola minowego wpisane są wartości `Niebezpieczeństwa` związane z tranzycjami przez komórkę. `Niebezpieczeństwa` nie należą do stanu, lecz są przez stan określone. Dozwolone są jedynie tranzycje w pionie i w poziomie.

`Niebezpieczenstwo całkowite` jest równe sumie `Niebezpieczenstw` przebytych kratek. Należy wyznaczyć drogę minimalizującą `Niebezpieczenstwo całkowite`. Odpowiedni program `2_22_pole_minowe.pl` ma postać:

```
/*1*/  top:-
/*2*/       assert(droga_najbezpieczniejsza([[]],50)),
/*3*/       pole_minowe.
```



Rysunek 2.19: Pole minowe

```
/*4*/       od_do([0,0],[0,1]).% kolumny, od dołu w górę
/*5*/       od_do([0,1],[0,2]).
/*6*/       od_do([0,2],[0,3]).

/*7*/       od_do([1,0],[1,1]).
/*8*/       od_do([1,1],[1,2]).
/*9*/       od_do([1,2],[1,3]).

/*10*/      od_do([2,0],[2,1]).
/*11*/      od_do([2,1],[2,2]).
/*13*/      od_do([2,2],[2,3]).

/*14*/      od_do([3,0],[3,1]).
/*15*/      od_do([3,1],[3,2]).
/*16*/      od_do([3,2],[3,3]).

/*17*/      od_do([0,0],[1,0]). % wiersze, od lewej w prawo
/*18*/      od_do([1,0],[2,0]).
/*19*/      od_do([2,0],[3,0]).

/*20*/      od_do([0,1],[1,1]).
/*21*/      od_do([1,1],[2,1]).
/*22*/      od_do([2,1],[3,1]).

/*23*/      od_do([0,2],[1,2]).
/*24*/      od_do([1,2],[2,2]).
/*25*/      od_do([2,2],[3,2]).

/*26*/      od_do([0,3],[1,3]).
/*27*/      od_do([1,3],[2,3]).
/*28*/      od_do([2,3],[3,3]).

/*29*/  tranzycja(A,B):-
```

```
/*30*/      od_do(A,B).
/*31*/  tranzycja(A,B):-
/*32*/      od_do(B,A).

/*33*/      niebezpieczenstwo([0,0],1).
/*34*/      niebezpieczenstwo([0,1],3).
/*35*/      niebezpieczenstwo([0,2],3).
/*36*/      niebezpieczenstwo([0,3],3).

/*37*/      niebezpieczenstwo([1,0],1).
/*38*/      niebezpieczenstwo([1,1],3).
/*39*/      niebezpieczenstwo([1,2],3).
/*40*/      niebezpieczenstwo([1,3],3).

/*41*/      niebezpieczenstwo([2,0],1).
/*42*/      niebezpieczenstwo([2,1],4).
/*43*/      niebezpieczenstwo([2,2],1).
/*44*/      niebezpieczenstwo([2,3],1).

/*45*/      niebezpieczenstwo([3,0],1).
/*46*/      niebezpieczenstwo([3,1],3).
/*47*/      niebezpieczenstwo([3,2],3).
/*48*/      niebezpieczenstwo([3,3],1).

/*49*/  pole_minowe:-
/*50*/      droga([[3,3]],Rozwiazanie),
/*51*/      niebezpieczenstwo_calkowite(Rozwiazanie,
                Niebezpieczenstwo_calkowite),
/*52*/      aktualizuj_najbezpieczniejsza(Rozwiazanie,
                Niebezpieczenstwo_calkowite),
/*53*/      fail.
/*54*/  pole_minowe:-
/*55*/      droga_najbezpieczniejsza(Rozwiazanie,
                Niebezpieczenstwo_calkowite),
/*56*/      write("Droga = "),write(Rozwiazanie),nl,
/*57*/      write("Niebezpieczenstwo_calkowite = "),
                write(Niebezpieczenstwo_calkowite),nl,nl,
/*58*/      fail.
/*59*/  pole_minowe:-
/*60*/      write("To wszystkie rozwiazania. "),nl.

/*61*/  droga([Stan_obecny|Droga_przebyta],Rozwiazanie):-
/*62*/      tranzycja(Stan_obecny,Stan_następny),
/*63*/      not(member(Stan_następny,Droga_przebyta)),
/*64*/      droga([Stan_następny,Stan_obecny|Droga_przebyta],
                Rozwiazanie).
/*65*/  droga([[0,0]|Droga_przebyta],[[0,0]|Droga_przebyta]).

/*66*/  niebezpieczenstwo_calkowite([H|T],N):-
```

```
/*67*/      niebezpieczenstwo_calkowite([H|T],N,0).
/*68*/  niebezpieczenstwo_calkowite([],N,N).
/*69*/  niebezpieczenstwo_calkowite([H|T],N,A):-
/*70*/      niebezpieczenstwo(H,NN),
/*71*/      A_Nowe is A+NN,
/*72*/      niebezpieczenstwo_calkowite(T,N,A_Nowe).

/*73*/  aktualizuj_najbezpieczniejsza(Rozwiazanie,Niebezpieczenstwo_calkowite):-
/*74*/      droga_najbezpieczniejsza(_,Niebezpieczenstwo_aktualne),
/*75*/      Niebezpieczenstwo_aktualne > Niebezpieczenstwo_calkowite,
/*76*/      retractall(droga_najbezpieczniejsza(_,_)),
/*77*/      assert(droga_najbezpieczniejsza(Rozwiazanie,
                Niebezpieczenstwo_calkowite)),!.

/*78*/  aktualizuj_najbezpieczniejsza(Rozwiazanie,Niebezpieczenstwo_calkowite):-
/*79*/      droga_najbezpieczniejsza(_,Niebezpieczenstwo_aktualne),
/*80*/      Niebezpieczenstwo_aktualne = Niebezpieczenstwo_calkowite,
/*81*/      assert(droga_najbezpieczniejsza(Rozwiazanie,
                Niebezpieczenstwo_calkowite)),!.

/*82*/  aktualizuj_najbezpieczniejsza(_,Niebezpieczenstwo_calkowite):-
/*83*/      droga_najbezpieczniejsza(_,Niebezpieczenstwo_aktualne),
/*84*/      Niebezpieczenstwo_aktualne < Niebezpieczenstwo_calkowite,!.
```

Program generuje następujący komunikat:

```
Droga = [[0, 0], [1, 0], [2, 0], [2, 1], [2, 2], [2, 3], [3, 3]]
Niebezpieczenstwo_calkowite = 10

To wszystkie rozwiazania minimalizujace niebezpieczenstwo.
```

Również w tym przykładzie w liniach 61,...,64 mamy do czynienie z rekurencją z dokładaniem głów.

2.6.3 Labirynt w Hampton Court

Często długości odcinków labiryntu nie są znane. Celem wówczas jest najczęściej znalezienie drogi z minimalną liczbą rozgałęzień. Pokazano to dla przykładu znalezienia drogi w labiryncie w Hampton Court. Jerome K. Jerome w znanej książce *Trzech panów w łódce (nie licząc psa)* opisuje przygody kogoś pragnącego znaleźć wyjście ze słynnego labiryntu żywopłotowego w Hampton Court koło Londynu[16], zob. rysunek 2.20:

"Harris zapytał mnie, czy byłem kiedy w labiryncie w Hampton Court. On sam wszedł tam raz, by kogoś oprowadzić. Przestudiował plan labiryntu - rzecz to była najprostsza pod słońcem, po prostu dziecinna igraszka nie warta dwóch pensów, jakie pobierano za wejście do środka. Doszedł do wniosku, że plan sporządzono tylko w celu nabierania gości, bo jego wartość równała się zeru i było to zwyczajne zawracanie głowy.

Rysunek 2.20: Plan labiryntu w Hampton Court

Osobą, którą Harris zabrał do labiryntu, był jego kuzyn z prowincji. - Wdepniemy tam na chwilkę - mówił Harris do krewniaka - byś mógł mówić, że byłeś w hamptońskim labiryncie, ale nic tam nie ma. Trzeba mieć źle w głowie, żeby to nazywać labiryntem. Cała sztuka w tym, żeby stale skręcać w pierwszy chodnik na prawo. Zrobimy sobie mały spacer w kółko - po dziesięciu minutach wyjdziemy i zjemy drugie śniadanie.

W chwilę po wejściu do środka Harris z kuzynem spotkali parę osób, które im powiedziały, że są tutaj od trzech kwadransów i że labirynt dał im się już we znaki. Harris odparł, że jeśli chcą, to mogą wszyscy iść za nim. Właśnie bowiem wybiera się na spacerek, zrobi kółko i wraca. Zbłąkana gromadka przyjęła z wdzięcznością zaproszenie Harrisa i ustawiła się za nim. Ruszyli razem. Po drodze zbierali zwiedzających, którzy szukali wyjścia, i pomału zgromadzili za sobą wszystkich ludzi z labiryntu. Ci, którzy porzucili już nadzieję, że dostaną się do środka lub że wyjdą kiedyś i zobaczą swój dom rodzinny i przyjaciół, nabierali otuchy na

[16]Tekst zaczerpnięto z polskiego wydania nakładem Młodzieżowej Agencji Wydawniczej, Warszawa 1986, w tłumaczeniu Kazimierza Piotrowskiego.

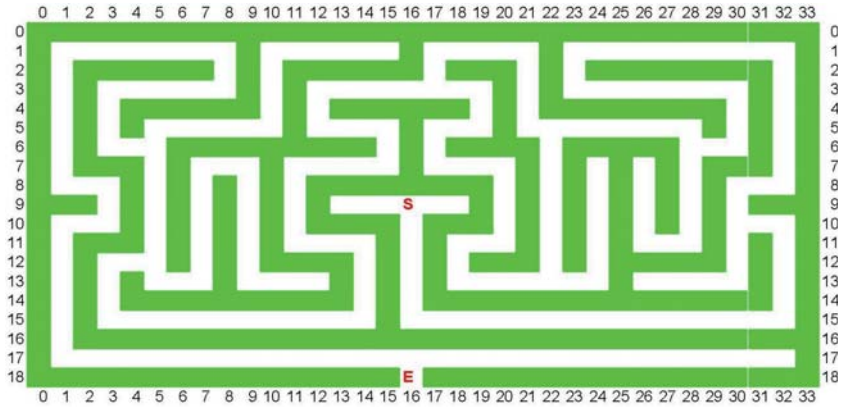
widok Harrisa i jego świty, przyłączali się do orszaku i błogosławili swemu przywódcy. Harris zapewniał potem, że w ślad za nim kroczyło co najmniej dwadzieścia osób. Jakaś kobiecina z dzieckiem na ręku, która błądziła tu od samego rana, uczepiła się ramienia Harrisa, drżąc na myśl, że może go stracić z oczu. Harris wciąż skręcał w prawo, drogi jednak nie ubywało. Kuzyn zauważył, że labirynt jest, zdaje się, bardzo duży. - Ba! Jeden z największych w Europie - odparł Harris.

- Chyba tak - kuzyn na to - bośmy zrobili już dobre dwie mile.

Harrisowi też zaczęło to się wydawać podejrzane, ale nic nie mówił. Gdy jednak zobaczyli na ziemi pół ciasteczka, a kuzyn przysiągł, że widział je przed siedmioma minutami, Harris bąknął "Och, niemożliwe", na co kobiecina z dzieckiem na ręku powiedziała: "Całkiem możliwe", bo sama wzięła dziecku kawałek ciastka i rzuciła na ziemię tuż przed spotkaniem Harrisa. Dodała też, że wolałaby nigdy go nie spotkać i że uważa go za zwykłego oszusta. Harris strasznie się rozgniewał. Rozłożył plan i objaśniał im, na czym polega jego system. - Plan bardzo by się przydał - zauważył ktoś z tłumu - gdybyś pan wiedział, gdzie jesteśmy teraz. Harris tego nie wiedział, odparł więc, że najlepiej będzie jeżeli powrócą do wejścia labiryntu i zaczną na nowo. Myśl, żeby zacząć na nowo, nie wzbudziła wielkiego entuzjazmu, natomiast wszyscy się ochoczo zgodzili, że trzeba wrócić do wejścia. Zrobili więc w tył zwrot i znów poszli z Harrisem w odwrotnym kierunku. Po dziesięciu minutach znaleźli się w samym środku labiryntu. Harris zrazu miał ochotę wmówić im, że o to właśnie chodziło. Ale tłum przyjął groźną postawę, przewodnik położył więc to na karb ślepego trafu. W każdym razie miejsce, w którym się znaleźli, można było przyjąć za punkt wyjścia. Wiedzieli nareszcie, gdzie są, jeszcze raz przestudiowali plan, wszystko wydało im się jasne jak słońce. Po raz trzeci ruszyli w drogę.

W trzy minuty później znów byli w samym środku labiryntu. Potem już absolutnie nigdzie nie mogli się dostać. W którąkolwiek stronę się zapuścili, zawsze wracali do środka labiryntu. Kursowali już tak regularnie, że niektórzy zatrzymali się tam i czekali, aż reszta odbędzie spacerek w koło i wróci do nich. Po jakimś czasie Harris znów wyciągnął swój plan, ale na sam jego widok tłum zaczął się gorączkować. Powiedziano mu, że może tego papieru użyć na papiloty i zrobić sobie baranka na głowie. Harris mówił potem, że, niestety, odniósł nieodparte wrażenie, iż stał się niepopularny. W końcu wszyscy dostali kręćka i nieprzytomnymi głosami wołali dozorcę labiryntu. Nadbiegł, wlazł na drabinę za ścianą i wykrzykiwał, co mają robić. Ale tak im się strasznie pomieszało w głowach, że już nic nie mogli zrozumieć. Dozorca więc kazał im czekać na miejscu i obiecał, że do nich przyjdzie. Zbili się w wystraszoną gromadkę i czekali. Dozorca zlazł z drabiny i wszedł do labiryntu. Nieszczęście chciało, że był to młody dozorca, który od niedawna tu pracował. Wszedł, ale nie mógł ich znaleźć i rychło sam zabłądził. Co chwila ukazywał się im w oddali - widać było, jak idzie pod drugiej stronie żywopłotu. Ujrzawszy ich, szybkim krokiem zmierzał ku nim, czekali więc przez pięć minut, aby znów ujrzeć go dokładnie w tym samym miejscu co przedtem. Pytał się wówczas, gdzie się podziali. Wyszli dopiero wtedy, gdy jeden ze starych dozorców wrócił z obiadu."

Dla celów modelowania dobrze jest przedstawić labirynt w postaci pokazującej współrzędne wszystkich rozwidleń jak na rysunku 2.21.
Poniżej przedstawiono program `2_23_hampton_court.pl` znajdowania najkrótszej drogi w labiryncie w Hampton Court od środka labiryntu (oznaczonego literą S) do wyjścia z labiryntu (oznaczonego literą E). Długość drogi jest mierzona liczbą przebytych nie rozwidlających się odcinków. Odcinek nie rozwidlający się jest odcinkiem łączącym jedno rozwidlenie z następnym.



Rysunek 2.21: Współrzędne rozwidleń labiryntu

Program `2_23_hampton_court.pl` jest następujący:

```
/*1*/  top:-
/*2*/       assert(droga_najkrotsza([[]],80)),
/*3*/       labirynt.

  %  Model labiryntu jest opisany predykatami:
  %  'od_do(Współrzędne_rozwidlenia, Współrzędne_sąsiedniego rozwidlenia)':

/*4*/       od_do([18,16],[17,16]).
/*5*/       od_do([17,16],[17,32]).
/*6*/       od_do([17,16],[6,5]).
/*7*/       od_do([6,5],[1,15]).
/*8*/       od_do([6,5],[12,5]).
/*9*/       od_do([12,5],[13,12]).
/*10*/      od_do([12,5],[3,17]).
/*11*/      od_do([3,17],[5,22]).
/*12*/      od_do([3,17],[13,22]).
```

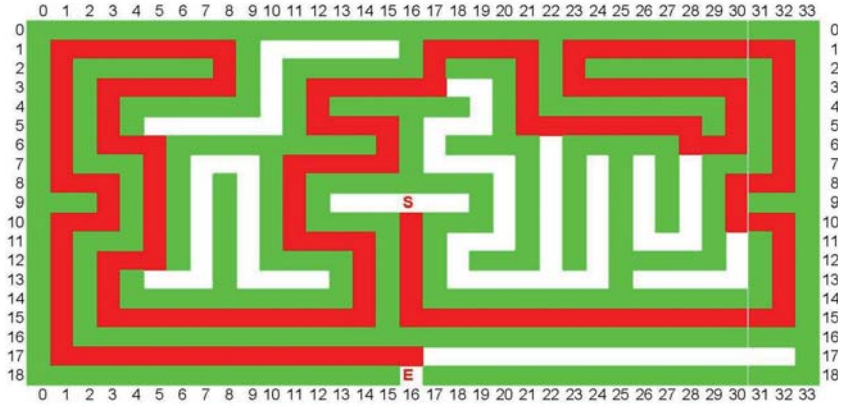
```
/*13*/      od_do([13,22],[7,24]).
/*14*/      od_do([13,22],[5,22]).
/*15*/      od_do([5,22],[6,28]).
/*16*/      od_do([6,28],[7,26]).
/*17*/      od_do([6,28],[10,30]).
/*18*/      od_do([10,30],[13,26]).
/*19*/      od_do([10,30],[9,16]).

/*20*/ tranzycja(A,B):-
/*21*/      od_do(A,B).
/*22*/ tranzycja(A,B):-
/*23*/      od_do(B,A).

/*24*/  labirynt:-
/*25*/      droga([[18,16]],Rozwiazanie_aktualne),
/*26*/      length(Rozwiazanie_aktualne,Dlugosc_aktualna),
/*27*/      aktualizuj_najkrotsza(Rozwiazanie_aktualne,Dlugosc_aktualna),
/*28*/      fail.

/*29*/  labirynt:-
/*30*/      droga_najkrotsza(Rozwiazanie,Dlugosc),
/*31*/      write("Najkrotsza droga to "),write(Rozwiazanie),nl,
/*32*/      write("Jej dlugosc (mierzona liczba rozwidlen) jest rowna "),
                        write(Dlugosc),nl,
/*33*/      fail.

/*34*/  labirynt:-
/*34*/      write("To wszystkie rozwiazania!"),nl.

/*35*/  droga([Stan_obecny|Droga_przebyta],Rozwiazanie):-
/*36*/     tranzycja(Stan_obecny,Stan_nastepny),
/*37*/      not(member(Stan_nastepny,Droga_przebyta)),
/*38*/      droga([Stan_nastepny,Stan_obecny|Droga_przebyta],Rozwiazanie).
/*39*/      droga([[9,16]|Droga_przebyta],[[9,16]|Droga_przebyta]).

/*40*/  aktualizuj_najkrotsza(Rozwiazanie_aktualne,Dlugosc_aktualna):-
/*41*/      droga_najkrotsza(_,Dlugosc),
/*42*/      Dlugosc_aktualna<Dlugosc,
/*43*/      retractall(droga_najkrotsza(_,_)),
/*44*/      assert(droga_najkrotsza(Rozwiazanie_aktualne,Dlugosc_aktualna)),!.

/*45*/  aktualizuj_najkrotsza(Rozwiazanie_aktualne, Dlugosc_aktualna):-
/*46*/      droga_najkrotsza(_,Dlugosc),
/*47*/      Dlugosc_aktualna=Dlugosc,
/*48*/      assert(droga_najkrotsza(Rozwiazanie_aktualne, Dlugosc_aktualna)),!.
```

```
/*49*/  aktualizuj_najkrotsza(_,Dlugosc_aktualna):-
/*50*/       droga_najkrotsza(_,Dlugosc),
/*51*/       Dlugosc_aktualna>Dlugosc,!.
```

Rozwiązaniem jest:

```
/*10*/ Najkrotsza droga to:
/*10*/ [[9,16], [10,30], [6,28], [5,22], [3,17], [12,5], [6,5], [17,16], [18,16]]
/*10*/ Jej dlugosc (mierzona liczba rozwidlen) jest rowna  9
/*10*/ To wszystkie rozwiazania!
```

Najkrótszą drogę przedstawia rysunek 2.22.



Rysunek 2.22: Najkrótsza droga dla labiryntu w Hampton Court

2.6.4 Przelewanie

Ostatnim przykładem generowania trajektorii w przestrzeni stanów jest przykład o przelewaniu. Przykładów o przelewaniu jest stosunkowo dużo. Wybraliśmy następujący:

Mamy do dyspozycji trzy naczynia pozbawione podziałki o pojemnościach ośmiu, pięciu i trzech litrów. Jak należy przelewać wodę, aby osiem litrów z naczynia ośmiolitrowego rozmieścić po połowie w naczyniach ośmio- i pięciolitrowym, przy wykorzystaniu tylko wymienionych naczyń i przy minimalnej

liczbie przelewań? Rozwiązanie przedstawia program `2_24_przelewanie.pl`:

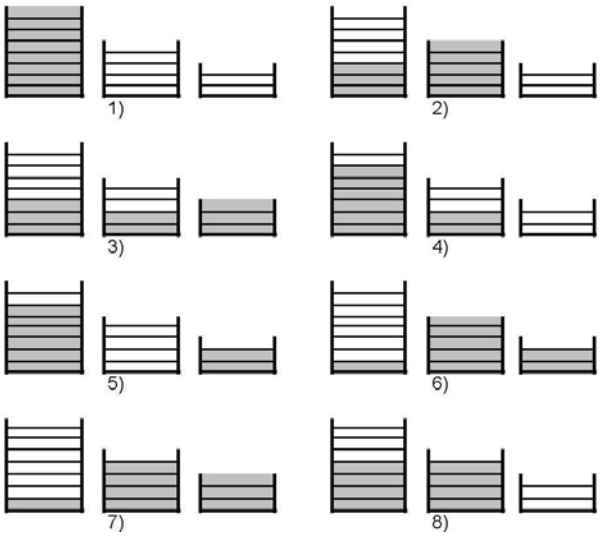
```
/*1*/  top:-
/*2*/       Stan_poczatkowy = stan(8,0,0),
/*3*/       przelej(Stan_poczatkowy,Ciag_stanow),
/*4*/       length(Ciag_stanow, N),
/*5*/       assert(trajektoria_stanow(N,Ciag_stanow)),
/*6*/       fail.
/*7*/  top:-
/*8*/  assert(najkrotsza_trajektoria_stanow(20,[])),
/*9*/  wyznacz_optymalne.

/*10*/ przelej(Stan_poczatkowy,Ciag_stanow):-
/*11*/  przelej(Stan_poczatkowy,[Stan_poczatkowy],Ciag_stanow).

/*12*/ przelej(Stan,Akumulator,Ciag_stanow):-
/*13*/  zmien(Stan,Nowy_stan),
/*14*/  not(member(Nowy_stan,Akumulator)),
/*15*/  przelej(Nowy_stan,[Nowy_stan|Akumulator],Ciag_stanow).

/*16*/ przelej(Stan_koncowy,Akumulator,Akumulator):-
/*17*/  stan_koncowy(Stan_koncowy),!.

/*18*/  stan_koncowy(stan(4,4,0)).

 % Możliwe przelewania:
 % Przelewanie z 1 do 2, w 2 może być co najwyżej 5 litrów:
/*19*/ zmien(stan(X,Y,Z),stan(K,L,Z)):-
/*20*/  przelewanie(X,Y,5,K,L).

 % Przelewanie z 2 do 1, w 1 może być co najwyżej 8 litrów:
/*21*/ zmien(stan(X,Y,Z),stan(K,L,Z)):-
/*22*/  przelewanie(Y,X,8,L,K).

 % Przelewanie z 1 do 3, w 3 mogą być conajwyżej 3 litry:
/*23*/ zmien(stan(X,Y,Z),stan(K,Y,M)):-
/*24*/  przelewanie(X,Z,3,K,M).

 % Przelewanie z 3 do 1, w 1 może być co najwyżej 8 litrów:
/*25*/ zmien(stan(X,Y,Z),stan(K,Y,M)):-
/*26*/  przelewanie(Z,X,8,M,K).

 % Przelewanie z 2 do 3, w 3 mogą być co najwyżej 3 litry:
/*27*/ zmien(stan(X,Y,Z),stan(X,L,M)):-
/*28*/  przelewanie(Y,Z,3,L,M).

 % Przelewanie z 3 do 2, w 2 może być co najwyżej 5 litrów:
/*29*/ zmien(stan(X,Y,Z),stan(X,L,M)):-
```

```
/*30*/  przelewanie(Z,Y,5,M,L).        %

 % przelewanie(Z_objętości_A, Do_objętości_B,
 %           Z_limitem_na_wypełnienie_B, Nowe_A, Nowe_B)

/*31*/ przelewanie(X,Y,LimitY,K,L):-
/*32*/  sprawdz(X,Y,LimitY),
/*33*/  !,
/*34*/  NX is X - 1,
/*35*/  NY is Y + 1,
/*36*/  przelewanie(NX,NY,LimitY,K,L).
/*37*/ przelewanie(X,Y,_,X,Y).

/*38*/ sprawdz(X,Y,Limit):-
/*39*/  X > 0,
/*40*/  Y < Limit,!.

/*41*/ wyznacz_optymalne:-
/*42*/  ciag_stanow_optymalny,
/*43*/  najkrotsza_trajektoria_stanow(N,Ciag_stanow),
/*44*/  reverse(Ciag_stanow, Ciag_odwrotny),
/*45*/  write("Optymalne rozwiazanie problemu : "),nl,
/*46*/  write(Ciag_odwrotny),nl,
/*47*/  write("Calkowita liczba przelewan: "),write(N).

/*48*/ ciag_stanow_optymalny:-
/*49*/  trajektoria_stanow(X,Trajektoria_X),
/*50*/  najkrotsza_trajektoria_stanow(Y,Trajektoria_Y),
/*51*/  aktualizuj(X,Y,Trajektoria_X,Trajektoria_Y),
/*52*/  fail.
/*53*/  ciag_stanow_optymalny.

/*54*/ aktualizuj(X,Y,Trajektoria_X,Trajektoria_Y):-
/*55*/  X < Y,
/*56*/  !,
/*57*/  retract(najkrotsza_trajektoria_stanow(Y,Trajektoria_Y)),
/*58*/  assert(najkrotsza_trajektoria_stanow(X,Trajektoria_X)).
/*59*/ aktualizuj(_,_,_,_).
```

Program generuje komunikat:

```
Optymalne  rozwiazanie problemu:
[stan(8,0,0),stan(3,5,0),stan(3,2,3),stan(6,2,0),
 stan(6,0,2),stan(1,5,2),stan(1,4,3),stan(4,4,0)]
 Calkowita liczba przelewan: 8
```

Przebieg przelewań przedstawiono na rysunku 2.23. Na rysunku tym - dla zobrazowania poprawności przelewań - narysowano podziałki, których naczynia nie mają, i które nie zostały wykorzystane w programie.



Rysunek 2.23: Przebieg przelewań dla 3 naczyń

2.7 Zadania

Dziedziny

W programach prologowych deklaracje dziedzin są z reguły niejawne i dokonywane w różnych dziwnych miejscach. Znajdź dziedziny zmiennych decyzyjnych dla wszystkich programów niniejszego rozdziału.

Pozycja elementu w liście

Napisz definicję predykatu:

```
pozycja(++Lista, ++Element, -Pozycja)
```

wyznaczającego pozycję elementu w liście różnych elementów.

Liczby Fibonacci'ego

Leonardo Fibonacci (c. 1170 – c. 1250) był znanym matematykiem i nauczycielem matematyki w Republice Pisa, będącej obecnie częścią Włoch. Jest znany głównie z prób modelowania rozwoju populacji królików, które w jego czasach były wielce pożądanym źródłem mięsa i futer.

Fibonacci zakładał, że nowo-narodzona para królików uzyska zdolność rozrodczą po miesiącu, tak że po upływie drugiego miesiąca samica może urodzić następną parę królików. Zakładając, że króliki żyją bardzo długo i że każda para królików reprodukuje następną parę w rytmie co miesiąc od drugiego miesiąca ich życia poczynając, liczba par króliczych wzrasta w rytmie miesięcznym następująco:

`0,1,1,2,3,5,8,13,21,34,55,89,144,...`,

Oznaczając liczbę par króliczych na początku miesiąca n-tego przez F_n (zwaną liczbą Fibonacciego), wzrost liczby par króliczych można opisać podwójną rekurencją:

$$F_n = F_{n-1} + F_{n-2}$$

gdzie: $F_0 = 0, \quad ,F_1 = 1$.

Napisz program prologowy wyznaczający liczby Fibonacciego za pomocą rekurencji nie-ogonowej, oraz program wyznaczający te liczby za pomocą rekurencji ogonowej, z wykorzystaniem akumulatorów.

Przyjaciółki Czarka

Czarek ma pięć przyjaciółek:
1)Anna jest 27-letnią blondynką, lekarką, zamężną, matką chłopca i dziewczynki. 2)Barbara jest 20-letnią blondynką, studentką singielką, bezdzietną, lubiącą gotować. 3)Celina jest 24-letnią brunetką, gospodynią domową, zamężną, bezdzietną, amatorską aktorką. 4)Danka jest 21-letnią blondynką, pracującą jako sekretarka, jest rozwódką, ma córkę. 5)Ewa jest 25-letnią blondynką, dyplomowaną pielęgniarką, bezdzietną rozwódką, lubiącą muzykę klasyczną. Zastosuj `findal/3` dla wyznaczenia danych wszystkich tych przyjaciółek Czarka, które nie są rozwódkami, nie są starsze niż 24 lata i mają niesportowe hobby.

Wieczór gier

W Parafialnym Wieczorze Gier, w konkurencjach Scrabble i szachy, uczestniczyło czterech chłopców. Arek wygrał z Markiem w szachy, Kuba był trzeci, a wygrał 16-latek. Arek był drugi w Scrabble, które wygrał 15-latek, Kuba pokonał 18-latka a 19-latek był trzeci. Bolek jest 3 lata młodszy niż

Marek. Chłopiec, który był ostatni w szachach, był trzeci w Scrabble. Tylko jeden chłopiec zajął to samo miejsce w obydwu grach. Napisz program wyznaczający wiek chłopców i ich miejsca w obydwu grach.

Recital

W Szkolnym Recitalu pięciu studentów (Arek, Ewa, Bolek, Danka i Czarek) odegrało pięć utworów: dwa Bacha, dwa Mozarta i jeden Vivaldiego. Występowało trzech skrzypków i dwóch pianistów. Każdy student odegrał tylko jeden utwór i grał tylko na jednym instrumencie. Wyznacz kolejność występów studentów, instrumenty na których grali i nazwiska kompozytorów, których utwory wykonywali, jeżeli:
1. Kompozytorzy nie byli wykonywani kolejno; Vivaldiego grano jako ostatniego, a Mozarta jako pierwszego. 2. Jeden utwór na fortepian został wykonany pomiędzy dwoma utworami na skrzypce, a dwa utwory na skrzypce zostały wykonane pomiędzy pierwszym i ostatnim utworem na fortepian. 3. Nie wykonywano utworów fortepianowych Mozarta. 4. Ewa grała trzecia. 5. Czarek grał na fortepianie, tuż po Arku, który grał Mozarta. 6. Danka nie grała utworu Vivaldiego.

Kursy mistrzowskie [17]

Słynny mezzo-sopran Flora Nebbiacorno wycofała się z międzynarodowej sceny operowej, lecz ciągle organizuje kursy mistrzowskie. W ostatnim kursie było pięciu studentów: jeden sopran, jeden mezzo-sopran, dwóch tenorów i jeden bas. Pierwsze dwa głosy to panie, ostatnie dwa - to panowie. Ich imiona to Chris, J.P., Lee, Pat i Val; każde z tych imion może być męskim lub damskim. Ich nazwiska to Kingsley, Robinson, Robinson (nie spokrewnieni, o tym samym nazwisku), Ulrich i Walker. Napisz program wyznaczający kolejność śpiewania na kursie oraz identyfikujący każdego uczestnika kursu z imienia, nazwiska i rodzaju głosu, jeżeli wiadomo, że: 1.Pierwszymi śpiewającymi byli (w nieznanej kolejności) Pat i bas. 2.Drugi i trzeci student to co najmniej jeden tenor. 3.Kingsley i piąty student (który nie nazywa się Robinson) to, w nieznanej kolejności, mezzo-sopran i tenor. 4.Ani trzeci student (o nazwisku Robinson), ani Walker, nie mają imienia Chris. 5.Ulrich nie jest basem ani mezzo-sopranem. 6.Ani Lee ani Val (który/która nie był/była na miejscu trzeci) nie jest tenorem. 7.J.P. nie był/była na miejscu trzecim, Chris nie był/była na miejscu piątym. 8 Bas nie nazywa się Robinson.

[17] Przykład pochodzi z http://brownbuffalo.sourceforge.net/

Konkurs konfiturowy

Na ostatnim Międzywydziałowym Konkursie Konfiturowym czterech kandydatów zakwalifikowało się do Finału Konfitur Truskawkowych. Wiek tych kandydatów to 14, 17, 20, 22. Tak się złożyło, że osoba, która zajęła ostatnie miejsce, była najstarsza, natomiast Czarek był o trzy lata starszy od osoby, która zajęła drugie miejsce. Bolek nie był ani najstarszy, ani najmłodszy, a Arek uplasował się przed 17-latkiem, lecz nie wygrał konkursu. Darkowi tym razem się nie poszczęściło i również nie wygrał konkursu. Napisz program wyznaczający miejsce i wiek każdego uczestnika finału konkursu.

Spotkania brydżowe

Cztery panie spotykają się co tydzień na partyjce brydża. Na każdym spotkaniu umawiają się, co każda przyniesie na następne spotkanie. Ostatnio postanowiono, że:
1. Pani Kowalska przyniesie tort czekoladowy. 2. Ani Pani Bogucka, ani Wanda ani Anna Cichocka nie przyniosą wypieków. 3. Roma, której nazwiskiem nie jest Dzielska, przyniesie kawę. 4. Maria nie przyniesie wina. Napisz program wyznaczający imię i nazwisko każdej z pań i to, co przyniesie na przyszło-tygodniową partyjką brydża.

Dwa naczynia

Masz dwa naczynia, naczynie 4-litrowe i naczynie 3-litrowe. Żadne z naczyń nie ma podziałki dla zawartości. Masz do dyspozycji kran, którym możesz się posłużyć dla napełniania naczyń wodą. Napisz program wyjaśniający, jak otrzymać w naczyniu 4-litrowym dokładnie 2 litry wody.

Statki

Na redzie portu jest pięć statków[18], niebawem wypływających w morze: 1. Grecki statek wypływa o szóstej i przewozi kawę. 2. Statek w środku ma czarny komin. 3. Angielski statek wypływa o dziewiątej. 4. Francuski statek z niebieskim kominem jest z lewej strony statku przewożącego kawę. 5. Z prawej strony statku przewożącego kakao jest statek, który popłynie do Marsylii. 6. Brazylijski statek popłynie do Manili. 7. Obok statku przewożącego ryż jest statek z zielonym kominem. 8. Statek do Genui wypływa o piątej. 9. Statek hiszpański wypływa o siódmej i jest na prawo od statku płynącego do Marsylii. 10. Statek z czerwonym kominem popłynie do Hamburga. 11. Obok statku wypływającego o siódmej jest statek z białym

[18] Przykład z http://www.mathsisfun.com/puzzles.

kominem. 12. Statek na granicy redy przewozi zboże. 13. Statek z czarnym kominem wypływa o ósmej. 14. Statek przewożący zboże jest obok statku przewożącego ryż. 15. Statek do Hamburga wypływa o szóstej.

Napisz program wyznaczający statek, który popłynie do Port Said i statek, który przewozi herbatę.

Przeprawa przez rzekę 1 [19]

Czterech podróżników (Alek, Bolek, Czarek i Darek) muszą przepłynąć rzekę w małej łódce. Łódka może przewieźć tylko 100 kg. Alek waży 90 kg, Bolek waży 80 kg, Czarek waży 60 kg a Darek 40 kg, i łącznie mają 20 kg niepodzielnego bagażu. Napisz program wyznaczający taką marszrutę, dla której wszyscy podróżnicy i ich bagaż bezpiecznie przepłyną rzekę.

Rozwiązanie:
Czarek i Darek przepływają rzekę, Darek powraca. Alek przepływa rzekę, Czarek powraca. Czarek i Darek przepływają rzekę ponownie, Darek powraca. Bolek przepływa rzekę z bagażem, Czarek powraca. Czarek i Darek przepływają rzekę ponownie po raz ostatni.

Przeprawa przez rzekę 2

Trzech podróżników i trzy małpy (jedna duża, dwie małe) muszą przepłynąć przez rzekę. Jest tylko jedna łódka i tylko podróżnicy lub duża małpa są wystarczająco silni, by wiosłować. Ponadto, liczba małp na którymkolwiek brzegu nie może być większa od liczby podróżników na tym brzegu; w przeciwnym przypadku małpy zaatakują podróżników. Napisz program wyznaczający taką marszrutę, dla której wszyscy przepłyną rzekę i żaden z podróżników nie zostanie zaatakowany przez małpy.

Przeprawa przez rzekę 3

Po jednej stronie rzeki jest rodzina składająca się z ojca, matki, syna, córki, pomocy domowej i psa. Muszą przeprawić się na drugi brzeg rzeki, dysponują tylko jedną małą łódką. Łódka umożliwia przewiezienie tylko dwóch osób lub jednej osoby i psa. Wyprawa musi spełnić jeszcze dodatkowe ograniczenia:

- tylko ojciec, matka i pomoc domowa potrafią wiosłować;
- ojciec nie może być sam z synem, bo zaraz będzie mu czynił gorzkie wymówki z powodu jego ślamazarstwa;

[19] Zadanie z http://www.mathsisfun.com/puzzles.

- matka nie może być sama z córką, bo zaraz będzie jej czynić gorzkie wymówki z powodu jej bałaganiarstwa;
- pomoc domowa musi stale być z psem, który w przeciwnym przypadku ugryzie kogokolwiek w pobliżu.

Napisz program wyznaczający taką marszrutę, dla której wszyscy przepłyną rzekę, nikt nie zostanie pogryziony przez psa i nikt nie będzie musiał wysłuchiwać gorzkich wymówek.

Przeprawa przez rzekę 4

Trzy pary małżeńskie AA, BB i CC (panowie Andrzej, Bogdan i Cezary, i odpowiednio panie Anna, Barbara i Celina) muszą przeprawić się przez rzekę łódką, która pomieści tylko dwie osoby. Niestety, żaden z panów nie ufa swej małżonce na tyle, by ją zostawić z którymś z pozostałych panów bez swojej obecności. Poza tym:
- Andrzej nie może przeprawiać się sam, bo boi się rzeki;
- Anna nie może wiosłować, bo jest w ciąży;
- Barbara nie może wiosłować, bo ma złamaną rękę na temblaku;
- Andrzej i Cezary nie mogą przeprawić się razem, bo się serdecznie nie lubią;
- z tych samych powodów Andrzej i Cezary nie chcą zostać sami na jednym z brzegów;
Napisz program prologowy wyznaczający marszrutę czyniącą zadość fobiom i ograniczeniom każdego z uczestników przeprawy, zakładając że każdy pan i Celina mogą wiosłować,

Kłamcy

Wiadomo, że tylko jedna z wymienionych w dalszym ciągu osób mówi prawdę. Pan April mówi, że pan May kłamie. May twierdzi, że pan June mówi kłamstwa. June uważa, że kłamie zarówno pan April jak i pan May. Napisz program określający, kto z wymienionych trzech osób mówi prawdę.

Zwierzaki

Na ostatnim spotkaniu Pets Anonymous obecni mówili o zwierzakach, które ostatnio mieli i obecnie mają w domach. Janek miał psa. Osoba, która miała mysz, teraz ma kota, lecz osoba która miała kota nie ma teraz myszy. Karol ma obecnie, lub miał (nie pamiętam dobrze) psa. Bożena nigdy nie miała myszy. Tylko jedna z osób ma obecnie tego samego zwierzaka, co poprzednio. Roma nie odzywała się w trakcie spotkania i nikt nie

wspomniał chomika. Napisz program określający, jakiego zwierzaka każdy z obecnych miał i ma obecnie.

Wyścigi ślimaków

Po ostatnich wyścigach ślimaków, czterej właściciele "zawodników" gratulowali sobie sukcesów. Tylko jeden ślimak miał taki sam numer, jak miejsce, które zajął w wyścigu. Ślimak Alfred nie był pomalowany na żółto ani na niebiesko. Ślimak z numerem 3, pomalowany na czerwono, pokonał ślimak który był trzeci. Ślimak Artura pokonał ślimaka Anny, a ślimak Alicji pokonał ślimaka z numerem 1. Pomalowany na zielono ślimak Alicji był drugi. Ślimak o kolorze niebieskim miał numer 4, a ślimak Anny miał numer 1. Napisz program określający miejsca ślimaków na mecie, ich numery i ich kolory.

Zawody

Panowie Rzeźnik, Piekarz, Stolarz i Malarz spotkali się po raz pierwszy od czasu ukończenia szkoły. Każdy z nich w międzyczasie raz zmieniał zawód. Nikt z nich nie jest obecnie zatrudniony ani nie pracował w zawodzie na jaki wskazuje jego nazwisko, oraz nikt z nich nie pracował w danym zawodzie dwa razy. Ani poprzednio, ani obecnie, nikt nie miał takiego samego zawodu, co ktoś z pozostałych. Czarek nigdy nie był stolarzem a pan Rzeźnik jest teraz malarzem. Darek był rzeźnikiem, natomiast pan Bolek Piekarz nigdy nim nie był. Pan Malarz nie ma na imię Edek i pan Stolarz nie był dotychczas rzeźnikiem. Napisz program, który wskaże imiona i nazwiska uczestników spotkania wraz z ich obecnymi i poprzednimi zawodami.

Szkolne eliminacje lekkoatletyczne

W Szkolnych Eliminacjach Lekkoatletycznych uczestniczyło czterech biegaczy w dwóch biegach na dystansie 400 m. Żaden z nich nie ukończył obydwu biegów na tej samej pozycji. John w obydwu biegach pokonał zawodnika o nazwisku Donald. Steve Curtail był trzeci w biegu drugim a Dave był ostatni w biegu pierwszym. Drugi bieg wygrał zawodnik nazwiskiem Arnold, a zawodnik nazwiskiem Bowler był ostatni. W pierwszym biegu Steve pokonał Keva, lecz Kev pokonał Johna. Napisz program wyznaczający (z imienia i nazwiska) miejsca zajęte przez uczestników obydwu biegów.

Zawodnik	Bieg 1	Bieg 2
Steve Curtail	1	3
Kev Bowler	2	4
John Arnold	3	1
Dave Donald	4	2

Tablica 2.8: Rozwiązanie zadania "Szkolne eliminacje lekkoatletyczne"

Dziewięcioro studentów

Dziewięcioro studentów (Alek, Bolek, Czarek, Darek, Edek, Fred, Gerd, Henryk i Jan) mieszka w 3-piętrowym niewielkim akademiku o trzech pokojach na każdym piętrze: jeden pokój jest w Skrzydle Zachodnim, drugi jest w Centrum, a trzeci w Skrzydle Wschodnim. Jeżeli popatrzysz na akademik z ulicy, jego lewa strona jest Skrzydłem Zachodnim, a prawa - Skrzydłem Wschodnim. Każdy student zajmuje jeden pokój. Napisz program określający, w którym pokoju mieszka każdy student, jeżeli:

1. Henryk nie mieszka na parterze.
2. Fred mieszka bezpośrednio nad Janem i bezpośrednio obok Bolka, który mieszka w Skrzydle Zachodnim.
3. Edek mieszka w Skrzydle Wschodnim, piętro wyżej aniżeli Fred.
4. Darek mieszka bezpośrednio nad Fredem.
5. Gerd mieszka bezpośrednio nad Czarkiem.

Beczki wina

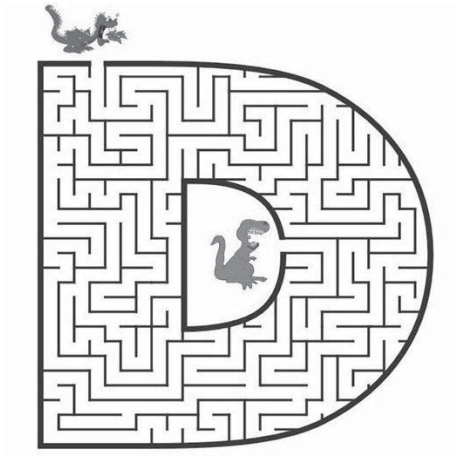
Właściciel winnicy pozostawił w spadku swym trzem synom 23 beczki (z tego 7 jest wypełnionych winem, 7 jest napełnionych winem w połowie, a pozostałe 7 to beczki puste), żądając w testamencie, by dokonać takiego podziału, który każdemu synowi zapewni tą samą liczbę beczek pełnych, beczek w połowie napełnionych i beczek pustych. Dokonania podziału spadku utrudnia brak przyrządów pomiarowych: można jedynie stwierdzić, czy beczka jest pusta, czy jest pełna lub też czy jest w połowie napełniona. Napisz program wyjaśniający, jak można dokonać podziału spadku.

Labirynt ze smokiem i dinozaurem [20]

Dla labiryntu z rysunku 2.24 napisz program umożliwiający smokowi dotarcie do dinozaura. Długość drogi jest mierzona liczbą przebytych nie

[20] Przykład z http://krazydad.com/mazes/

rozwidlających się odcinków. Odcinek nie rozwidlający się jest odcinkiem łączącym jedno rozwidlenie z następnym.



Rysunek 2.24: Labirynt ze smokiem i dinozaurem

Powitania

Arek i Alicja prosili swych przyjaciół na kolację. Niektórzy z ich przyjaciół zjawili się ze swymi małżonkami, inni przyszli sami. Każdy z gości przywitał się z parą gospodarzy i z każdym z pozostałych gości. Gdy dwaj panowie się witali, wymieniali uścisk dłoni. Gdy dwie panie się witały, całowały się. Pocałunek był wymieniany również wtedy, gdy pan witał się z panią. wiadomo, że wymieniono 6 uścisków dłoni i 12 pocałunków. Napisz program wyznaczający liczbę gości-panów, liczbę gości-pań, oraz liczbę par małżonków i liczbę samotnych gości. Jest oczywistym, że gdy goście byli małżonkami, małżonkowie nie witali się ze sobą.

Politycznie poprawni misjonarze i kanibale

Zmodyfikuj program `2_19_mik.pl` tak, by spełniał kryteria poprawności politycznej wzmiankowane w odnośniku dla podrozdzialu 2.5.2.

Kradzież dzieł sztuki

Po głośnej kradzieży dzieł sztuki policja przesłuchiwała sześciu podejrzanych. Poniżej jest skrót ich zeznań:

Arek powiedział: To nie był Bolek. To nie był Darek. To nie był Edek.
Bolek powiedział: To nie był Arek. To nie był Czarek. To nie był Edek.
Czarek powiedział: To nie był Bolek. To nie był Fred. To nie był Edek.
Darek powiedział: To nie był Arek. To nie był Fred. To nie był Czarek.
Edek powiedział: To nie był Czarek. To nie był Darek. To nie był Fred.
Fred powiedział: To nie był Czarek. To nie był Darek. To nie był Arek

Policja wie, że dokładnie czterech podejrzanych powiedziało po jednym kłamstwie, a wszystkie pozostałe stwierdzenia są prawdziwe. Napisz program wyznaczający złodzieja (złodziei?).

Współzawodnictwo

Pięciu przyjaciół stara się o pracę w Dużej Międzynarodowej Korporacji. Po wszystkich rozmowach i egzaminach, kandydatom przedstawiono wyniki. Przygodny świadek tego mimochodem usłyszał, że:

- Arek z przykrością skonstatował, że nie jest na pierwszym miejscu;
- Bolek przyznał, że był trzeci za Czarkiem;
- Arek dodał,że Czarek nie był ani pierwszym, ani ostatnim w rankingu;
- Darek przyznał, że był zaraz za Arkiem.

Czy przygodny świadek ma wystarczająco dużo danych dla sklasyfikowania wszystkich pięciu przyjaciół? Napisz program wyjaśniający tę sprawę.

Tajni współpracownicy [21]

Sześciu tajnych współpracowników byłej komunistycznej służby bezpieczeństwa mieszka w tym samym sześcio-piętrowym apartamentowcu. Każdy z panów-współpracowników (Alek, Bolek i Czarek) i każda z pań-współpracowniczek (Danka, Ewa i Fela) mieszkają na różnych piętrach. Nazwiska współpracowników to Marzec, Kwiecień, Maj, Czerwiec, Lipiec i Sierpień. Napisz program określający imiona i nazwiska wszystkich współpracowników oraz liczbę napisanych przez nich donosów, jeżeli wiadomo, że:

[21] Przykład oparty na http://brownbuffalo.sourceforge.net/

- napisali łącznie 180 donosów, a każdy współpracownik napisał co najmniej jeden donos;
- Czarek mieszka piętro poniżej Lipca;
- nazwiskiem Ewy nie jest ani Czerwiec, ani Maj;
- Fela mieszka na wyższym piętrze niż Danka, lecz na niższym niż Czerwiec;
- Bolek mieszka dokładnie nad Sierpniem i dokładnie poniżej tej osoby, która napisała 40 donosów;
- Marzec nie mieszka ani na pierwszym, ani na szóstym piętrze;
- Alek napisał połowę liczby donosów napisanych przez osobę mieszkającą na szóstym piętrze, która to osoba napisała połowę tej liczby donosów co Kwiecień;
- Danka nie mieszka na pierwszym piętrze;
- Bolek napisał 20 donosów więcej niż Kwiecień;
- Kwiecień nie ma na imię Danka;
- Sierpień napisał o 10 donosów mniej niż Marzec.

Rozdział 3

CLP z predykatami elementarnymi dla rozwiązań dopuszczalnych

3.1 Klasyfikacja predykatów

Różnorodność predykatów standardowych (tzn. zaprojektowanych przez twórców języka i dostępnych dla programistów języka) jest dla języków *CLP* znacznie większa, a ich "moc modelowania" znacznie poważniejsza, aniżeli dla predykatów standardowych języka *Prolog*. Wyróżnia się następujące grupy predykatów standardowych:

- *Predykaty elementarne*, definiujący podstawowe relacje z bibliotek `lib(ic)` i `lib(branch_and_bound)`, o zmiennych decyzyjnych, zawartych co najwyżej w jednej liście;
- *Predykaty globalne*, definiujące złożone relacje z bibliotek `lib{ic_global}`, `lib(ic_cumulative)`, `lib(ic_edge_finder)`, `lib(ic_edge_finder3)`, których *argumenty* są na ogół zawarte w szeregu *listach*.

Predykaty elementarne są podstawowymi cegiełkami dla tworzenia programów CLP, ich właściwości i sposoby ich stosowania zasługują na baczną uwagę. Dlatego właśnie od nich zaczynamy rozważania na temat CLP, odkładając rozważania nad właściwościami i zastosowaniami predykatów globalnych do dal-

szych rozdziałów 4 i 6. Stosując bibliotekę `lib(ic)` należy symbole operacji arytmetycznych i relacyjnych dla zmiennych dyskretnychch prefiksować symbolem `#`.

3.2 Czym różnią się języki CLP od Prologu?

3.2.1 Zasadnicze różnice

1. W *Prolog*u dziedziny zmiennych były deklarowane *implicite*, za pomocą elementów list znajdujących się w różnych predykatach w różnych miejscach programu. Programy *CLP* zawierają, w swej części wstępnej, jawne deklaracje dziedzin dla wszystkich zmiennych decyzyjnych programu. Są one listami dopuszczalnych wartości zmiennych problemu. Dziedziny te - ze względu na wymogi stosowanych metod propagacji ograniczeń - są (w przypadkach zadań kombinatorycznych) dziedzinami liczb całkowitych. Skutkuje to tym, że rozwiązania otrzymuje się również w postaci list liczb całkowitych, które - dla czytelności uzyskanego rozwiązania - wymagają konwertowania na nazwy zmiennych. Jest to cena płacona za ogromny wzrost efektywności poszukiwań uzyskany dzięki stosowaniu propagacji metodami *technik zgodnościowych*.

2. *Prolog* mógł przetwarzać tylko zmienne z dziedziny termów. Języki *CLP* mogą przetwarzać zmienne ze znacznie szerszego wachlarza zmiennych, np. zmienne całkowite, zmienne symboliczne, zmienne rzeczywiste, zmienne zbiorowe.

3. W programach prologowych propagacja ograniczeń była przeprowadzana na drodze *propagacji wartości zmiennej* i *unifikacji*. Języki *CLP* stosują znacznie bardziej efektywne metody unifikacji znane pod nazwą *technik zgodnościowych*.

4. W językach *CLP* stosowane są bardziej efektywne metody poszukiwań aniżeli prologowe poszukiwania w głąb ze standardowymi nawrotami. Metodami tymi są np. poszukiwania ze *sprawdzaniem kroku w przód* lub ze *sprawdzaniem dwóch kroków w przód*.

5. W językach *CLP* dokonano integracji wymienionych metod poszukiwań i technik zgodnościowych w bardzo efektywne *metody poszukiwania i propagacji*, udostępniane przez wygodne w użyciu solwery.

6. W programach prologowych poszukiwania rozpoczynały się automatycznie po aktywacji zapytania programu. W programach clp-owych poszukiwania są inicjowane specjalnym (najczęściej standardowym) predykatem, uziemiającym zmienne decyzyjne w określonej kolejności. Najprostszym standardowym predykatem tego typu jest `labeling/1`, którego właściwości przedstawia poniższa reguła:

```
labeling([H|T]):-
    indomain(H),
    labeling(T).
labeling([]).,
```

   gdzie standardowy predykat `indomain(Lista_zmiennych)` przyporządkowuje zmiennym z `Listy_zmiennych` wartości z ich dziedzin. Krótko mówiąc - uziemia zmienne (w kolejności ich występowania w liście, z lewa na prawo) na wartościach z ich dziedzin. Kolejność ta nie zawsze jest najbardziej efektywna; dlatego języki *CLP* (w tym również ECL^iPS^e) udostępnia szereg heurystyk poszukiwania, różnych od poszukiwań realizowanych za pomocą `labeling/1`, patrz 3.3.

7. Programowanie w ECL^iPS^e *Prolog*u nie wymaga dołączania do programu dodatkowych bibliotek. W przypadku programowania w ECL^iPS^eCLP, program musi rozpoczynać się deklaracją potrzebnych bibliotek. Najczęściej będziemy korzystać z następujących bibliotek:

   - Biblioteka `ic` (`interval constraint`) udostępnia hybrydowy solwer ograniczeń, rozwiązujący ograniczenia zarówno w dziedzinach liczb całkowitych, jak i rzeczywistych (stosując arytmetykę przedziałową[1]) oraz ograniczenia mieszane całkowito-rzeczywiste. Hybrydowość oznacza możliwość rozwiązywania problemów zawierających zarówno zmienne decyzyjne całkowite, jak i ciągłe. Uzyskano to dzięki zastosowaniu dla obydwu typów zmiennych tych samych algorytmów propagacji ograniczeń. Biblioteka ta jest biblioteka podstawową dla programów przedstawionych w rozdziale bieżącym i w rozdziałach 4, 5 i 6.

[1] Arytmetyka przedziałowa jest arytmetyką liczb rzeczywistych scharakteryzowanych przedziałami, których przetwarzanie generuje wyniki również w postaci przedziałów.

- Biblioteka `lib(branch_and_bound)` udostępnia silnie sparametryzowany solwer dla algorytmu `branch-and-bound`, stosowany w rozdziałach 5 i 6.
- Biblioteka `eplex` udostępnia solwery LP, MIP i QP, umożliwiające również tworzenie interfejsów do cudzych programów optymalizacji.
- Biblioteka `lib(ic_global)` udostępnia szereg globalnych predykatów dla list zmiennych całkowitych.
- Biblioteka `lib(ic_cumulative)` udostępnia predykat `cumulative/4`, bardzo użyteczny w zadaniach szeregowania i harmonogramowania, patrz rozdział 6.
- Biblioteki `lib(ic_edge_finder)` i `lib(ic_edge_finder3)` udostępniają mocniejsze wersje dla szeregu wariantów globalnego predykatu `cumulative/n` oraz `disjunctive/2`.
- Biblioteka `lib(ic_sets)` udostępnia solwer dla ograniczeń w dziedzinie zbiorów uporządkowanych i pozbawionych powtórzeń liczb całkowitych.
- Biblioteka `lib(ic_symbolic)` udostępnia solwer dla ograniczeń w dziedzinie uporządkowanych symboli.

Szczegółowe omówienie bibliotek można znaleźć w pliku ECL^iPS^e *Constraint Library Manual*, dostępnym w dokumentacji, patrz *ECLiPSe Documentation*, rysunek 5.

Wymienione pojęcia dobrze zilustrować na kilku przykładach. Pierwszym z nich będzie znany już problem `N` hetmanów.

3.2.2 Zasadnicze podobieństwa

Zarówno Prolog jak i języki CLP wnioskują za pomocą poszukiwań i propagacji ograniczeń. Wymienione pojęcia dobrze zilustrować na kilku przykładach. Pierwszym z nich będzie znany już problem `N` hetmanów.

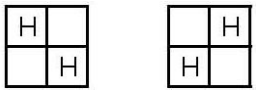
3.2.3 Problem N hetmanów w CLP

Dotychczas przedstawiono dwa podejścia do tego problemu:

1. Przegląd zupełny, dla którego generuje się kolejno wszystkie permutacje `[X1,X2,...,Xn] = [1,2,...,n]` i sprawdza, czy przedstawiają bezpieczne ustawienie hetmanów.

2. Poszukiwania w głąb ze standardowymi nawrotami, dla których do bezpiecznego ustawienia części hetmanów `[Xj,Xk,...]` dodaje się nowego hetmana i sprawdza, czy obecne ustawienie jest nadal bezpieczne; jeżeli nie, wykonuje się nawrót do takiego najbliższego poprzedniego ustawienia, dla którego istnieje inny wybór dostawianego hetmana. Standardowe nawroty obcinają duże fragmenty drzewa poszukiwań, zwiększając efektywność poszukiwań w porównaniu z przeglądem zupełnym.

Poszukiwania w głąb ze standardowymi nawrotami mają jednak dwie poważne wady: 1)poszukiwania nowych rozwiązań są inicjowane dopiero w wyniku naruszenia ograniczenia, oraz 2) poszukiwaniom może towarzyszyć *błądzenie*, tzn. wielokrotny powrót do złych rozwiązań cząstkowych, np. takich jak na rysunku 3.1.



Rysunek 3.1: Fragment ustawienia hetmanów powodujący *błądzenie*

Spróbujemy obecnie pokazać, jak wprowadzenie bardziej efektywnych metod propagacji może przyśpieszyć rozwiązanie problemu `N` hetmanów w porównaniu z poszukiwaniami w głąb ze standardowymi nawrotami.

3.2.4 Sprawdzanie kroku w przód

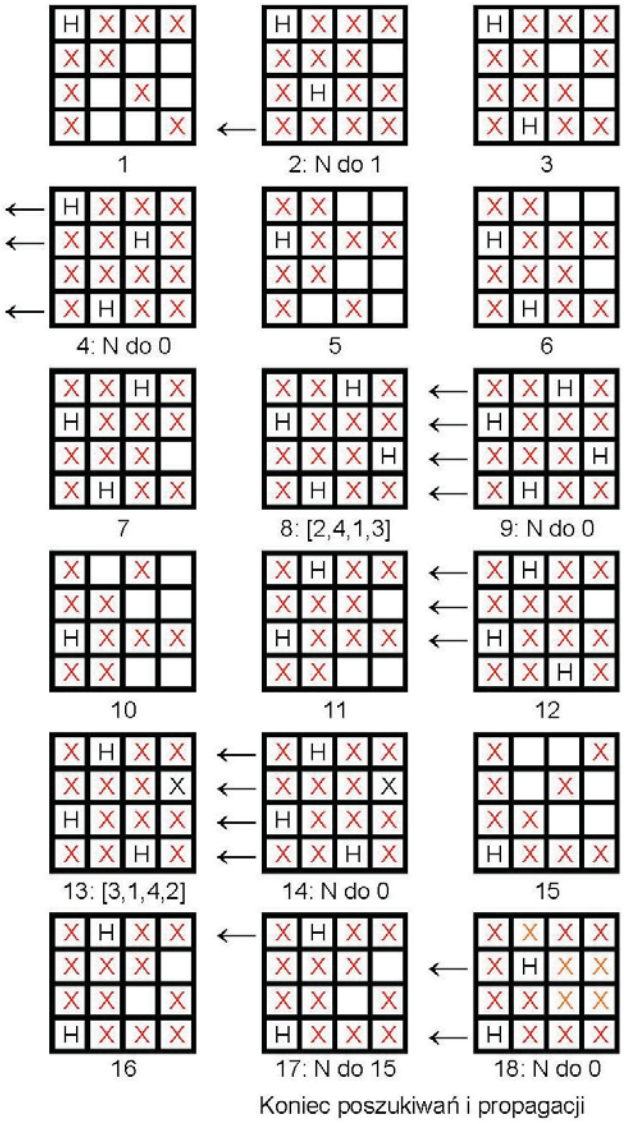
Lepszą techniką poszukiwań aniżeli standardowe nawroty są *poszukiwania ze sprawdzaniem kroku w przód*. Ściśle biorąc - jest to nie tylko *technika poszukiwań*, tzn. technika uziemiania, odziemiania i reuziemiania zmiennych w określo-

nej kolejności. Łączy ona poszukiwania z bardziej efektywnymi metodami propagacji ograniczeń aniżeli unifikacja, mianowicie z *technikami zgodnościowymi* (ang. *consistency techniques*). Istotą poszukiwania i propagacji ze sprawdzaniem kroku w przód jest wykonywanie nawrotu już wtedy, gdy w następnym kroku nieuchronne będzie naruszenie ograniczenia. Można to zilustrować graficznie na prostym przykładzie ustawiania 4 hetmanów. Zakłada się, że:

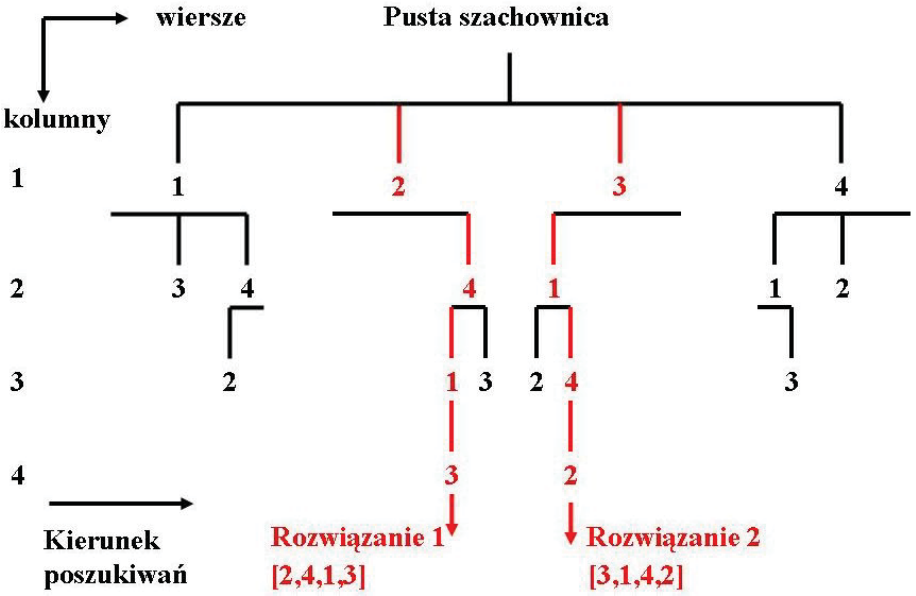
- każdy hetman i ma swoją dziedzinę: zbiór dopuszczalnych wartości zmiennej *Xi*, będącej numerem wiersza zajmowanego przez hetmana w kolumnie i-tej;
- na początku dziedziny są jednakowe i dane elementami listy `[1,2,3,4]`;
- do już bezpiecznie ustawionych hetmanów [x1,x2,..] dodaje się nowego hetmana na pozycji wynikającej z jego dziedziny i uaktualnia się dziedziny hetmanów jeszcze nie ustawionych. Uaktualnianie dziedzin jest szczególnym przypadkiem *propagacji ograniczeń*: ograniczenie wprowadzone przez ustawienie nowego hetmana jest propagowane na dziedziny hetmanów jeszcze nie ustawionych;
- jeżeli jedna z tych dziedzin jest pusta, wykonuje się *nawrót* do najbliższego poprzedniego ustawienia, dla którego istnieją niepuste dziedziny hetmanów jeszcze nie ustawionych.

Tak wykonane poszukiwania, propagacje i nawroty ilustruje rysunek 3.2, na którym czerwonym symbolem $\times$ oznaczono kratki "niedozwolone", tzn. wyeliminowane z dziedzin w wyniku zaistniałych już ustawień hetmanów. Rysunkowi 3.2 odpowiada drzewo poszukiwań z rysunku 3.3.

Jak widać, w przypadku sprawdzania kroku w przód, zdarzeniem wyzwalającym nawrót nie jest *naruszenie ograniczenia*, lecz *nieuchronność naruszenia ograniczenia* w następnym kroku, objawiająca się zaistnieniem pustej dziedziny w sytuacji, gdy jeszcze nie wszystkie zmienne zostały uziemione. *Sprawdzanie kroku w przód* generuje nowe ustawienie hetmanów przez dołączenie nowego hetmana do już istniejącego bezpiecznego ustawienia tylko wtedy, gdy nowy hetman ma niepustą dziedzinę. W przeciwnym przypadku następuje nawrót. Dzięki temu *sprawdzanie kroku w przód* obcina dodatkowe fragmenty drzewa poszukiwań w porównaniu z poszukiwaniem i propagacją metodą standardowych nawrotów, zwiększając efektywność poszukiwań.



Rysunek 3.2: Przebieg poszukiwań i propagacji ze *sprawdzaniem kroku w przód*



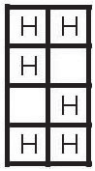
Rysunek 3.3: Drzewo poszukiwań dla poszukiwań i propagacji ze *sprawdzaniem kroku w przód*

3.2.5 Sprawdzanie dwóch kroków w przód

Sprawdzanie kroku w przód ma jeszcze wadę: nie dostrzega dalszych niż w następnym kroku konsekwencji pewnych sytuacji, usiłując lokować hetmanów w polach, w których zagrażać będą sobie w którymś z dalszych kroków (np. w drugim). Ilustruje to rysunek 3.4.

Aby temu zapobiec należy wykonać nawrót już wtedy, gdy w drugim kroku nieuchronne będzie naruszenie ograniczenia. Uzyskujemy to stosując metodę *sprawdzania dwóch kroków w przód*. Jest ona praktycznie zawsze łączona z metodą *sprawdzania kroku w przód*. Można to najlepiej zilustrować graficznie na prostym przykładzie ustawianie 4 hetmanów, pokazanym na rysunkach 3.5 i 3.6. Zwróćmy uwagę na to, że:

- *sprawdzanie kroku w przód* nie sprawdza niepustych dziedzin hetmanów nie ustawionych;



Rysunek 3.4: Fragment ustawienia hetmanów powodujący niepotrzebne poszukiwanie przy stosowaniu *sprawdzania kroku w przód*

- *sprawdzanie dwóch kroków w przód* sprawdza, czy niepuste dziedziny hetmanów nie ustawionych zawierają pozycje, które w drugim kroku doprowadza do porażki; jeżeli tak jest, to już wtedy wykonuje się nawrót do najbliższego poprzedniego ustawienia, dla którego istnieją niepuste dziedziny hetmanów jeszcze nie ustawionych.

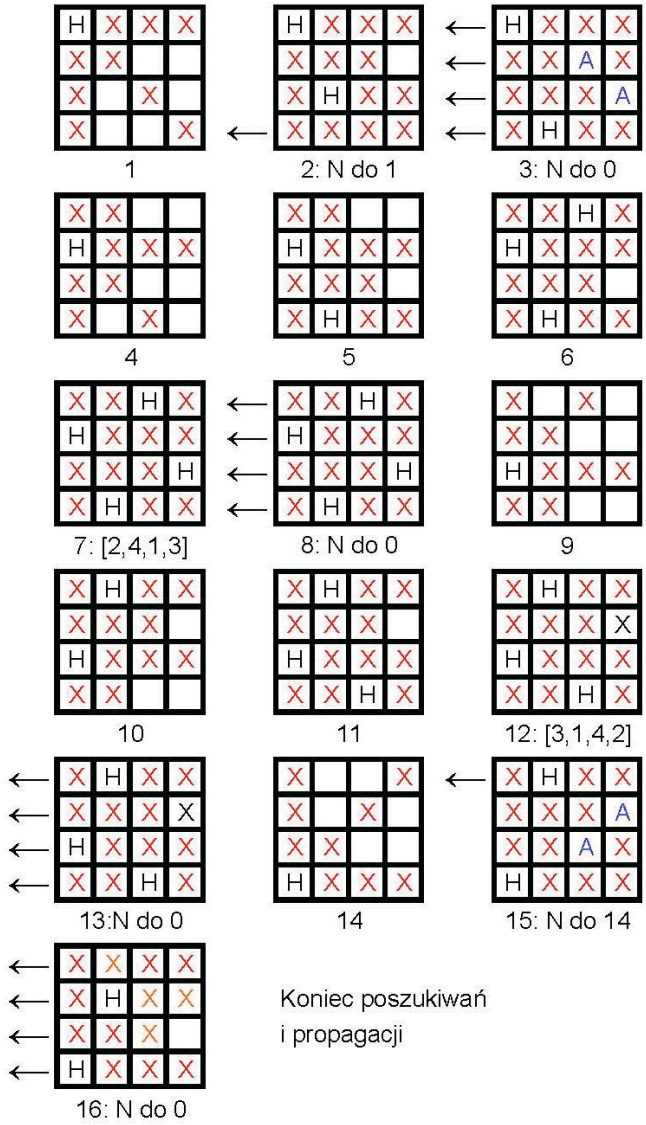
Czy można sprawdzać na więcej kroków w przód? Teoretycznie tak, ale praktycznie nie ma to większego sensu: czas potrzebny na sprawdzania staje się wtedy bardzo szybko większy od czasu potrzebnego dla sprawdzania jednego i dwóch kroków w przód.

3.3 Heurystyki poszukiwań

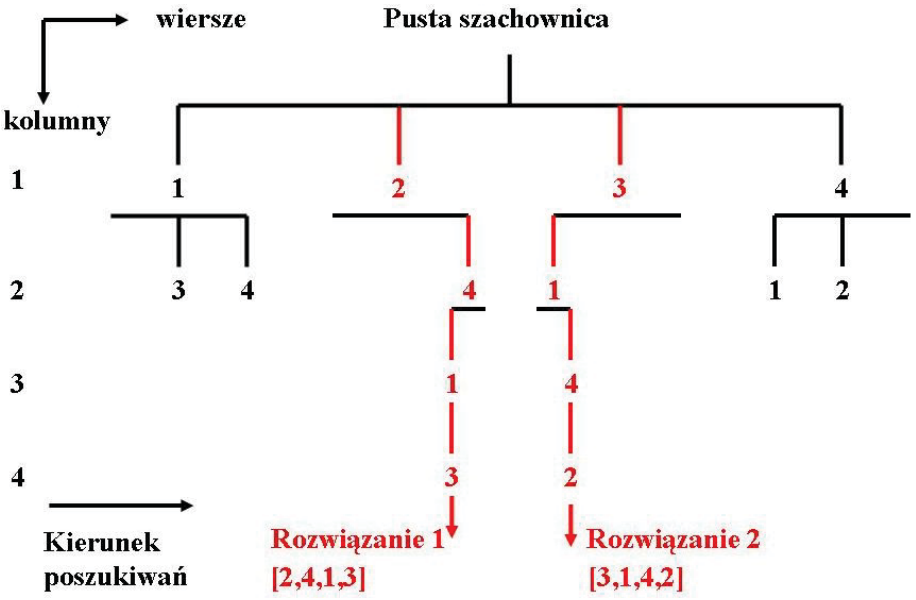
Sposoby wyboru kolejności zmiennych i wartości z dziedzin noszą w *CLP* nazwę *heurystyk poszukiwania. Heurystyką* nazywa się metodę rozwiązywania problemu nie gwarantującą sukcesu. Tym właśnie różni się heurystyka od algorytmu, który jest metodą rozwiązywania problemu gwarantującą sukces. Z przykładu ustawiania hetmanów wynika, że dla prowadzenia poszukiwań należy jeszcze wyjaśnić dwie sprawy:

1. Od jakiej zmiennej należy rozpocząć poszukiwania, jaką zmienną wybrać potem, jaką w następnej kolejności? Rozstrzygają to *heurystyki wyboru zmiennej.*

2. Którą wartość z dziedziny należy przyporządkować najpierw wybranej zmiennej, którą potem, którą w następnej kolejności? Rozstrzygają to *heurystyki wyboru wartości.*



Rysunek 3.5: Przebieg poszukiwań i propagacji metodą *sprawdzania dwóch kroków w przód*



Rysunek 3.6: Drzewo poszukiwań dla poszukiwań i propagacji metodą *sprawdzania dwóch kroków w przód*

W rozpatrywanych przykładach dokonywaliśmy wyboru zmiennych do uziemienia poczynając od głowy listy zmiennych, następnie wybieraliśmy głowę ogona listy zmiennych, itd. Można zauważyć, że nie był to najlepszy wybór. Gdybyśmy zaczynali od "środka" listy (w naszym przypadku wybierając najpierw drugą zmienną z listy), to jak widać na rysunkach 3.5 i 3.6, rozwiązanie zostałoby wyznaczone szybciej.
W rozpatrywanych przykładach przyporządkowaliśmy wybranej zmiennej najpierw pierwszą wartość z dziedziny, potem drugą itd. Można również zauważyć, że nie był to najlepszy wybór. Gdybyśmy zaczynali od "środka" dziedziny (w naszym przypadku ukonkretniając zmienną najpierw na wartości 2), to jak widać na rysunkach 3.5 i 3.6, rozwiązanie zostałoby wyznaczone szybciej.

W rozdziale 5.4.2 poznamy heurystyki poszukiwań stosowane dla bardziej niż `labeling/1` efektywnego i wszechstronnego predykatu poszukiwań, którym jest `search/6`.

Jednakże już w tym miejscu trzeba podkreślić, że praktycznie jedyną skuteczną metodą wyboru *najskuteczniejszych* (dla danego problemu) heurystyk poszukiwań jest niestety *przegląd zupełny*: należy doświadczalnie zbadać efektywność wszystkich heurystyk udostępnianych przez platformę ECL^iPS^e i zastosować najkorzystniejszą z nich.

3.4 Techniki zgodnościowe

Zadaniem technik zgodnościowych jest usunięcie niezgodnych wartości z dziedzin zmiennych. W językach *CLP* problemy spełnienia ograniczeń kombinatorycznych są definiowane dla dziedzin zmiennych całkowitych. Dla dziedzin tych istnieje szeroki wachlarz *algorytmów zgodnościowych*, zmierzających do redukcji dziedzin zmiennych występujących w ograniczeniach. Techniki zgodnościowe przyjęły nazwę od grafów, za pomocą których można przedstawić ograniczenia: wierzchołkami tych grafów są zmienne, a krawędziami są ograniczenia wiążące te zmienne. Obszerne omówienie algorytmów zgodnościowych można znaleźć w literaturze podręcznikowej, patrz np. [Tsang-95], [Dechter-03] i [Rossi-06]. W zależności od liczby zmiennych związanych ograniczeniami mówi się o:
- zgodności węzłowej (*NC - node consistency*) w przypadku ograniczeń unarnych;
- zgodności łukowej (*AC - arc consistency*) w przypadku ograniczeń binarnych;
- zgodności ścieżkowej (*PC - path consistency*) w przypadku ograniczeń ternarnych i ograniczeń o wyższej *arności*.

Praktycznie algorytmy wymuszające zgodność ścieżkową nie są stosowane, gdyż zgodność tę można wyrazić w sposób prostszy. Np. ścieżkowe ograniczenie $X = Y + Z$ przy dziedzinach odpowiednio D_X, D_Y i D_Z można wyrazić w postaci:

$$X >= min(D_Y) + min(D_Z)$$
$$X <= max(D_Y) + max(D_Z)$$
$$Y >= min(D_X) - max(D_Z)$$
$$Y <= max(D_X) - min(D_Z)$$
$$Z >= min(D_X) - max(D_Y)$$
$$Z <= max(D_X) - max(D_Y)$$

3.5 Propagacja ograniczeń z porażką

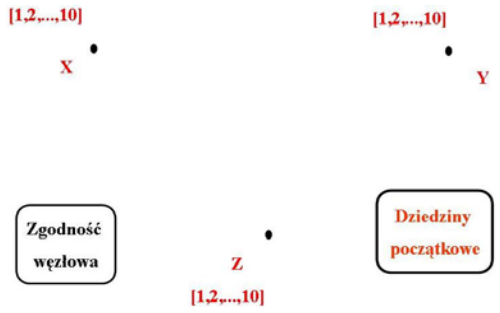
Istotę technik zgodnościowych ilustruje program `3_1_dziedzina_0.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/       [X,Y,Z]::1..10,
/*4*/       get_domain(X,PX),
/*5*/       get_domain(Y,PY),
/*6*/       get_domain(Z,PZ),
/*7*/       write("X = "), write(PX),nl,
/*8*/       write("Y = "), write(PY),nl,
/*9*/       write("Z = "), write(PZ),nl,nl,

/*10*/  write("Propagacja ograniczenia Y < Z daje:"),nl,
/*11*/      Y#<Z,
/*12*/      get_domain(X,CX),
/*13*/      get_domain(Y,CY),
/*14*/      get_domain(Z,CZ),
/*15*/      write("X = "), write(CX),nl,
/*16*/      write("Y = "), write(CY),nl,
/*17*/      write("Z = "), write(CZ),nl,nl,

/*18*/  write("Propagacja ograniczenia X = Y + Z daje:"),nl,
/*19*/      X#=Y+Z,
/*20*/      get_domain(X,DX),
/*21*/      get_domain(Y,DY),
/*22*/      get_domain(Z,DZ),
/*23*/      write("X = "), write(DX),nl,
/*24*/      write("Y = "), write(DY),nl,
/*25*/      write("Z = "), write(DZ),nl,nl,

/*26*/  write("Propagacja ograniczenia X = Z + 3 daje:"),nl,
/*27*/      X#=Z+3,
/*28*/      get_domain(X,TX),
/*29*/      get_domain(Y,TY),
/*30*/      get_domain(Z,TZ),
/*31*/      write("X = "), write(TX),nl,
/*32*/      write("Y = "), write(TY),nl,
/*33*/      write("Z = "), write(TZ),nl,nl,

/*34*/  write("Propagacja ograniczenia X > 2+Z daje:"),nl,
/*35*/      X#>2+Z,
/*36*/      get_domain(X,TTX),
/*37*/      get_domain(Y,TTY),
/*38*/      get_domain(Z,TTZ),
/*39*/      write("X = "), write(TTX),nl,
/*40*/      write("Y = "), write(TTY),nl,
```

```
/*41*/      write("Z = "), write(TTZ),nl,nl,

/*42*/  write("Propagacja ograniczenia Y = 2*Z daje porażkę:"),nl,
/*43*/      Y#=2*Z,
/*44*/      get_domain(X,SX),
/*45*/      get_domain(Y,SY),
/*46*/      get_domain(Z,SZ),
/*47*/      write("X = "), write(SX),nl,
/*48*/      write("Y = "), write(SY),nl,
/*49*/      write("Z = "), write(SZ).
```

```
Rozwiązanie ma postać następującą:
     X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
     Y = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
     Z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Wyznaczone dziedziny początkowe przedstawiono na rysunku 3.7

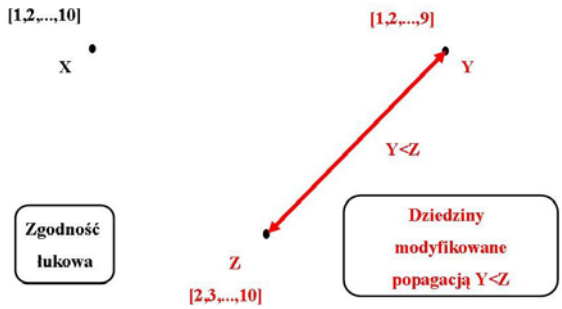


Rysunek 3.7: Dziedziny początkowe zmiennych X, Y i Z

```
Propagacja ograniczenia Y < Z daje:
     X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
     Y = [1, 2, 3, 4, 5, 6, 7, 8, 9]
     Z = [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Wyniki tej propagacji przedstawiono na rysunku 3.8.

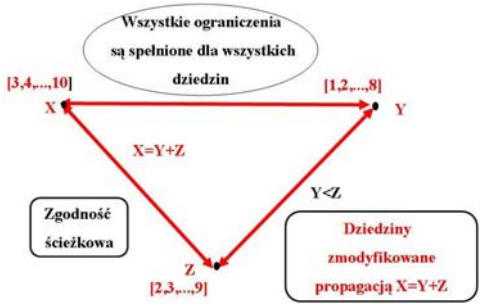
```
Propagacja ograniczenia X = Y + Z daje:
```



Rysunek 3.8: Wyniki udanej propagacji $Y < Z$

```
X = [3, 4, 5, 6, 7, 8, 9, 10]
Y = [1, 2, 3, 4, 5, 6, 7, 8]
Z = [2, 3, 4, 5, 6, 7, 8, 9]
```

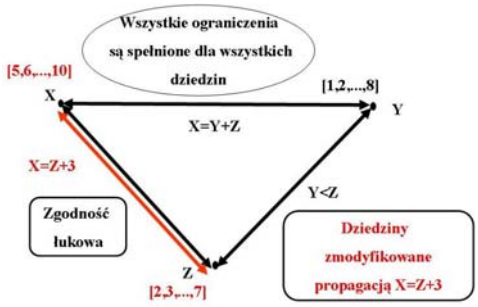
Wyniki tej propagacji przedstawiono na rysunku 3.9.



Rysunek 3.9: Wyniki udanej propagacji $X = Y + Z$

```
Propagacja ograniczenia X = Z + 3 daje:
      X = [5, 6, 7, 8, 9, 10]
      Y = [1, 2, 3, 4, 5, 6]
      Z = [2, 3, 4, 5, 6, 7]
```

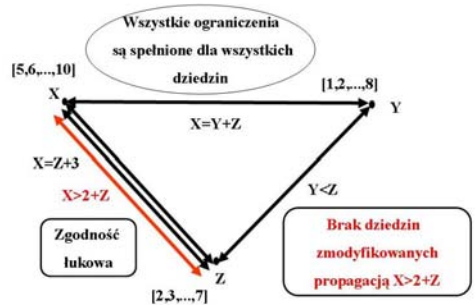
Wyniki tej propagacji przedstawiono na rysunku 3.10.



Rysunek 3.10: Wyniki udanej propagacji $X = Z + 3$

```
Propagacja ograniczenia X > 2+Z daje:
      X = [5, 6, 7, 8, 9, 10]
      Y = [1, 2, 3, 4, 5, 6]
      Z = [2, 3, 4, 5, 6, 7]
```

Wyniki tej propagacji przedstawiono na rysunku 3.11.

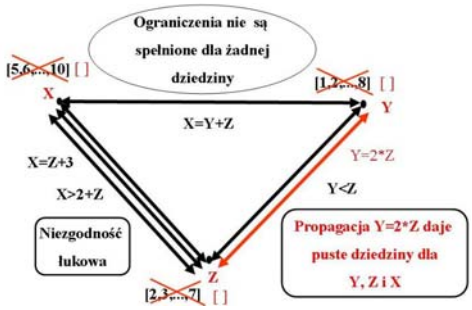


Rysunek 3.11: Wyniki udanej propagacji $X > 2 + Z$

```
Propagacja ograniczenia Y = 2*Z daje porażkę:
```

```
No
```

Wyniki tej propagacji przedstawiono na rysunku 3.12.



Rysunek 3.12: Wyniki nieudanej propagacji $Y = 2 * Z$

Do linii `41` propagacja ograniczeń zmniejsza dziedziny zmiennych. W linii `42` wprowadzono ograniczenie sprzeczne z ograniczeniem z linii `10`, co sprawia, że dziedziny `Y` i `Z` stają się dziedzinami pustymi i program kończy się porażką: układ nierówności okazał się dla dziedzin początkowych układem sprzecznym.

3.6 Propagacja ograniczeń czasami wystarcza

Aczkolwiek propagacja ograniczeń nie jest *kompletną* metodą wnioskowania, to w szczególnych przypadkach doprowadzić do wyznaczenia istniejącego jednoznacznego rozwiązania. Ilustrują to poniższe przykłady.

3.6.1 Prosty przykład

Dany jest program `3_2_dziedzina_1.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/       [X,Y,Z]::1..10,
/*4*/       get_domain(X,PX),
/*5*/       get_domain(Y,PY),
/*6*/       get_domain(Z,PZ),
```

```
/*7*/       write("X = "), write(PX),nl,
/*8*/       write("Y = "), write(PY),nl,
/*9*/       write("Z = "), write(PZ),nl,nl,

/*10/       write("Propagacja ograniczenia X = Y+3 daje:"),nl,
/*10*/      X#=Y+3,
/*11*/      get_domain(X,CX),
/*12*/      get_domain(Y,CY),
/*13*/      get_domain(Z,CZ),
/*14*/      write("X = "), write(CX),nl,
/*15*/      write("Y = "), write(CY),nl,
/*16*/      write("Z = "), write(CZ),nl,nl,

/*17/       write("Propagacja ograniczenia Y < 3 daje:"),nl,
/*18*/      Y#<3,
/*19*/      get_domain(X,DX),
/*20*/      get_domain(Y,DY),
/*21*/      get_domain(Z,DZ),
/*22*/      write("X = "), write(DX),nl,
/*23*/      write("Y = "), write(DY),nl,
/*24*/      write("Z = "), write(DZ),nl,nl,

/*25/       write("Propagacja ograniczenia X > 2+Z daje:"),nl,
/*26*/      X#>2+Z,
/*27*/      get_domain(X,TX),
/*28*/      get_domain(Y,TY),
/*29*/      get_domain(Z,TZ),
/*30*/      write("X = "), write(TX),nl,
/*31*/      write("Y = "), write(TY),nl,
/*32*/      write("Z = "), write(TZ),nl,nl,

/*33/       write("Propagacja ograniczenia Y = 2*Z daje:"),nl,
/*33*/      Y#=2*Z,
/*34*/      get_domain(X,SX),
/*35*/      get_domain(Y,SY),
/*36*/      get_domain(Z,SZ),
/*37*/      write("X = "), write(SX),nl,
/*38*/      write("Y = "), write(SY),nl,
/*39*/      write("Z = "), write(SZ),nl.
```

Rozwiązanie ma postać:

```
X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Y = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Propagacja ograniczenia X = Y+3 daje:
X = [4, 5, 6, 7, 8, 9, 10]
Y = [1, 2, 3, 4, 5, 6, 7]
Z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Propagacja ograniczenia Y < 3 daje:
X = [4, 5]
Y = [1, 2]
Z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Propagacja ograniczenia X > 2+Z daje:
X = [4, 5] Y = [1, 2]
Z = [1, 2]

Propagacja ograniczenia Y = 2*Z daje:
X = [5]
Y = [2]
Z = [1]
```

Rozwiązanie otrzymano korzystając tylko z propagacji ograniczeń.

3.6.2 Kto był z kim?

Dziedziny liczb całkowitych mają nieoczekiwane zastosowania dla rozwiązywania różnorodnych łamigłówek. Rozpatrzmy następujący przykład[2]:

Kto był z kim wczoraj wieczorem, jeżeli:
1. Andrzej poszedł na koncert.
2. Bronek spędził wieczór z Olą.
3. Czarek nie widział Ewy.
4. Paulina była w kinie.
5. Ewa była w teatrze.
6. Jedna para była na wystawie.

Do towarzystwa należy jeszcze Darek i Sabina.
Każdy chłopak miał wspólny program z którąś z dziewcząt.

Rozwiązanie wyznacza program `3_3_kto_z_kim.ecl`:

[2]Temat przykładu pochodzi z książki [Bizam-75].

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/       [Andrzej,Bronek,Czarek,Darek] :: [1..4],
/*4*/       [Ola, Ewa,Paulina,Sabina] :: [1..4],
          % koncert=1, kino=2, teatr=3, wystawa = 4
          % co znaczy: jeżeli np. Bronek=Ola=4, to
          % Bronek z Olą poszli na wystawę

/*5*/       Andrzej#=1,     % Andrzej poszedł na koncert
/*6*/       Bronek#=Ola,    % Bronek spędził wieczór z Olą
/*7*/       Czarek#\=Ewa,   % Czarek nie widział Ewy
/*8*/       Paulina#=2,     % Paulina była w kinie
/*9*/       Ewa#=3,         % Ewa była w teatrze

/*10*/      Andrzej#\=Bronek,
/*11*/      Andrzej#\=Czarek,
/*12*/      Andrzej#\=Darek,
/*13*/      Bronek#\=Czarek,
/*14*/      Bronek#\=Darek,
/*15*/      Czarek#\=Darek,

/*16*/      Ola#\=Ewa,
/*17*/      Ola#\=Paulina,
/*18*/      Ola#\=Sabina,
/*19*/      Ewa#\=Paulina,
/*20*/      Ewa#\=Sabina,
/*21*/      Paulina#\=Sabina,

/*22*/      write(Andrzej),write(" "),write(Bronek),write(" "),
            write(Czarek),write(" "),write(Darek),nl,
/*23*/      write(Ola),write(" "),write(Ewa),write(" "),
            write(Paulina),write(" "),write(Sabina).
```

W programie brak predykatu `labeling(_)` inicjującego poszukiwania. Wprowadzenie tego predykatu sprawi, że rozwiązanie zostanie uzyskane w krótszym czasie.

Rozwiązanie ma postać słabo czytelnego komunikatu:

```
1 4 2 3
4 3 2 1
```

oznaczającego, że:

```
Andrzej   (pozycja pierwsza listy chłopców)
i Sabina  (pozycja czwarta listy dziewcząt)
byli na koncercie (1).
Bronek    (pozycja druga listy chłopców)
i Ola     (pozycja pierwsza listy dziewcząt)
byli na wystawie (4).
Czarek    (pozycja trzecia listy chłopców)
i Paulina (pozycja trzecia listy dziewcząt)
byli w kinie (2).
Darek     (pozycja czwarta listy chłopców)
i Ewa     (pozycja druga listy dziewcząt)
byli w teatrze (3).
```

Czytelność komunikatu zostanie poprawiona w rozdziale 4.4.3.

W tym miejscu należy podkreślić, że to co zrobiono w liniach `/*3*/` i `/*4*/` programu oraz w poniższych komentarzach (zdefiniowanie zmiennych, ich dziedzin, ustalenia znaczenia zmiennych i ustalenie znaczenia liczb z dziedzin) jest czymś, czego nie trzeba było robić programując w Prologu. Jest to jednak niezbędne (i nie ukrywajmy tego - często trudne) do zrobienia w programach clp-owych.

3.6.3 Studenci i języki

Często dla stosunkowo złożonych problemów może wystarczyć propagacja. Ilustruje to następujący przykład[3]:

Na Mazurach spotkało się pięciu studentów: Polak, Węgier, Fin, Szwed i Niemiec. Napisać program wyznaczający języki, którymi każdy z nich włada, jeżeli:

1. Każdy z nich zna jeden lub więcej języków obcych, ale tylko takie które są ojczystymi językami któregoś z nich.
2. Nie ma takiego języka, którym władałaby cała piątka.
3. Każdy z nich może się porozumieć z każdym innym z tej piątki w jakimś języku.
4. Wśród tych wspólnych języków występują języki ojczyste wszystkich pięciu studentów.
5. Studenci znają średnio po dwa języki obce.
6. Polak i Węgier znają po trzy języki obce.

[3]Temat przykładu pochodzi z książki [Bizam-75].

7. Kiedy Szwed poszedł się kąpać, czterej pozostali od razu znaleźli "wspólny język".
8. Podobna sytuacja zaistniała, kiedy Szwed wrócił, ale Fin poszedł wiosłować.
9. Aby można było rozmawiać po szwedzku, dwaj studenci musieliby się oddalić.
10. Po polsku i po fińsku mówią tylko dwaj studenci.
11. Polak i Fin mogą się porozumieć w dwóch językach, język niemiecki jednak nie należy do tych języków.
12. Węgier i Szwed znają tylko jeden wspólny język.

Łamigłówkę można rozwiązać programem (`3_4_studenci_jezyku.ecl`):

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/      Studenci=["Polak","Wegier","Fin","Szwed","Niemiec"],
/*4*/      Jezyki=["polski","wegierski","finski","szwedzki",
                 "niemiecki"],

/*5*/      Polak=[PP,PW,PF,PS,PN], % PS=1 - Polak zna szwedzki,
                                     % PF=0 - Polak nie zna fińskiego
/*6*/      Wegier=[WP,WW,WF,WS,WN],
/*7*/      Fin=[FP,FW,FF,FS,FN],
/*8*/      Szwed=[SP,SW,SF,SS,SN],
/*9*/      Niemiec=[NP,NW,NF,NS,NN],
/*10*/     L=[Polak,Wegier,Fin,Szwed,Niemiec],
/*11*/     Polak::0..1,
/*12*/     Wegier::0..1,
/*13*/     Fin::0..1,
/*14*/     Szwed::0..1,
/*15*/     Niemiec::0..1,

              %ograniczenie0
              %każdy z nich zna swój język ojczysty:
/*16*/     PP#=1,
/*17*/     WW#=1,
/*18*/     FF#=1,
/*19*/     SS#=1,
/*20*/     NN#=1,

              %ograniczenie1
              %Każdy z nich zna jeden lub więcej języków
              %obcych, ale tylko takie które"),
              %są ojczystymi językami któregoś z nich:
/*21*/     PW+PF+PS+PN#>0,
/*22*/     WP+WF+WS+WN#>0,
/*23*/     FP+FW+FS+FN#>0,
/*24*/     SP+SW+SF+SN#>0,
```

```
/*25*/     NP+NW+NF+NS#>0,

               %ograniczenie2
               %Nie ma takiego języka, którym władałaby cała piątka:
/*26*/     WP+FP+SP+NP#<4,
/*27*/     PW+FW+SW+NW#<4,
/*28*/     PF+WF+SF+NF#<4,
/*29*/     PS+WS+FS+NS#<4,
/*30*/     PN+WN+FN+SN#<4,

               %ograniczenie3
               %Każdy z nich może się porozumieć z każdym
               %innym z tej piątki w jakimś języku:
/*31*/     ograniczenie3(Polak,Wegier),
/*32*/     ograniczenie3(Polak,Fin),
/*33*/     ograniczenie3(Polak,Szwed),
/*34*/     ograniczenie3(Polak,Niemiec),
/*35*/     ograniczenie3(Wegier,Fin),
/*36*/     ograniczenie3(Wegier,Szwed),
/*37*/     ograniczenie3(Wegier,Niemiec),
/*38*/     ograniczenie3(Fin,Szwed),
/*39*/     ograniczenie3(Fin,Niemiec),
/*40*/     ograniczenie3(Szwed,Niemiec),

               %ograniczenie5
               %Studenci znają średnio po dwa języki obce:
/*41*/     PW+PF+PS+PN+WP+WF+WS+WN+FP+FW+FS+FN+
/*42*/     SP+SW+SF+SN+ NP+NW+NF+NS#=10,

               %ograniczenie6
               %Polak i Węgier znają po trzy języki obce:
/*43*/     PW+PF+PS+PN#=3,
/*44*/     WP+WF+WS+WN#=3,

               %ograniczenie7
               %Kiedy Szwed poszedł się kąpać, czterej
               %pozostali od razu znaleźli "wspólny język":
/*45*/     ograniczenie7(WP,FP,NP,PW,FW,NW,PF,WF,NF,PN,WN,FN),

               %ograniczenie8
               %Podobna sytuacja zaistniała, kiedy
               %Szwed wrócił,ale Fin poszedł wiosłować:
/*46*/     ograniczenie8(WP,SP,NP,PW,SW,NW,PS,WS,NS,PN,WN,SN),

               %ograniczenie9
               %Aby można było rozmawiać po szwedzku,
               %dwaj studenci musieliby się oddalić:
/*46*/     PS+WS+FS+NS#=2,
```

```
                %ograniczenie4
                %Wśród tych wspólnych języków występują
                %języki ojczyste wszystkich pięciu studentów:
/*47*/      getval(p,1),
/*48*/      getval(w,1),
/*49*/      getval(f,1),
/*50*/      getval(s,1),
/*51*/      getval(n,1),

                %ograniczenie10
                %Po polsku i po fińsku mówią tylko dwaj studenci:
/*52*/      WP+FP+SP+NP#=1,
/*53*/      PF+WF+SF+NF#=1,

                %ograniczenie11
                %Polak i Fin mogą się porozumieć w dwóch językach,
                %język niemiecki jednak nie należy do tych języków:
/*53*/      ograniczenie11(PW,FW,FP,PF,PS,FS,PN,FN),

                %ograniczenie12
                %Węgier i Szwed znają tylko jeden wspólny język:
/*54*/      ograniczenie12(Wegier,Szwed),

/*55*/      wypisz(Studenci,L,Jezyki).

/*56*/    ograniczenie3([A1,A2,A3,A4,A5],[B1,B2,B3,B4,B5]):-
/*57*/      2#=A1+B1,
/*58*/      setval(p,1);     % uwaga: tu jest dyzjunkcja ("lub")
/*59*/      2#=A2+B2,
/*60*/      setval(w,1);
/*61*/      2#=A3+B3,
/*62*/      setval(f,1);
/*63*/      2#=A4+B4,
/*64*/      setval(s,1);
/*65*/      2#=A5+B5,
/*66*/      setval(n,1).

/*67*/    ograniczenie7(WP,FP,NP,PW,FW,NW,PF,WF,NF,PN,WN,FN):-
/*68*/      WP#=1,FP#=1,NP#=1;      % uwaga - operator "lub"
/*69*/      PW#=1,FW#=1,NW#=1;
/*70*/      PF#=1,WF#=1,NF#=1;
/*71*/      PN#=1,WN#=1,FN#=1.

/*72*/    ograniczenie8(WP,SP,NP,PW,SW,NW,PS,WS,NS,PN,WN,SN):-
/*73*/      WP#=1,SP#=1,NP#=1;
/*74*/      PW#=1,SW#=1,NW#=1;
/*75*/      PS#=1,WS#=1,NS#=1;
/*76*/      PN#=1,WN#=1,SN#=1.
```

```
/*77*/   ograniczenie11(PW,FW,FP,PF,PS,FS,PN,FN):-
/*78*/     ograniczenie11b(PN,FN),
/*79*/     ograniczenie11a(PW,FW,FP,PF,PS,FS).

/*80*/   ograniczenie11a(PW,FW,FP,PF,PS,FS):-
/*81*/     PW#=1,FW#=1,FP#=1;
/*82*/     PF#=1,FP#=1;
/*83*/     PS#=1,FS#=1,FP#=1;
/*84*/     PW#=1,PF#=1,FW#=1;
/*85*/     PW#=1,PS#=1,FW#=1,FS#=1;
/*86*/     PF#=1,PS#=1,FS#=1.

/*87*/   ograniczenie11b(PN,FN):-
/*88*/     PN#=0;FN#=0.

/*89*/   ograniczenie12([G1|_],[G2|_]):-
/*90*/     G1#=1,
/*91*/     G2#=1,
/*92*/     !.

/*93*/   ograniczenie12([G1|O1],[G2|O2]):-
/*94*/     ograniczenie12a(G1,G2),
/*95*/     ograniczenie12(O1,O2).

/*96*/   ograniczenie12a(G1,G2):-
/*97*/     G1#=0;G2#=0.        % uwaga na "lub".

/*98*/   wypisz([G1|O1],[G2|O2],L3):-
/*99*/     write(G1),writeln(" zna:"),
/*100*/    wypisz1(G2,L3),
/*101*/    wypisz(O1,O2,L3).
/*102*/  wypisz([],[],_).

/*103*/  wypisz1([1|O1],[G2|O2]):-
/*104*/    write("   "),writeln(G2),
/*105*/    wypisz1(O1,O2).
/*106*/  wypisz1([],[]).

/*107*/  wypisz1([0|O1],[_|O2]):-
/*108*/    wypisz1(O1,O2).
```

Rozwiązanie generowane przez program ma postać:

```
Polak zna:
 polski
 wegierski
 szwedzki
```

```
  niemiecki
Wegier zna:
  polski
  wegierski
  finski
  niemiecki
Fin zna:
  wegierski
  finski
  szwedzki
Szwed zna:
  szwedzki
  niemiecki
Niemiec zna:
  wegierski
  niemiecki
```

Również w tym przykładzie, pomimo jego złożoności, można z poszukiwań zrezygnować. Sama propagacja ograniczeń wystarcza dla wyznaczenia rozwiązania. Ograniczenie 4 następuje po ograniczeniu 9, co wynika stąd, że kolejność ograniczeń nie jest obojętna dla efektywności programu.

3.6.4 Tajni Współpracownicy i Opozycyjni Etosowcy

ECL^iPS^e ma również bibliotekę ograniczeń symbolicznych (`ic_symbolic`), wspierającą ograniczenia dla zmiennych definiowanych za pomocą nazw. Wymaga ona prefiksowania symboli operacji na nazwach symbolem `&`. Zastosowania tej biblioteki ilustruje następujący przykład:

Członkami ekskluzywnego klubu "Black and White" mogą być wyłącznie byli *Tajni Współpracownicy* oraz byli *Opozycyjni Etosowcy*. Taki wybór profilu klubu okazał się strzałem w dziesiątkę. Sprzyja on ożywionym i kontradyktoryjnym dyskusjom klubowym ściągającym liczną publiczność oraz wpływa korzystnie na konsumpcję wszystkich tych produktów, bez których nie sposób zrozumieć całej złożoności uwarunkowań, w których członkom klubu przyszło kiedyś żyć i pracować. Atrakcyjność dyskusji jest dodatkowo wzmacniana powszechną wiedzą o tym, że *Opozycyjni Etosowcy* zawsze mówią prawdę, zaś *Tajni Współpracownicy* na przemian kłamią i mówią prawdę. Poczytny tabloid

głównego nurtu "Der Rynsztok" wysłał ostatnio do klubu swego dziennikarza w nadziei na odpowiednio skandalizujący reportaż. Niestety, dziennikarz na samym początku napotkał na zasadniczą trudność: w klubie owego dnia byli tylko trzej członkowie, z których dwaj, `Członek_1` i `Członek_2`, zaciekle się kłócili, co wskazywało na ich przynależność do różnych grup klubowiczów. Dziennikarz, nie chcąc przeszkadzać adwersarzom, zapytał `Członka_3`, nie biorącego udziału w dyskusji, czy jest byłym *Opozycyjnym Etosowcem*, czy też byłym *Tajnym Współpracownikiem*. Niestety, `Członek_3` był po tak intensywnych próbach zrozumienia złożoności wzmiankowanych uwarunkowań, że tylko burknął coś niezrozumiałego. Dziennikarz zwrócił się więc do pozostałych członków z prośbą o wyjaśnienie, co też takiego powiedział `Członek_3`. `Członek_1` wyjaśnił mu: "`Członek_3` rzekł, że jest byłym opozycyjnym etosowcem". `Członek_2` natomiast najpierw powiedział: "`Członek_3` jest tajnym współpracownikiem", a potem dodał: "`Członek_3` kłamał".
Czy dziennikarz ma wystarczająco danych, by stwierdzić kto jest kim?

Aby lepiej zrozumieć problem, popatrzmy na jego przestrzeń stanu zapisaną w postaci tabeli prawdy z rysunku 3.13. Tabelę tę opisują trzy zmienne boolowskie (`C1`, `C21` i `C22`), odpowiadające wartościom logicznym wypowiedzi `Członka_1`, pierwszej wypowiedzi `Członka_2` i drugiej wypowiedzi `Członka_2`. Przyjęto, że wartość `0` oznacza, że odpowiednia wypowiedź jest kłamstwem, a wartość `1` - prawdą. Rozpatrywana przestrzeń stanu ma osiem stanów (C1, C21, C22), wynikających z współrzędnych wierszy i kolumn tablicy.

Rysunek 3.13: Tablica prawdy przestrzeni stanu dla przykładu TW-OE

Liczby wewnątrz tabeli są wartością logiczną koniunkcji wszystkich ograniczeń przykładu: `0` oznacza niespełnienie ograniczeń, `1` oznacza spełnienie ograniczeń. I tak:

- pierwsza kolumna musi zawierać zera, gdyż żaden członek klubu nie wypowiada kolejno dwóch kłamstw;
- druga kolumna jest "wewnętrznie sprzeczna": jeżeli `Członek_3` kłamał, to pierwsza odpowiedź `Członka_2` nie może być kłamstwem;
- to samo odnosi się do czwartej kolumny: jeżeli `Członek_3` nie kłamał, to oczywiście pierwsza odpowiedź `Członka_2` nie może być prawdą;
- rozpatrzmy dolny kwadrat kolumny trzeciej: jeżeli wypowiedź `Członka_1` jest prawdą, to obydwie wypowiedzi `Członka_2` nie mogą być prawdą;
- pozostaje więc górny kwadrat kolumny trzeciej, który odpowiada spełnionej koniunkcji warunków czyli stanowi dopuszczalnemu: jeżeli odpowiedź `Członka_1` jest kłamstwem, wówczas obydwie odpowiedzi `Członka_2` są prawdą;
- stąd wynika, że `Członek_1` i `Członek_3` są byłymi *Tajnymi Współpracownikami*, natomiast `Członek_2` jest byłym *Opozycyjnym Etosowcem*, *Q.E.D.*

Po upewnieniu się co do istnienia rozsądnego i jednoznacznego rozwiązania, przejdźmy do programu `3_5_black_and_white.ecl`[4]:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(ic_symbolic).
/*3*/   :- local domain(czlonek(opozycyjny_etosowiec,tajny_wspolpracownik)).

/*4*/   top :-
/*5*/       wyznacz(_).

/*6*/   wyznacz([Czlonek_1,Czlonek_2,Czlonek_3]):-
     % Deklaracja dziedziny zmiennej symbolicznej:
/*7*/       [Czlonek_1,Czlonek_2,Czlonek_3] &:: czlonek,
     % Deklaracja dziedziny zmiennej dwuwartościowej:
/*8*/       [Czlonek_3_Byc_Moze_Rzekl,Czlonek_3_Rzekl,
                Czlonek_1_Rzekl,Czlonek_2_Rzekl_Najpierw,
                    Czlonek_2_Rzekl_Potem] :: 0..1,
```

[4]Program ten bazuje na programie "Liars" opracowanym przez J. Schimpf'a, zob. [Schimpf-10a].

```
/*9*/       Czlonek_1 &\= Czlonek_2,

     % Co Czlonek_3 być może powiedział:
/*10*/      Czlonek_3_Byc_Moze_Rzekl #= (Czlonek_3 &= opozycyjny_etosowiec),
/*11*/      pojedyncze_stwierdzenie(Czlonek_3,
                Czlonek_3_Byc_Moze_Rzekl),
     % Co Czlonek_1 powiedział:
/*12*/      Czlonek_1_Rzekl #= (Czlonek_3_Rzekl #=
                Czlonek_3_Byc_Moze_Rzekl),
/*13*/      pojedyncze_stwierdzenie(Czlonek_1, Czlonek_1_Rzekl),

     % Co Czlonek_2 rzekł najpierw:
/*14*/      Czlonek_2_Rzekl_Najpierw #= (Czlonek_3 &=
                tajny_wspolpracownik),
/*15*/      pojedyncze_stwierdzenie(Czlonek_2,
                Czlonek_2_Rzekl_Najpierw),

     % Co Czlonek_2 rzekł potem:
/*16*/      Czlonek_2_Rzekl_Potem #=(Czlonek_3_Rzekl #= 0),
/*17*/      pojedyncze_stwierdzenie(Czlonek_2,
                Czlonek_2_Rzekl_Potem),
/*18*/      kolejne_stwierdzenia(Czlonek_2,
                Czlonek_2_Rzekl_Najpierw,
                    Czlonek_2_Rzekl_Potem),

/*19*/      ic_symbolic:indomain(Czlonek_1),
/*20*/      ic_symbolic:indomain(Czlonek_2),
/*21*/      ic_symbolic:indomain(Czlonek_3),
/*22*/      writeln("Czlonek_1":Czlonek_1),
/*23*/      writeln("Czlonek_2":Czlonek_2),
/*24*/      writeln("Czlonek_3":Czlonek_3),
/*25*/      writeln("Czlonek_2_Rzekl_Najpierw":Czlonek_2_Rzekl_Najpierw),
/*26*/      writeln("Czlonek_2_Rzekl_Potem":Czlonek_2_Rzekl_Potem).

     % Opozycyjny_etosowiec zawsze mówi prawdę. Pojedyncze
     % stwierdzenie tajnego_wspolpracownika może
     % być prawdą lub nieprawdą:
/*27*/  pojedyncze_stwierdzenie(Czlonek, Prawda) :-
/*28*/       (Czlonek &= opozycyjny_etosowiec) => Prawda.
     % Proszę to sprawdzić w programie 3_12_test_TW_OE.ecl.

     % Tajny_wspolpracownik na przemian kłamie i mówi prawdę.
     % Opozycyjny_etosowiec zawsze (kolejno) mówi prawdę:
/*29*/  kolejne_stwierdzenia(Czlonek, Prawda1, Prawda2) :-
/*30*/  (Czlonek &= tajny_wspolpracownik) #= (Prawda1 #\= Prawda2).
     % Proszę to sprawdzić w programie test_TW_OE.ecl.
```

Program generuje następujący komunikat:

```
Czlonek_1 : tajny_wspolpracownik
Czlonek_2 : opozycyjny_etosowiec
Czlonek_3 : tajny_wspolpracownik
Czlonek_2_Rzekl_Najpierw : 1
Czlonek_2_Rzekl_Potem : 1
```

Również dla tego (stosunkowo skomplikowanego) problemu uzyskano rozwiązanie tylko na drodze propagacji.

Aby lepiej zrozumieć program `3_5_black_and_white.ecl`, dobrze jest prześledzić poniższy pomocniczy program `3_6_test_TW_OE.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(ic_symbolic).
/*3*/   :-local domain(czlonek(opozycyjny_etosowiec,  tajny_wspolpracownik)).

/*4*/   top :-
% Kolejno odkomentowujemy jeden i tylko jeden z poniższych warunków.
% Dla każdego odkomentowanego warunku otrzymuje się odpowiedź Yes/No.
/*5*/     pojedyncze_stwierdzenie(opozycyjny_etosowiec, 1). %  Yes
/*6*/  %  pojedyncze_stwierdzenie(opozycyjny_etosowiec, 0). %  No
/*7*/  %  pojedyncze_stwierdzenie(tajny_wspolpracownik, 0). %  Yes
/*8*/  %  pojedyncze_stwierdzenie(tajny_wspolpracownik, 1). %  Yes
/*9*/  %  kolejne_stwierdzenia(opozycyjny_etosowiec, 1, 0). %  No
/*10*/ %  kolejne_stwierdzenia(opozycyjny_etosowiec, 1, 1). %  Yes
/*11*/ %  kolejne_stwierdzenia(opozycyjny_etosowiec, 0, 1). %  No
/*12*/ %  kolejne_stwierdzenia(tajny_wspolpracownik, 1, 0). %  Yes
/*13*/ %  kolejne_stwierdzenia(tajny_wspolpracownik, 0, 0). %  No
/*14*/ %  kolejne_stwierdzenia(tajny_wspolpracownik, 0, 1). %  Yes
/*15*/ %  kolejne_stwierdzenia(tajny_wspolpracownik, 1, 1). %  No

     % Opozycyjny_etosowiec zawsze mówi prawdę. Pojedyncze stwierdzenie
     %  tajnego_wspolpracownika może być prawdą lub nieprawdą:
/*16*/  pojedyncze_stwierdzenie(Czlonek, Prawda) :-
/*17*/      (Czlonek &= opozycyjny_etosowiec) => Prawda.

     % Tajny_wspolpracownik na przemian kłamie i mówi prawdę.
     % Opozycyjny_etosowiec zawsze (kolejno) mówi prawdę:
/*18*/  kolejne_stwierdzenia(Czlonek, Prawda1, Prawda2) :-
/*19*/  (Czlonek &= tajny_wspolpracownik) #= (Prawda1 #\= Prawda2).
```

Dla odkomentowanego warunku `/*5*/` otrzymuje się odpowiedź: `Yes`.

Celem przykładów z rozdziału 3.6 było pokazanie, że aczkolwiek algorytmy zgodnościowe nie są kompletne, w pewnych przypadkach mogą same umożliwić wyznaczenie rozwiązania. Oczywiście, zastosowanie dla tych przypadków poszukiwań (tzn. wprowadzenie predykatu `labeling/1`) nigdy nie przeszkadza w wyznaczeniu rozwiązania, a najczęściej przyśpiesza wyznaczenie rozwiązania.

3.7 Propagacja sama najczęściej nie wystarcza

Przedstawione (w poprzednim rozdziale) przykłady wyznaczanie rozwiązań wyłącznie za pomocą propagacji ograniczeń były czymś wyjątkowym. Poszukiwania z reguły są niezbędne[5]. Poniżej zostanie przedstawiona seria przykładów, dla których wspomaganie propagacji za pomocą poszukiwań jest niezbędne: sama propagacja nie doprowadza dla tych przykładów do wyznaczenia rozwiązania.

3.7.1 Trzy równania w liczbach całkowitych

Rozpatrzmy program `3_7_trzy_rownania.ecl` rozwiązywania układu trzech równań liniowych w liczbach całkowitych:

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/       [X,Y,Z]::0..6,

/*4*/       X + Y + Z #= 9,
/*5*/       write("Uwzglednienie X + Y + Z #= 9"),nl,
/*6*/       write("nie redukuje dziedzin:"),nl,
/*7*/       get_domain(X, LX),write("dziedzina X ="),write(LX),nl,
/*8*/       get_domain(Y, LY),write("dziedzina Y ="),write(LY),nl,
/*9*/       get_domain(Z, LZ),write("dziedzina Z ="),write(LZ),nl,

/*10*/      2*X+4*Y+3*Z #= 28,
/*11*/      write("Dodatkowe uwzglednienie  2*X + 4*Y +3* Z #= 28"),
/*12*/      nl,write("rowniez nie redukuje dziedzin:"),nl,
/*13*/      get_domain(X, LLX),write("dziedzina X ="),write(LLX),nl,
/*14*/      get_domain(Y, LLY),write("dziedzina Y ="),write(LLY),nl,
/*14*/      get_domain(Z, LLZ),write("dziedzina Z ="),write(LLZ),nl,
```

[5]Nawet dla problemów dających się rozwiązać wyłącznie propagacją zastosowanie poszukiwań może być zalecane, gdyż przyśpieszy wyznaczenie rozwiązania.

```
/*16*/      4*X+2*Y+Z #= 18,
/*17*/      write("A teraz uwzgledniam jeszcze 4*X + 2*Y +Z #= 18"),
/*18*/      nl,write("co redukucje dziedziny:"),nl,
/*19*/      get_domain(X, LLLX),write("dziedzina X ="),write(LLLX),
/*20*/      nl,get_domain(Y, LLLY),write("dziedzina Y ="),write(LLLY),
/*21*/      nl,get_domain(Z, LLLZ),write("dziedzina Z ="),write(LLLZ),
/*22*/      nl,write("pozostawiajac w nich duzo wartosci niezgodnych"),

/*23*/      labeling([X,Y,Z]),
/*24*/      write("Labeling inicjuje poszukiwania, redukujac  dziedziny:",nl,
/*25*/      get_domain(X, KX),write("dziedzina X ="),write(KX),nl,
/*26*/      get_domain(Y, KY),write("dziedzina Y ="),write(KY),nl,
/*27*/      get_domain(Z, KZ),write("dziedzina Z ="),write(KZ),nl,
/*28*/      write("i dajac rozwiazanie:"),nl,
/*29*/      write("X = "),write(X),nl,
/*30*/      write("Y = "),write(Y),nl,
/*31*/      write("Z = "),write(Z),fail.
/*32*/  top.
```

Rozwiązanie jest następujące:

```
Uwzglednienie X + Y + Z #= 9
nie redukuje dziedzin:
dziedzina X =[0, 1, 2, 3, 4, 5, 6]
dziedzina Y =[0, 1, 2, 3, 4, 5, 6]
dziedzina Z =[0, 1, 2, 3, 4, 5, 6]

Dodatkowe uwzglednienie  2*X + 4*Y +3* Z #= 28
rowniez nie redukuje dziedzin:
dziedzina X =[0, 1, 2, 3, 4, 5, 6]
dziedzina Y =[0, 1, 2, 3, 4, 5, 6]
dziedzina Z =[0, 1, 2, 3, 4, 5, 6]

A teraz uwzgledniam jeszcze 4*X + 2*Y +Z #= 18
co redukuje dziedziny:"
dziedzina X =[0, 1, 2, 3, 4]
dziedzina Y =[1, 2, 3, 4, 5, 6]
dziedzina Z =[0, 1, 2, 3, 4, 5, 6]
pozostawiajac w nich duzo wartosci niezgodnych.

Labeling inicjuje poszukiwania, redukujac  dziedziny:
dziedzina X =[2]
dziedzina Y =[3]
dziedzina Z =[4]
```

```
   i dajac rozwiazanie:
   X = 2
   Y = 3
   Z = 4
```

3.7.2 Golfiści ponownie

Zastosowanie ograniczeń dla dziedziny całkowitoliczbowej umożliwia również uproszczenie programu `2_10_golfisci.pl` z rozdziału 2.4.1. Ilustruje to program `3_8_golfisci.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/       [Fred,Jurek,Tomek,Robert]::1..4,
            % Tomek - zmienna określająca pozycję Tomka w szeregu.

/*4*/       [Czerwone,Pomaranczowe,Niebieskie,Biale]::1..4,
            % Niebieskie - zmienna określająca pozycję koloru
            % niebieskiego w szeregu.

   % (2) Golfista na prawo od Freda ma spodnie niebieskie:
/*5*/       Niebieskie#=Fred+1,

   % (3) Jurek jest na miejscu drugim:
/*6*/       Jurek#=2,

   % (4) Robert ma spodnie białe:
/*7*/       Robert#=Biale,

   % (5) Tomek nie jest na miejscu drugim ani na czwartym
   %     i nie ma spodni pomarańczowych:
/*8*/       Tomek#\=2,
/*9*/       Tomek#\=4,
/*10*/      Tomek#\=Pomaranczowe,

   % Golfiści są różni:
/*11*/      Fred#\=Jurek,
/*12*/      Fred#\=Tomek,
/*13*/      Fred#\=Robert,
/*14*/      Jurek#\=Tomek,
/*15*/      Jurek#\=Robert,
/*16*/      Tomek#\=Robert,

   % Kolory są różne:
/*17*/      Czerwone#\=Pomaranczowe,
```

```
/*18*/      Czerwone#\=Niebieskie,
/*19*/      Czerwone#\=Biale,
/*20*/      Pomaranczowe#\=Niebieskie,
/*21*/      Pomaranczowe#\= Biale,
/*22*/      Niebieskie#\=Biale,

/*13*/      labeling([Fred,Jurek,Tomek,Robert,Pomaranczowe,
                        Niebieskie,Czerwone,Biale]),

/*14*/      write("Fred,Jurek,Tomek,Robert"),nl,
/*15*/      write([Fred,Jurek,Tomek,Robert]),nl,
/*16*/      write("Czerwone,Pomaranczowe,Niebieskie,Biale"),nl,
/*17*/      write([Czerwone,Pomaranczowe,Niebieskie,Biale]),nl.
```

Wyświetlony komunikat jest następujący:

```
Fred,Jurek,Tomek,Robert
[1, 2, 3, 4]

Czerwone,Pomaranczowe,Niebieskie,Biale
[3, 1, 2, 4]
```

Sens jego jest następujący: np. Jurek jest na pozycji 2 w liście golfistów i nosi spodnie o kolorze odpowiadającym pozycji 2 z listy kolorów, a więc niebieskie. Czytelność komunikatu zostanie poprawiona w rozdziale 4.4.4.

Tym razem `labeling/1` jest również konieczny dla uzyskania rozwiązania: sama propagacja ograniczeń nie wystarcza.

3.7.3 Wartownicy

Konieczność stosowania poszukiwań nie ma związku z liczbą ograniczeń i liczbą zmiennych zadania. Ilustruje to następujący prosty przykład, dla rozwiązania którego poszukiwania są konieczne:

Ważna baza wojskowa jest ulokowana na terenie kwadratu i otoczona murem, w którego narożnikach i w środku każdego boku są wielopoziomowe wieże wartownicze. Wartownik znajdujący się w narożnikowej wieży wartowniczej ma możliwość obserwowania obydwu przylegających do wieży murów. Wartownik znajdujący się w środkowej wieży wartowniczej może obserwować tylko swój bok muru. Jak rozmieścić 12 wartowników, by każdy bok muru mógł być obserwo-

wany przez pięciu wartowników?

Rozmieszczenie takie wyznacza program `3_9_wartownicy.ecl`:

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/       Wartownicy = [NW,N,NE,W,E,SW,S,SE],
                %NW - liczba wartowników w narożniku NorthWest
                %E - liczba wartowników w środku boku East

/*4*/       Wartownicy :: 0..12,
/*5*/       sum(Wartownicy) #= 12,
/*6*/       NW + N + NE #= 5,
/*7*/       NE + E + SE #= 5,
/*8*/       NW + W + SW #= 5,
/*9*/       SW + S + SE #= 5,

/*10*/      labeling(Wartownicy),

/*11*/      printf("%3d%3d%3d\n", [NW,N,NE]),
/*12*/      printf("%3d %5d\n", [ W, E]),
/*13*/      printf("%3d%3d%3d\n", [SW,S,SE]).
```

Rozwiązanie ma postać następującą:

```
  0  0  5
  2     0
  3  2  0
```

Pomimo wyjątkowej prostoty przykładu, `labeling/1` jest również konieczny dla uzyskania rozwiązania: sama propagacja ograniczeń nie wystarcza.

3.7.4 Egzamin

Deklaracja dziedziny zmiennych umożliwia uproszczenie programu o sali egzaminacyjnej z rozdziału 2.4.7. Odpowiedni program clp-owy (`3_10_egzamin.ecl`) ma postać:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/      L=[M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12,M13,M14,M15,
               M16,M17],
```

```
/*4*/      L :: 1..4,
%  Sens zmiennych: jeżeli np. M7 = 3, to na miejscu M7 będzie rozwiązywane zadanie 3.

/*5*/      M1 #\= M2,     /*6*/      M1 #\= M5,
/*7*/      M1 #\= M6,     /*8*/      M1 #\= M7,
/*9*/      M2 #\= M6,     /*10*/     M2 #\= M7,
/*11*/     M2 #\= M3,     /*12*/     M2 #\= M8,
/*13*/     M3 #\= M7,     /*14*/     M3 #\= M8,
/*15*/     M3 #\= M9,     /*16*/     M3 #\= M4,
/*17*/     M4 #\= M8,     /*18*/     M4 #\= M9,
/*19*/     M5 #\= M6,     /*20*/     M5 #\= M10,
/*21*/     M5 #\= M11,    /*22*/     M6 #\= M10,
/*23*/     M6 #\= M11,    /*24*/     M6 #\= M7,
/*25*/     M6 #\= M12,    /*26*/     M7 #\= M11,
/*27*/     M7 #\= M12,    /*28*/     M7 #\= M8,
/*29*/     M7 #\= M13,    /*30*/     M8 #\= M12,
/*31*/     M8 #\= M13,    /*32*/     M8 #\= M14,
/*33*/     M8 #\= M9,     /*34*/     M9 #\= M13,
/*35*/     M9 #\= M14,    /*36*/     M10 #\= M11,
/*37*/     M11 #\= M15,   /*38*/     M11 #\= M12,
/*39*/     M12 #\= M15,   /*40*/     M12 #\= M16,
/*41*/     M12 #\= M13,   /*42*/     M13 #\= M15,
/*43*/     M13 #\= M16,   /*44*/     M13 #\= M17,
/*45*/     M13 #\= M14,   /*46*/     M14 #\= M16,
/*47*/     M14 #\= M17,   /*48*/     M15 #\= M16,
/*49*/     M16 #\= M17,
/*50*/     labeling([M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,
                    M11,M12,M13,M14,M15,M16,M17]),

/*51*/     write("   "),write(M1),write(", "),write(M2),
             write(", "),write(M3),write(", "),write(M4),nl,
/*52*/     write(M5),write(", "),write(M6),write(", "),write(M7),
             write(", "),write(M8),write(", "),write(M9),nl,
/*53*/     write(M10),write(", "),write(M11),write(", "),write(M12),
             write(", "),write(M13),write(","),write(M14),nl,
/*54*/     write("      "),write(M15),write(", "),write(M16),
             write(", "),write(M17), nl.
```

Jednym z wielu możliwych rozwiązań jest:

```
   1, 2, 1, 2
2, 3, 4, 3, 4
4, 1, 2, 1, 2
     3, 4, 3
```

Rozwiązanie to uzyskano tym razem błyskawicznie, w czasie poniżej 0.00 sekundy. Jest to piekną ilustracją większej efektywności propagacji i poszukiwań w ECL^iPS^e CPS w porównaniu z propagacją w ECL^iPS^e $Prolog$u.

3.7.5 Hetmani

Rozpatrzmy obecnie clp-ową wersję programu `2_14_hetmani_p.pl`. Program ten (`3_11_hetmani.ecl`) wyznacza również ustawienie 8 hetmanów na szachownicy, dla których żaden hetman nie atakuje żadnego innego. Jest on prostszy, szybszy i bardziej przejrzysty od swego poprzednika:

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/      hetmani(L),
/*4*/      write(L).

/*5*/   hetmani([X1,X2,X3,X4,X5,X6,X7,X8]):-
/*6*/      [X1,X2,X3,X4,X5,X6,X7,X8]::1..8,
/*7*/      bezpieczny([X1,X2,X3,X4,X5,X6,X7,X8]),
/*8*/      labeling([X1,X2,X3,X4,X5,X6,X7,X8]),
/*9*/      write([X1,X2,X3,X4,X5,X6,X7,X8]),nl,
/*10*/     fail.

/*11*/  hetmani(_):-
/*12*/     write("To juz wszystkie ustawienia bezpieczne!"),nl.

/*13*/  bezpieczny([]).
/*14*/  bezpieczny([H|T]):-
/*15*/     nie_atakuje(H,T),
/*16*/     bezpieczny(T).

/*17*/  nie_atakuje(X,Xs):-
/*18*/     nie_atakuje(X,Xs,1).

/*19*/  nie_atakuje(_,[],_).
/*20*/  nie_atakuje(X,[Y|Ys],Nb):-
/*21*/     X #\= Y,
/*22*/     X #\= Y + Nb,
/*23*/     Y #\= X + Nb,
/*24*/     Nb1 is Nb+1,
/*25*/     nie_atakuje(X,Ys,Nb1).
```

Rozwiązanie ma postać:

```
[1, 5, 8, 6, 3, 7, 2, 4]
[1, 6, 8, 3, 7, 4, 2, 5]
[1, 7, 4, 6, 8, 2, 5, 3]
.....................
[8, 2, 5, 3, 1, 7, 4, 6]
[8, 3, 1, 6, 2, 5, 7, 4]
[8, 4, 1, 3, 6, 2, 7, 5]
```

Również tym razem `labeling/1` jest konieczny dla uzyskania rozwiązania: sama propagacja ograniczeń nie wystarcza.

3.7.6 Konfigurowanie

Konfigurowanie metodą poszukiwań i propagacji znanego już z rozdziału 2.2.3 systemu 3-elementowego wykonuje program `3_12_konf_CLP.ecl`:

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/       Konf=[A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2],
/*4*/       Konf :: 0..1,
   %   Sens zmiennych: jeżeli np. A_1 = 0, to konfiguracja nie zawiera elementu A_1
/*5*/       Koszt:: 1..2100,

/*6*/       A_1 + A_2 + A_3 #= 1, % konfiguracja zawiera jeden element typu A
/*7*/       B_1 + B_2 + B_3 + B_4 #= 1, % konfiguracja zawiera jeden element typu B
/*8*/       C_1 + C_2 #= 1, % konfiguracja zawiera jeden element typu C
/*9*/       C_1 + A_2 #=< 1,  % C_1 i A_2 są niekompatybilne
/*10*/      B_2 + C_2 #=< 1,  % B_2 i C_2 są niekompatybilne
/*11*/      C_2 + B_3 #=< 1,  % C_2 i B_3 są niekompatybilne
/*12*/      B_4 + A_2 #=< 1,  % B_4 i A_2 są niekompatybilne
/*13*/      B_3 + A_1 #=< 1,  % B_3 i A_1 są niekompatybilne
/*14*/      A_3 + B_3 #=< 1,  % A_3 i B_3 są niekompatybilne

/*15*/      Koszt #= A_1 * 1900 + A_2 * 750 +
              A_3 * 900 + B_1 * 300 + B_2 * 500 + B_3 * 450 +
                  B_4 * 600 + C_1 * 700 + C_2 * 850,

/*16*/      labeling(Konf),

/*17*/      writeln("Dopuszczalna konfiguracja o koszcie":Koszt),
/*18*/      write(Koszt),write(":"),nl,
/*19*/      pisz_konfiguracja([A_1,A_2,A_3,B_1,B_2,B_3,B_4,
/*20*/            C_1,C_2],
```

```
/*21*/      ["A_1","A_2","A_3","B_1","B_2","B_3","B_4",
/*22*/           "C_1","C_2"]),nl,nl, fail.

/*23*/   top :-
/*26*/      write("To wszystkie dopuszczalne konfiguracje.").

/*27*/   pisz_konfiguracja([H1|T1],[H2|T2]):-
/*24*/      H1 is 1, write(H2),write("  "),
/*25*/      pisz_konfiguracja(T1,T2).

/*26*/   pisz_konfiguracja([H1|T1],[_|T2]):-
/*27*/      H1 is 0,
/*28*/      pisz_konfiguracja(T1,T2).

/*29*/   pisz_konfiguracja([],[]).
```

Rozwiązanie ma postać:

```
Dopuszczalna konfiguracja o koszcie 2100:
A_3  B_2  C_1
Dopuszczalna konfiguracja o koszcie 2050:
A_3  B_1  C_2
Dopuszczalna konfiguracja o koszcie 1900:
A_3  B_1  C_1
Dopuszczalna konfiguracja o koszcie 1900:
A_2  B_1  C_2
To wszystkie dopuszczalne konfiguracje.
```

Również dla tego przykładu `labeling/1` jest konieczny dla uzyskania rozwiązania.

3.8 Zadania

Równania i reguły 1

Napisz program wyznaczający rozwiązanie w liczbach całkowitych (możliwie najmniejszych) dla następujących równań i reguł:
A + E = G
C + D = 10
E + F = 8
A + C = 6
Jeżeli F <> 6 to H > F

Jeżeli E <> 1 to H > B
Jeżeli G <> 8 to F > B
Jeżeli B <> 5 to G <> 5
Jeżeli E <> 3 to C <> 4
gdzie <> oznacza dyzrówność.

Rozwiązanie:
A=4, B=1, C=2, D=8, E=3, F=5, G=7, H=6

Równania i reguły 2

Napisz program wyznaczający rozwiązanie w liczbach całkowitych (możliwie najmniejszych) dla następujących równań i reguł:
B + G = D
B + C = A
C + E + G = F
Jeżeli D < A to C = 2
Jeżeli D >= A to E = 2.

Rozwiązanie:
A=5, B=4, C=1, D=7, E=2, F=6, G=3

Wizyta

Rodzina Smith ze swą trójką dzieci chciałaby pójść z wizytą do dalekich krewnych, ale wszyscy jej członkowie mają bardzo dziwne fochy jeżeli chodzi o udział w wizycie, a mianowicie:
1. Pan Smith pójdzie, jeżeli jego żona pójdzie również.
2. Co najmniej jeden z ich synów, Matt i John, pójdzie również.
3. Albo Pani Smith albo Tim pójdzie, ale obydwoje nie pójdą razem.
4. Albo Tim i John pójdą, albo żaden z nich.
5. Jeżeli Matt pójdzie, wówczas John i jego ojciec pójdą również.
Napisz program wyznaczający osoby, które pójdą i które nie pójdą z wizytą.

Końskie derby

Na ostatnich wyścigach konnych 10 dobrych koni ukończyło wyczerpujący 3 milowy wyścig. Co było do przewidzenia, jak co roku rezultaty tajemniczo zaginęły. Jednakże, różni obserwatorzy zapamiętali następujące urywki informacji: Sylwester przegrał z Zebra Wings. Zebra Wings pokonała

Sylwestra, Frogman's Flippers i Tweetie Pie. Fizzy Pop przegrał z Minty Mouse, Sylwestrem i CD Player. Frogman's Flippers pokonał Windy Miller, CD Player i Sylwestra. Top Trumps przegrał z CD Player, Kool Kat i Tweetie Pie. CD Player pokonał Top Trumps i Fizzy Pop. Tweetie Pie przegrał z Zebra Wings i Sylwestrem. Kool Kat przegrał z Tweetie Pie i Frogman's Flippers. Frogman's Flippers pokonał Fizzy Pop, Minty Mouse i CD Player. CD Player przegrał z Frogman's Flippers, Kool Kat i Tweetie Pie. Top Trumps pokonał Fizzy Pop i Windy Miller. Minty Mouse przegrał z Windy Miller i Sylwestrem. Windy Miller przegrała z Tweetie Pie i CD Player. Napisz program, który określi kto skończył na którym miejscu.

Targi Wiosenne

Na ostatnich Targach Wiosennych czterech znanych ogrodników[6] chwaliło się swymi dokonaniami w zakresie hodowli róż. Róże były w czterech kolorach. Każdy ogrodnik przedstawiał dwie róże w dwóch różnych kolorach. Pan Green miał żółtą różę. Pan Yellow nie miał czerwonej róży. Jeden z ogrodników z czerwoną różą miał również zieloną różę. Jeden z ogrodników z żółtą również miał niebieską. Jeden z ogrodników z zieloną różą nie miał czerwonej. Żaden ogrodnik z żółtą różą nie miał zielonej. Żaden ogrodnik nie przedstawiał dwóch róż tego samego koloru. Żadna para ogrodników nie przedstawiała pary róż o tych samych kolorach. Napisz program wyjaśniający, kto przedstawiał róże jakiego koloru.
Rozwiązanie:

Ogrodnik	Róża 1	Róża 2
Pan Blue	czerwona	zielona
Pan Green	żółta	czerwona
Pan Red	niebieska	żółta
Pan Yellow	niebieska	zielona

Tablica 3.1: Rozwiązanie dla "Targów Wiosennych"

[6]Zadania pochodzi z http://www.brainbashers.com/

Kradzież ciasta

Podczas ostatniego policyjnego dochodzenia, Główny Inspektor Stone przesłuchiwał pięciu lokalnych przestępców w celu zidentyfikowania, kto ukradł ciasto Pani Kowalskiej z jarmarku świętojańskiego. Poniżej znajduje się podsumowanie ich zeznań:

1)Arek: to nie był Edek, to był Bolek.
2)Bolek: to nie był Czarek, to nie był Edek.
3)Czarek: to był Edek, to nie był Arek.
4)Darek: to był Czarek, to był Bolek.
5)Edek: to był Darek, to nie był Arek.

Było wiadome, że każdy z podejrzanych powiedział dokładnie jedno kłamstwo. Napisz program, który wskaże kto ukradł ciasto.

Wyścig koni

Hazardzista - miłośnik puzzli postawił znaczną sumę w wyścigu koni. Droczący się z nim znajomy bookmacher nie chciał mu jednak wyjawić wyników wyścigu, jeżeli na podstawie fragmentarycznych danych nie odgadnie, w jakiej kolejności konie skończyły bieg, przy czym nie wykluczył zajęcia tego samego miejsca przez kilka koni. Fragmentaryczne dane były następujące:
1. Penuche Fudge ukończył bieg przed Near Miss i za Whispered Promises. 2. Whispered Promises zremisował z Penuche Fudge tylko wtedy gdy Happy Go Lucky nie zremisował z Skipper's Gal. 3. Penuche Fudge ukończył bieg za Skipper's Gal tyle samo miejsc ile Skipper's Gal za Whispered Promises wtedy i tylko wtedy gdy Whispered Promises ukończył bieg za Near Miss. Hazardzista pomyślał chwilę i odpowiedział poprawnie. Napisz program wyznaczający kolejność koni na mecie.

Oceny

Piątka brytyjskich maturzystów wybrała na maturę (zwaną w Zjednoczonym Królestwie A-level) taką samą kombinację przedmiotów. Każdy z nich dostał inną ocenę z każdego przedmiotu, i nie było dwóch mających taką samą ocenę z tego samego przedmiotu. Stosowane są oceny A, B, C, D i E, ocena A jest najlepsza. Napisz program wyznaczający oceny każdego maturzysty z każdego przedmiotu, jeżeli wiadomo, że:
- Andrew był lepszy od Bridget z Fizyki oraz od Neil z matematyki.
- Wendy była jedyną dziewczyną, która miała C, ale nie miała ani jednego A.
- Osoba mająca E z matematyki, ma także B z chemii, ale nie ma C z

fizyki.
- Ocena Paula z fizyki to D a jego najwyższą oceną jest C.
- Oceny B z matematyki nie dostała ta sama osoba, który dostała E z fizyki.
- Bridget najlepsza była z chemii, a jej ocena z matematyki była niższa niż Paula.

Rozwiązanie: * Andrew dostał A z matematyki, C z chemii i B z fizyki.
Bridget dostała D z matematyki, A z chemii i E z fizyki.
Neil dostał E z matematyki, B z chemii i A z fizyki.
Paul dostał C z matematyki, E z chemii i D z fizyki.
Wendy dostała B z matematyki, D z chemii i C z fizyki.

Szlak Jesiennych Liści

Każdą jesienią tysiące turystów jeżdżą Szlakiem Jesiennych Liści by cieszyć się barwami nowej pory roku. Szlak zaczyna się w Summerset i idzie na północ 100 mil do Fallbrook. Pięć malowniczych miejsc uatrakcyjnia jazdę, każde zapewniające parking wzdłuż wąskiej drogi ze spektakularnym widokiem na różne atrakcje; każde malownicze miejsce jest oznaczone na innym słupie informacyjnym wyznaczającym ich odległość od Summerset w milach. Wziąwszy pod uwagę poniższe dane z mapy drogowej, napisz program określający w jakiej odległości od Summerset wzdłuż Szlaku Jesiennych Liści jest ulokowany każdy punkt widokowy, jeżeli:
1. Żadne dwa kolejne malownicze miejsca nie są jednakowo odległe; najdłuższa odległość między dwoma kolejnymi miejscami (uwzględniając również punkty końcowy i poczatkowy) na szlaku wynosi 36 mil, natomiast najkrótsza wynosi 4 mile.
2. Odległość wzdłuż Szlaku Jesiennych Liści od Summerset do Cucumber Creek równa się odległości od Old Man Mountain do White Oak Inn.
3. White Oak Inn jest na północ od Old Man Mountain. 4. Amish Covered Bridge, który nie jest ostatnim malowniczym miejscem wzdłuż trasy, jest 10 mil na południe od Fallbrook.
5. Cucumber Creek jest dwukrotnie dalej od Sugar Maple Farm niz od Old Man Mountain.
6. White Oak Inn i Cucumber Creek są odległe o więcej niż 50 mil.
7. Pierwsze malownicze miejsce na szlaku jest oznaczone na słupie informacyjnym 18 mil na północ od Summerset.

Rozwiązanie:

```
100     Fallbrook (północ)
96      White Oak Inn
90      Amish Covered Bridge
54      Old Man Mountain
42      Cucumber Creek
18      Sugar Maple Farm
0       Summerset (południe)
```

Ogrody

Pięciu przyjaciół ma sąsiadujące ogrody, gdzie uprawiają najróżnorodniejsze uprawy: owoce (jabłka, gruszki, orzechy, wiśnie), jarzyny (marchewkę, pietruszkę, dynię, cebulę) i kwiaty (astry, róże, tulipany, lilie)[7].

1. Łącznie mają 12 upraw.
2. Każdy z nich uprawia cztery różne uprawy.
3. Każda uprawa jest co najmniej w jednym ogrodzie.
4. Tylko jedna uprawa jest w czterech ogrodach.
5. Tylko w jednym ogrodzie są wszystkie trzy uprawy.
6. Tylko w jednym ogrodzie są wszystkie cztery rodzaje jednej z upraw.
7. Gruszki są tylko w dwóch granicznych ogrodach.
8. Ogród Pawła jest w środku i nie ma lilii.
9. Hodowca aster nie uprawia jarzyn.
10. Hodowca róż nie uprawia pietruszki.
11. Hodowca orzechów ma w ogrodzie również dynie i pietruszki.
12. W pierwszym ogrodzie są jabłka i wiśnie.
13. Wiśnie są tylko w dwóch ogrodach.
14. Stefan uprawia cebulę i wiśnie.
15. Łukasz uprawia dwa rodzaje owoców.
16. Tulipany są tylko w dwóch ogrodach.
17. Jabłka są tylko w jednym ogrodzie.
18. Tylko w jednym ogrodzie obok ogrodu Zygmunta jest pietruszka.
19. Ogród Stefana nie jest ogrodem granicznym.
20. Henryk nie uprawia ani jarzyn, ani aster.
21. Paweł uprawia trzy rodzaje jarzyn.

Napisz program określający kto, co i gdzie uprawia.

Rozwiązanie:

[7] Zadanie zaczerpnięte z http://www.mathsisfun.com/puzzles

Ogrodnik	Uprawy			
Henryk	gruszki	jabłka	wiśnie	róże
Stefan	wiśnie	cebula	róże	tulipany
Paweł	marchewka	dynia	cebula	róże
Zygmunt	astry	róże	tulipany	lilie
Łukasz	gruszki	orzechy	dynia	pietruszka

Tablica 3.2: Rozwiązanie dla "Ogrodów"

Dni Otwarte [8]

Pięciu studentów (trzy dziewczyny: Celina, Danka i Ewa, oraz dwóch chłopców: Arek i Bolek) organizują w tym roku dni otwarte szkoły. Każdy ze studentów studiuje inny kierunek (geografia, języki, matematyka, filozofia, rzeźbiarstwo). Niektórzy ze studentów zaciągnęli jednego lub więcej swoich krewnych do pomocy w organizowaniu dni otwartych (mamę, tatę, babcię), ale żaden student nie zaciągnął więcej niż jednego krewnego. Napisz program, który wskaże:

• jakie są nazwiska studentów (Marzec, Kwiecień, Maj, Czerwiec, Lipiec),

• jakie studiują kierunki,

• jacy krewni pomagają przy organizowaniu dni otwartych,

jeżeli wiadomo, że:

1. Maj (to nie jest Bolek ani Ewa) nie studiuje filozofii.
2. Dwóch krewnych studenta o nazwisku Czerwiec bierze udział w pomaganiu przygotowywania dni otwartych w szkole.
3. Lipiec ma mniej swoich krewnych niż przynajmniej jeden inny student.
4. Osoba studiująca rzeźbiarstwo jest jedyną osobą, która nie zaciągnęła do pomocy do przygotowania dni otwartych w szkole żadnego ze swoich krewnych.
5. Studenci Celina i Kwiecień zaciągnęli swojego jednego rodzica do pomocy, ale nikt z nich nie zaciągnął babci i nikt z nich nie studiuje matematyki.
6. Bolek i osoba studiująca geografię albo obydwaj zaciągnęli swoich ojców do pomocy, albo żaden z nich tego nie zrobił.
7. Dwóch studentów tej samej płci nie zaciągnęło swoich mam do pomocy.
8. Tata Marzec nie uczestniczy w organizowaniu dni otwartych szkoły.
9. Ewa ma jednego krewnego więcej, niż osoba studiująca matematykę.
10. Danka studiuje języki i jej tata nie pomaga w organizowaniu dni otwar-

[8] Przykład z http://brownbuffalo.sourceforge.net/

tych szkoły.

Zawody pływackie

Pięciu zawodników - A, B, C, D i E - brało udział w zawodach pływackich uzyskując złoty, srebrny i brązowy medal za pierwsze trzy miejsca. Każde z następujących zdań złożonych jest fałszywe, ale w każdym jedno z dwóch może być prawdą.
- A nie otrzymał złotego medalu i B nie otrzymał medalu srebrnego.
- D nie otrzymał srebrnego medalu i E nie otrzymał medalu brązowego.
- C otrzymał medal, D nie otrzymał medalu.
- A otrzymał medal, C nie otrzymał medalu.
- D i E obaj uzykali medale.

Napisz program wyjaśniający, kto otrzymał jaki medal.

Rozwiązanie:

Zawodnik A uzyskał medal złoty.
Zawodnik B nie uzyskał medalu.
Zawodnik C uzyskal medal brązowy.
Zawodnik D uzyskał medal srebrny.
Zawodnik E nie uzyskał medalu.

Kolejka po bilety [9]

Pięć osób stoi w kolejce po bilety na samolot Każda z nich ma imię, jest w określonym wieku, ma ulubioną witrynę internetową, ma miejsce stałego zamieszkania i cel przelotu, a mianowicie: ich imiona to Arek, Bolek, Czarek, Darek i Edek, ich wiek to 14, 21, 46, 52 i 81, ich ulubione witryny internetowe to "St. Michalkiewicz", "J.Korwin-Mikke", "R.A.Ziemkiewicz", "Rebelya" i "Antysocial", mieszkają w Warszawie, w Gdańsku, w Gliwicach, w Krakowie i w Łodzi, ich zawody to architekt, dekarz, uczeń, lekarz i barman, ich celem jest Waszyngton, Wrocław, Berlin, Rzym i Londyn. Ponadto:
1. Osoba w środku kolejki odwiedza witrynę "Rebelya".
2. Arek jest pierwszy w kolejce.
3. Osoba odwiedzająca witrynę "St. Michalkiewicz" jest następną przed osobą mieszkającą w Łodzi.
4. Osoba udająca się do Rzymu stoi za Czarkiem.
5. Osoba mieszkająca w Gliwicach ma 52 lata.

[9] Przykład na podstawie http://www.mathsisfun.com/puzzles

6. Osoba wybierająca się do Wrocławia jest uczniem.
7. Osoba udająca się do Rzymu odwiedza witrynę "Rebelya".
8. 14-latek jest na końcu kolejki.
9. Edek odwiedza witrynę "R.A.Ziemkiewicz".
10. Osoba udająca się do Londynu jest dekarzem.
11. Bolek mieszka w Gliwicach.
12. 46-latek jest barmanem.
13. Czwarta osoba w kolejce udaje się do Berlina.
14. Lekarz i osoba mieszkająca w Gliwicach stoją obok siebie.
15. Osoba odwiedzająca witrynę "J.Korwin-Mikke" stoi obok osoby udającej się do Waszyngtonu.
16. Osoba obok Czarka jest architektem.
17. 21-latek mieszka w Łodzi.
18. Osoba odwiedzająca witrynę "J.Korwin-Mikke" jest dekarzem.
19. 81-latek mieszka w Krakowie.
20. Osoba udająca się do Waszyngtonu mieszka w Warszawie.

Napisz program określający imiona, wiek, ulubione witryny internetowe, miejsca zamieszkania, zawód i cel przelotu wszystkich osób stojących w kolejce.

Naukowy jarmark

Arek i Bolek przedstawiają wyniki Międzynarodowego Jarmarku Naukowego[10], na którym m.in. konkurowało trzech zawodników, Ludwik, Romek i Janek. Arek twierdzi, że Ludwik wygrał konkurs, a Romek był drugi. Natomiast Bolek twierdzi, że Janek wygrał konkurs, a Ludwik był drugi. Naprawdę to ani Arek ani Bolek nie ocenili poprawnie wyników konkursu. Każdy z nich miał jedno stwierdzenie prawdziwe, a drugie - fałszywe. Napisz program wyznaczający pozycje trzech uczestników konkursu.

Rozwiązanie: Janek wygrał; Romek był drugi; Ludwik był trzeci.

[10] zob. http://www.rinkworks.com/brainfood/

Rozdział 4

CLP z predykatami globalnymi dla rozwiązań dopuszczalnych

4.1 Uwagi wstępne

Pojęcia wprowadzone w tym rozdziale oraz w rozdziale 6 mają podstawowe znaczenie dla modelowania i rozwiązywania skomplikowanych kombinatorycznych problemów decyzyjnych. Opracowanie wydajnych platform dla modelowania i rozwiązywania problemów typu PSO i POSO wymaga przede wszystkim zbioru adekwatnych podstawowych pojęć i predykatów odpowiadających tym pojęciom. Dla ciagłych systemów statycznych i dynamicznych, będących przedmiotem rozważań w mechanice, elektrotechnice i teorii sterowania, potrzebne pojęcia zostały opracowane i dojrzewały na przestrzeni wieków, poczynając od pionierskich prac Newtona i Leibnitza nad równaniami różniczkowymi. Dla problemów kombinatorycznych rozwój pojęć podstawowych rozpoczął się *de facto* dopiero w początkach prac nad *Prolog*iem i *CLP*, a kulminował opracowaniem szeregu bardzo przydatnych i mocno abstrakcyjnych pojęć, zaimplementowanych w postaci tzw. *predykatów globalnych*. *Predykatami globalnymi* nazywa się predykaty standardowe definiujące bardzo złożone relacje, najczęściej o różnorodnych argumentach, które mogą być w szeregu listach. Idea predykatów globalnych została po raz pierwszy przedstawiona i uzasadniona w publikacji

[Baldiceanu-94] dla systemu *CLP* o nazwie *CHIP*, gdzie zwrócono uwagę na to, że specyficzne ograniczenia występujące w pewnych często spotykanych aplikacjach trudno modelować w sposób efektywny za pomocą predykatów elementarnych, oraz na to, że aplikacje te wymagają stworzenia specjalnych abstrakcji bardzo wysokiego poziomu. Idea ta zaowocowała obszernymi zbiorami predykatów globalnych (zob. [Baldiceanu-10]), z których duża liczba została zaimplementowana w ECL^iPS^eCLP.

Platforma ECL^iPS^eCLP udostępnia szereg specjalistycznych predykatów globalnych w bibliotekach `ic_global`, `ic_cumulative`, `ic_edge_finder`, `ic_edge_finder3`. Stosowanie predykatów globalnych bardzo poprawia czytelność, deklaratywność i efektywność programów *CLP* wyznaczających rozwiązania dopuszczalne, przy bardzo znacznym zmniejszeniu czasu potrzebnego dla opracowania programu. Stosowane w ECL^iPS^e predykaty globalne zostały zestawione (wraz ze standardowymi predykatami elementarnymi) w spisie znajdującym się w opcji *Alphabetical Predicate Index* menu *ECLiPSe Documentation* z rysunku 5. Omawiane poniżej predykaty globalne `alldifferent/1` i `element/3` wymagają umieszczenia w programie deklaracji[1]:

```
:- lib(ic_global).
```

lub

```
:- use_module(library(ic_global)).
```

4.2 Predykat 'alldifferent/1'

Predykat ten jest spełniony, gdy wszystkie zmienne z listy `Lista=[X1,...,Xn]` mają różne wartości ze swych dziedzin. Predykat ten jest jednym z najczęściej stosowanych i najbardziej użytecznych. Teoretycznie odpowiada on sytuacji:

```
X1 #\= X2,
X1 #\= X3,
.......

X1 #\= Xn,
X2 #\= X3,
```

[1]Biblioteka ic obsługuje również predykaty 'alldifferent/1' i 'element/3', ale czyni to w sposób mniej efektywny.

```
.......

X2 #\= Xn,
.......

X(n-1) #\= Xn,
```

Jednakże sposób propagacji stosowany w predykacie `alldifferent([X1,...,Xn])` jest znacznie bardziej efektywny aniżeli stosowany dla powyższego zapisu. Rozpatrzmy przykład `4_1_all_diff.ecl`:

```
/*1*/   :- lib(ic).

/*2*/   top_1:-
/*3*/       [X,Y,Z]::1..4,
/*4*/       ic_global: alldifferent([X,Y,Z]),
/*5*/       indomain(X),
/*6*/       indomain(Y),
/*7*/       indomain(Z),
/*8*/       writeln("X":X),
/*9*/       writeln("Y":Y),
/*10*/      writeln("Z":Z),
/*11*/      fail.

/*12*/  top_1:-
/*13*/      write("Koniec."),nl.

/*14*/   top_2:-
/*15*/      [X,Y,Z]::1..4,
/*16*/       alldifferent([X,Y,Z]),
/*17*/       indomain(X),
/*18*/       indomain(Y),
/*19*/       indomain(Z),
/*20*/       writeln("X":X),
/*21*/       writeln("Y":Y),
/*22*/      writeln("Z":Z),
/*23*/      fail.

/*24*/  top_2:-
/*25*/      write("Koniec."),nl.
```

Aby uniknąć tego, że ECL^iPS^e nie bedzie wiedział, czy definicję `alldifferent/1` wziąć z biblioteki `ic_global`, czy też z zawsze potrzebnej biblioteki `ic`, deklaracja `ic_global:` została w części `top_1` umieszczona tylko w wierszu `/*4*/` dla predykatu `alldifferent/1`.

Rozwiązanie (dla `top_1` lub `top_2`) ma postać następującą:

```
  X =1 Y =2 Z =3
  X =1 Y =2 Z =4
  X =1 Y =3 Z =2
  X =1 Y =3 Z =4
  X =1 Y =4 Z =2
  X =1 Y =4 Z =3
  X =2 Y =1 Z =3
  X =2 Y =1 Z =4
  X =2 Y =3 Z =1
  X =2 Y =3 Z =4
  X =2 Y =4 Z =1
  X =2 Y =4 Z =3
  X =3 Y =1 Z =2
  X =3 Y =1 Z =4
  X =3 Y =2 Z =1
  X =3 Y =2 Z =4
  X =3 Y =4 Z =1
  X =3 Y =4 Z =2
  X =4 Y =1 Z =2
  X =4 Y =1 Z =3
  X =4 Y =2 Z =1
  X =4 Y =2 Z =3
  X =4 Y =3 Z =1
  X =4 Y =3 Z =2
```

Koniec.

Oczywiście, gdy brak rozwiązania, jak to ma miejsce np. w przypadku programu `4_2_all_diff.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/       [V,W,X,Y,Z]::1..4,
/*4*/       ic_global: alldifferent([V,W,X,Y,Z]),
/*5*/       indomain(V),
/*6*/       indomain(W),
/*7*/       indomain(X),
/*8*/       indomain(Y),
```

```
/*9*/       indomain(Z),
/*10*/      writeln('V':V),
/*11*/      writeln('W':W),
/*12*/      writeln('X':X),
/*13*/      writeln('Y':Y),
/*14*/      writeln('Z':Z).
```

otrzymujemy odpowiedź:

```
No.
```

4.3 Predykat 'element/3'

Predykat ten, przedstawiający relację pomiędzy elementem listy zmiennych całkowitych a jego pozycją w liście, ma postać:

```
element(?Pozycja_elementu_listy, ++Lista, ?Element_listy)
```

i jest spełniony, jeżeli `Element_listy` znajduje się w liście `Lista` na pozycji `Pozycja_elementu_listy`, gdzie `Lista` jest uziemioną listą lub tablicą liczb całkowitych, `Element_listy` jest wolną lub uziemioną zmienną całkowitą, a `Pozycja_elementu_listy` jest również wolną lub uziemioną zmienną całkowitą. Predykat ten jest użyteczny właśnie dlatego, że zarówno `Pozycja_elementu_listy` jak i `Element_listy` mogą być zmiennymi wolnymi, co ułatwia rozwiązywanie licznych problemów indeksowania list lub tablic.
Ilustruje to program `4_3_element.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top:-
/*3*/        ic_global: element(Pozycja,[2,1,4,3],2),
/*4*/       writeln("Pozycja ":Pozycja),
/*5*/        ic_global: element(3,[](2,1,4,3),Element_na_pozycji),
/*6*/       writeln("Element_na_pozycji ":Element_na_pozycji).
```

Program generuje komunikat:

```
Pozycja : 1
Element_na_pozycji : 4.
```

Rozpatrywane poniżej przykłady można uważać za należące do następujących dwóch grup problemów:

1. Problemy kombinatorycznego przyporządkowania, których istota polega na odpowiednim łączeniu elementów szeregu zbiorów.

2. Problemy kombinatorycznego szeregowania, których istota polega na ustawieniu elementów pewnego zbioru w szeregi wykazujące określone prawidłowości.

Przymiotnik *kombinatoryczny* służy odróżnianiu tych problemów od ich imienników rozpatrywanych w rozdziale 5 przy dodatkowym wymaganiu optymalizacji pewnego wskaźnika jakości.

4.4 Kombinatoryczne przyporządkowanie

Istotą problemów *przyporządkowania* jest wyznaczenie, dla każdego elementu jednego zbioru, pewnych elementów innych zbiorów w sposób spełniający *ograniczenia przynależnosci*. Ograniczenia przynależności definiują związki łączące elementy różnych zbiorów.

4.4.1 Send More Money

Przykład ten należy do folkloru przykładów programowania w logice z ograniczeniami:

Należy literom S,E,N,D,M,O,R,Y przyporządkować takie cyfry ze zbioru 0,1,2,3,4,5,6,7,8,9, by dodawanie:

```
  S E N D
  M O R E
----------
M O N E Y
```

było poprawne. Rozwiązanie przedstawia program `4_4_smm.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  top:-
```

```
/*3*/     sendmore(_).
/*4*/  sendmore(L) :-
/*5*/     L = [S,E,N,D,M,O,R,Y],
/*6*/     L :: [0..9],
/*7*/     alldifferent(L),
/*8*/     S #\= 0,
/*9*/     M #\= 0,
/*10*/    1000*S+100*E+10*N+D+1000*M+100*O+10*R+E#=10000*M+1000*O+100*N+10*E+Y,
/*11*/    labeling(L),
/*12*/    write("  "),write(S),write(E),write(N),write(D),
/*13*/    write("   S E N D"),nl,
/*14*/    write(" "),write(M),write(O),write(R),write(E),
/*15*/    write("   M O R E"),nl,
/*16*/    write(" ----------------"),nl,
/*17*/    write(" "),write(M),write(O),write(N),write(E),write(Y),
/*18*/    write(" M O N E Y"),nl.
```

Rozwiązaniem jest:

```
 9567      S E N D
 1085      M O R E
------  ----------
10652    M O N E Y
```

4.4.2 FIFTEEN

Bardziej zaawansowaną łamiugłówką kryptograficzną jest znana pod nazwą FIFTEEN: dana jest następująca suma:

```
        @
    @@@FIVE
    @@FIVE@
  + @FIVE@@
  ------------
    FIFTEEN
```

Różne litery odpowiadają różnym cyfrom, jednakowe litery odpowiadają jednakowym cyfrom, znak @ odpowiada dowolnej cyfrze, a cyfry na najbardziej

znaczących pozycjach nie mogą być zerami. Jeżeli `FIVE` jest podzielne przez `5` i `ELEVEN` jest podzielne przez `11`, należy wyznaczyć liczbę odpowiadającą `FIFTEEN`. Odpowiedni program `4_5_FIFTEEN.ecl` ma postać:

```
/*1*/  :- lib(ic).
/*2*/  top:-
/*3*/     assert(licznik(0)),
/*4*/     P = [F,I,V,E,T,N,L,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10],
/*5*/     P :: [0..9],
/*6*/     alldifferent([F,I,V,T,N,L,E]),

/*7*/     F #\= 0,
/*8*/     E#\=0,
/*9*/     E #= 5,
/*10*/    A1#\=0,
/*11*/    A4#\=0,
/*12*/    A7#\=0,
/*13*/    A10#\=0,
/*14*/    moje_modulo_5(F,I,V,E),
/*15*/    moje_modulo_11(E,L,E,V,E,N),

/*16*/    A1 +
/*17*/    E + 10*V + 100*I + 1000*F + 10000*A2 + 100000*A3 + 1000000*A4 +
/*18*/    A5 + 10*E + 100*V + 1000*I + 10000*F + 100000*A6 + 1000000*A7 +
/*19*/    A8 + 10*A9 + 100*E + 1000*V + 10000*I + 100000*F + 1000000*A10 #=
/*19*/    1000000*F + 100000*I+ 10000*F + 1000*T + 100*E + 10*E + N,

/*20*/    labeling(P),
/*21*/    count,
/*22*/    licznik(Numer_rozwiazania),
/*23*/    write("Solution "), write(Numer_rozwiazania),write(":"),nl,

/*24*/    write("  "),write("  "),write("  "),write(A1),
/*25*/    write("  "),write("  "),write("  "),write("  "),write("  "),
                  write("  "),write("  "),write("  "),write("  A1"),nl,
/*26*/    write(A4),write(A3),write(A2),write(F),write(I),write(V),write(  E),
/*27*/    write("  A4 A3 A2 F I  V  E"),nl,
/*28*/    write(A7),write(A6),write(F),write(I),write(V),write(E),write(A5),
/*29*/    write("  A7 A6  F I V  E A5"),nl,
/*30*/    write(A10),write(F),write(I),write(V),write(E),write(A9),write(A8),
/*31*/    write(" A10  F  I V E A9 A8"),nl,
/*32*/    write("------- --------------------"),nl,
/*33*/    write(F),write(I),write(F),write(T),write(E),write(E),write(N),
/*34*/    write("  F  I  F  T  E  E  N"),nl,nl,
/*35*/    fail.

/*36*/ top:-nl,nl,
/*37*/    write("To wszystkie rozwiazania!").
```

```
/*38*/ moje_modulo_5(F,I,V,E):-
/*39*/    integers(X),
/*40*/    E+10*V+100*I+1000*F #= X*5.

/*41*/ moje_modulo_11(E,L,E,V,E,N):-
/*42*/    integers(X),
/*43*/    N+10*E+100*V+1000*E+10000*L+100000*E #= X*11.

/*44*/ count:-
/*45*/    retract(licznik(Stary)),
/*46*/    Nowy is Stary + 1,
/*47*/    assert(licznik(Nowy)).
```

Problem ma 18 rozwiązań dla tej samej liczby `FIFTEEN`. Pierwsze i ostatnie są następujące:

```
    Rozwiazanie 1:
      8                   A1
1094085  A4 A3 A2  F  I  V  E
1540859  A7 A6  F  I  V  E A5
1408599 A10  F  I  V  E A9 A8
------- --------------------
4043551   F  I  F  T  E  E  N

.........................

    Rozwiazanie 18:
      9                   A1
1594085  A4 A3 A2  F  I  V  E
1040859  A7 A6  F  I  V  E A5
1408598 A10  F  I  V  E A9 A8
------- --------------------
4043551   F  I  F  T  E  E  N
```

4.4.3 Kto był z kim raz jeszcze

Obydwa poznane predykaty globalne umożliwią inne zaprogramowanie łamigłówki *Kto był z kim?* z rozdziału 3.6.2 i rozwiązanie problemu generowania czytelnego komunikatu o rozwiązaniu.
Odpowiedni program `4_6_kto_z_kim_raz_jeszcze.ecl` ma postać:

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/       [Andrzej,Bronek,Czarek,Darek] :: [1..4],
/*4*/       [Ola, Ewa,Paulina,Sabina] :: [1..4],
          % koncert=1, kino=2, teatr=3, wystawa = 4
          % co znaczy: jeżeli np. Bronek=Ola=4, to
          % Bronek z Olą poszli na wystawę

/*5*/       Andrzej#=1,     % Andrzej poszedł na koncert
/*6*/       Bronek#=Ola,    % Bronek spędził wieczór z Olą
/*7*/       Czarek#\=Ewa,   % Czarek nie widział Ewy
/*8*/       Paulina#=2,     % Paulina była w kinie
/*9*/       Ewa#=3,         % Ewa była w teatrze

/*10*/      alldifferent([Andrzej,Bronek,Czarek,Darek]),
/*11*/      alldifferent([Ola,Ewa,Paulina,Sabina]),

/*12*/      write(Andrzej),write(" "),write(Bronek),write(" "),
/*13*/      write(Czarek),write(" "),write(Darek),nl,
/*14*/      write(Ola),write(" "),write(Ewa),write(" "),
/*15*/      write(Paulina),write(" "),write(Sabina),nl,nl,

  % Tu otrzymujemy ponownie słabo czytelny komunikat:
  %     1 4 2 3
  %     4 3 2 1
  % oznaczający, że:
  %   Andrzej i Sabina byli na koncercie.
  %   Bronek i Ola  byli na wystawie.
  %   Czarek i Paulina byli w kinie.
  %   Darek i Ewa byli w teatrze.
  % Uczyńmy ten komunikat bardziej czytelnym:

  % Koniec części programu rozwiązującej zadanie

  % Poczatek części programu piszącej rozwiązanie.

  % Istota problemu: znaleźć trójki (Numer_imprezy, Chłopiec, Dziewczyna) o tych
  % samych numerach. Numery te odpowiadają uczestnikom i nazwie wspólnej imprezy.

  %    Wyznaczanie numeru chłopca na pierwszej pozycji w liście:
/*16*/   element(Numer_1_chlopca,[Andrzej,Bronek,Czarek,Darek],1),

/*17*/   element(Numer_2_chlopca,[Andrzej,Bronek,Czarek,Darek],2),
/*18*/   element(Numer_3_chlopca,[Andrzej,Bronek,Czarek,Darek],3),
```

```
/*19*/   element(Numer_4_chlopca,[Andrzej,Bronek,Czarek,Darek],4),

  %    Wyznaczanie numeru dziewczyny na pierwszej pozycji w liście:
/*20*/   element(Numer_1_dziewczyny,[Ola, Ewa,Paulina,Sabina],1),

/*21*/   element(Numer_2_dziewczyny,[Ola, Ewa,Paulina,Sabina],2),
/*22*/   element(Numer_3_dziewczyny,[Ola, Ewa,Paulina,Sabina],3),
/*23*/   element(Numer_4_dziewczyny,[Ola, Ewa,Paulina,Sabina],4),

  %    Translacja numeru chłopca na imię chłopca:
/*24*/   wyznacz_imie_chlopca(Numer_1_chlopca,Imie_1_chlopca),
/*25*/   wyznacz_imie_chlopca(Numer_2_chlopca,Imie_2_chlopca),
/*26*/   wyznacz_imie_chlopca(Numer_3_chlopca,Imie_3_chlopca),
/*27*/   wyznacz_imie_chlopca(Numer_4_chlopca,Imie_4_chlopca),

  %    Translacja numeru dziewczyny na imię dziewczyny:
/*28*/ wyznacz_imie_dziewczyny(Numer_1_dziewczyny,Imie_1_dziewczyny),
/*29*/ wyznacz_imie_dziewczyny(Numer_2_dziewczyny,Imie_2_dziewczyny),
/*30*/ wyznacz_imie_dziewczyny(Numer_3_dziewczyny,Imie_3_dziewczyny),
/*31*/ wyznacz_imie_dziewczyny(Numer_4_dziewczyny,Imie_4_dziewczyny),

/*32*/   write(Imie_1_chlopca),write(" i "),
/*33*/   write(Imie_1_dziewczyny), write(" poszli na koncert."),nl,
/*34*/   write(Imie_2_chlopca),write(" i "),
/*35*/   write(Imie_2_dziewczyny), write(" poszli do kina."),nl,
/*36*/   write(Imie_3_chlopca),write(" i "),
/*37*/   write(Imie_3_dziewczyny), write(" poszli do teatru."),nl,
/*38*/   write(Imie_4_chlopca),write(" i "),
/*39*/   write(Imie_4_dziewczyny), write(" poszli na wystawę."),nl.

/*40*/      wyznacz_imie_chlopca(1,"Andrzej").
/*41*/      wyznacz_imie_chlopca(2,"Bronek").
/*42*/      wyznacz_imie_chlopca(3,"Czarek").
/*43*/      wyznacz_imie_chlopca(4,"Darek").

/*44*/      wyznacz_imie_dziewczyny(1,"Ola").
/*45*/      wyznacz_imie_dziewczyny(2,"Ewa").
/*46*/      wyznacz_imie_dziewczyny(3,"Paulina").
/*47*/      wyznacz_imie_dziewczyny(4,"Sabina").
```

Ostatnia część programu generuje komunikat:

```
Andrzej i Sabina poszli na koncert.
Czarek i Paulina poszli do kina.
Darek i Ewa poszli do teatru.
Bronek i Ola poszli na wystawe.
```

4.4.4 Golfiści raz jeszcze

Program z rozdziału 3.7.2 można zmodyfikować korzystając z wprowadzonych predykatów globalnych. Aktualną wersją jest `4_7_golfisci_raz_jeszcze.ecl`:

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/     [Fred,Jurek,Tomek,Robert]::1..4,
        % Tomek - zmienna określająca numer pozycji Tomka w szeregu
/*4*/     alldifferent([Fred,Jurek,Tomek,Robert]),

/*5*/     [Czerwone,Pomaranczowe,Niebieskie,Biale]::1..4,
        % Niebieskie - zmienna określająca numer pozycji koloru
        % niebieskiego w szeregu

        % (1) Jeden ma spodnie czerwone:
/*5*/     alldifferent([Czerwone,Pomaranczowe,Niebieskie,Biale]),

        % (2) Golfista na prawo od Freda ma spodnie niebieskie:
/*6*/     Niebieskie#=Fred+1,

        % (3) Jurek jest na miejscu drugim:
/*7*/     Jurek#=2,

        % (4) Robert ma spodnie białe:
/*8*/     Robert#=Biale,

        % (5) Tomek nie jest na miejscu drugim ani na czwartym
        % i nie ma spodni pomarańczowych:
/*9*/     Tomek#\=2,
/*10*/    Tomek#\=4,
/*11*/    Tomek#\=Pomaranczowe,

/*12*/    labeling([Fred,Jurek,Tomek,Robert,Pomaranczowe,Niebieskie,
                    Czerwone,Biale]),

/*13*/    write("Fred,Jurek,Tomek,Robert"),nl,
/*14*/    write([Fred,Jurek,Tomek,Robert]),nl,
/*15*/    write("Czerwone,Pomaranczowe,Niebieskie,Biale"),nl,
/*16*/    write([Czerwone,Pomaranczowe,Niebieskie,Biale]),nl,

        % Koniec części programu rozwiązującej zadanie
```

```
       % Początek części programu piszącej rozwiązanie.

   % Istota problemu: znaleźć trójki (Numer_pozycji, Imie_golfisty, Kolor_spodni)
   % o tych samych numerach. Odpowiadają one pozycjom golfistow, ich imionom
   % i kolorom ich spodni.

/*17*/    element(Numer_golfisty_1,[Fred,Jurek,Tomek,Robert],1),
/*18*/    element(Numer_golfisty_2,[Fred,Jurek,Tomek,Robert],2),
/*19*/    element(Numer_golfisty_3,[Fred,Jurek,Tomek,Robert],3),
/*20*/    element(Numer_golfisty_4,[Fred,Jurek,Tomek,Robert],4),

/*21*/    imie(Numer_golfisty_1,Imie_golfisty_1),
/*22*/    imie(Numer_golfisty_2,Imie_golfisty_2),
/*23*/    imie(Numer_golfisty_3,Imie_golfisty_3),
/*24*/    imie(Numer_golfisty_4,Imie_golfisty_4),

/*25*/    element(Numer_Koloru_1,[Czerwone,Pomaranczowe,Niebieskie,
                    Biale],1),
/*26*/    element(Numer_Koloru_2,[Czerwone,Pomaranczowe,Niebieskie,
                    Biale],2),
/*27*/    element(Numer_Koloru_3,[Czerwone,Pomaranczowe,Niebieskie,
                    Biale],3),
/*28*/    element(Numer_Koloru_4,[Czerwone,Pomaranczowe,Niebieskie,
                    Biale],4),

/*29*/    kolor(Numer_Koloru_1,Kolor1),
/*30*/    kolor(Numer_Koloru_2,Kolor2),
/*31*/    kolor(Numer_Koloru_3,Kolor3),
/*32*/    kolor(Numer_Koloru_4,Kolor4),

/*33*/    write(Imie_golfisty_1),write(" nosi spodnie "),
                    write(Kolor1),write("."),nl,
/*34*/    write(Imie_golfisty_2),write(" nosi spodnie "),
                    write(Kolor2),write("."),nl,
/*35*/    write(Imie_golfisty_3),write(" nosi spodnie "),
                    write(Kolor3),write("."),nl,
/*36*/    write(Imie_golfisty_4),write(" nosi spodnie "),
                    write(Kolor4),write("."),nl.

/*37*/    imie(1,"Fred").
/*38*/    imie(2,"Jurek").
/*39*/    imie(3,"Tomek").
/*40*/    imie(4,"Robert").

/*41*/    kolor(1,"czerwone").
/*42*/    kolor(2,"pomaranczowe").
/*43*/    kolor(3,"niebieskie").
```

```
/*44*/    kolor(4,"biale").
```

Program generuje następujący komunikat:

```
Fred,Jurek,Tomek,Robert
[1, 2, 3, 4]
Czerwone,Pomaranczowe,Niebieskie,Biale
[3, 1, 2, 4]

Fred nosi spodnie pomaranczowe.
Jurek nosi spodnie niebieskie.
Tomek nosi spodnie czerwone.
Robert nosi spodnie biale.
```

ECL^iPS^e *CLP* jest wyraźnie mocniejszy w logice, aniżeli w generowaniu komunikatów z rozwiązaniem; to ostatnie wymagało więcej linii kodu, aniżeli samo rozwiązanie problemu.

4.4.5 Trzy kule raz jeszcze

Przykład *Trzy kule* z rozdziału 2.4.2 można również uczynić bardziej przejrzystym korzystając z wprowadzonych predykatów globalnych. Zmodyfikowanym programem jest `4_8_trzy_kule_raz_jeszcze.ecl`:

```
/*1*/   :-lib(ic).
/*2*/   top:-
/*3*/     Kolor=[Czarny, Szary, Bialy],
/*4*/     Rozmiar=[Maly, Duzy, Sredni],
  %  Jeżeli Czarny=Sredni=3, tzn. kula czarna ma rozmiar średni i numer 3
/*5*/     [Czarny,Szary, Bialy] :: [1..3],
/*6*/     [Maly, Duzy, Sredni] :: [1..3],

/*7*/     alldifferent([Czarny, Szary, Bialy]),
/*8*/     alldifferent([Maly, Duzy, Sredni]),

/*9*/     Kule = [ kula(1,_,_), kula(2,_,_), kula(3,_,_)],
  %    kula(Numer_kuli,Rozmiar_kuli,Kolor_kuli)

/*10*/    Ograniczenia = [ kula(Czarny,_,czarna),
           kula(Szary,Rozmiar_szarej,szara),
           kula(Bialy,_,biala),

        %(2) Mała kula ma numer 2:
           kula(2,malaa,_),
```

```
          kula(Sredni,sredni,Kolor_sredniej),
          kula(Duzy,duza,Kolor_duzej),
          kula(3,Rozmiar_3,_) ],

       %(1) Duża kula ma jaśniejszy kolor niż kula średnia:
/*11*/  jasniej(Kolor_duzej,Kolor_sredniej),

       %(3) Numer na kuli czarnej jest większy niż numer na kuli białej:
/*12*/  Czarny#>Bialy,

       %(4) Rozmiar kuli o numerze 3 jest mniejszy niż rozmiar kuli szarej:
/*13*/  mniejszy_rozmiar(Rozmiar_3,Rozmiar_szarej),

/*14*/  uziemianie(Ograniczenia, Kule),
       % Elementy listy "Ograniczenia" są uziemieniami elementów listy "Kule".
/*15*/  writeln(Kolor), writeln(Rozmiar),

  % Rozwiązanie tej części ma postać:
  %     [3, 1, 2]                    [2, 1, 3]
  % co odpowiada listom:
  %     "Kolor"               i       "Rozmiar":
  % "[Czarny,Szary,Bialy]"    i   "[Maly, Duzy, Sredni]"

% Koniec części programu rozwiązującej zadanie

% Początek części programu piszącej rozwiązanie

% Istota problemu: znaleźć trójki (Numer_kuli, Kolor, Rozmiar)
%  o tych samych numerach. Odpowiadają one numerom kul, ich kolorom i rozmiarom.

/*16*/  element(Numer_koloru_1,[Czarny,Szary, Bialy],1),
           % numer koloru na pozycji 1, itd.
/*17*/  element(Numer_koloru_2,[Czarny,Szary, Bialy],2),
/*18*/  element(Numer_koloru_3,[Czarny,Szary, Bialy],3),

           % translacja numeru koloru na nazwę koloru:
/*19*/  kolor(Numer_koloru_1,Nazwa_koloru_1),
/*20*/  kolor(Numer_koloru_2,Nazwa_koloru_2),
/*21*/  kolor(Numer_koloru_3,Nazwa_koloru_3),

/*22*/  element(Numer_rozmiaru_1,[Maly,Duzy,Sredni],1),
           %  numer rozmiaru na pozycji 1, itd.
/*23*/  element(Numer_rozmiaru_2,[Maly,Duzy,Sredni],2),
/*24*/  element(Numer_rozmiaru_3,[Maly,Duzy,Sredni],3),
```

```
          % translacja numeru rozmiaru na nazwę rozmiaru:
/*25*/  rozmiar(Numer_rozmiaru_1,Nazwa_rozmiaru_1),
/*26*/  rozmiar(Numer_rozmiaru_2,Nazwa_rozmiaru_2),
/*27*/  rozmiar(Numer_rozmiaru_3,Nazwa_rozmiaru_3),

          % łączenie trzech elementów pierwszej trójki, itd.:
/*28*/  write("Kula "),write(Nazwa_koloru_1),write(" o numerze 1"),
/*29*/  write(" jest "),write(Nazwa_rozmiaru_1),nl,

/*30*/  write("Kula "),write(Nazwa_koloru_2),write(" o numerze 2"),
/*31*/  write(" jest "),write(Nazwa_rozmiaru_2),nl,

/*32*/  write("Kula "),write(Nazwa_koloru_3),write(" o numerze 3"),
/*33*/  write(" jest "),write(Nazwa_rozmiaru_3),nl.

/*34*/  kolor(1,czarna).
/*35*/  kolor(2,szara).
/*36*/  kolor(3,biala).

/*37*/  rozmiar(1,mala).
/*38*/  rozmiar(2,duza).
/*39*/  rozmiar(3,srednia).

/*40*/  mniejszy_rozmiar(mala,duza).
/*41*/  mniejszy_rozmiar(mala,srednia).
/*42*/  mniejszy_rozmiar(srednia,duza).

/*43*/  jasniej(biala,szara).
/*44*/  jasniej(biala,czarna).
/*45*/  jasniej(szara,czarna).

/*46*/  uziemianie([],_).
/*47*/  uziemianie([H|T],Lista):-
/*48*/      member(H,Lista),
/*49*/      uziemianie(T,Lista).
```

Rozwiązanie ma postać:

```
Kula szara o numerze 1 jest duza
Kula biala o numerze 2 jest mala
Kula czarna o numerze 3 jest srednia
```

4.4.6 Hetmani raz jeszcze

Wprowadzone predykaty globalne umożliwiają zastosowanie dosyć oryginalnego podejścia do rozwiązania problemu 8 hetmanów. Przedstawia je program `4_9_hetmani_raz_jeszcze.ecl`:

```
/*1*/   :-lib(ic).

/*2*/   top:-
/*3*/     hetmani(_).

/*4*/   hetmani([X1,X2,X3,X4,X5,X6,X7,X8]):-
/*5*/     [X1,X2,X3,X4,X5,X6,X7,X8]::1..8,
/*6*/     [X11,X22,X33,X44,X55,X66,X77,X88]::1..16,
/*7*/     [X18,X27,X36,X45,X54,X63,X72,X81]::1..16,
/*8*/     alldifferent([X1,X2,X3,X4,X5,X6,X7,X8]),
/*9*/     X11 #= X1+1,
/*10*/    X22 #= X2+2,
/*11*/    X33 #= X3+3,
/*12*/    X44 #= X4+4,
/*13*/    X55 #= X5+5,
/*14*/    X66 #= X6+6,
/*15*/    X77 #= X7+7,
/*16*/    X88 #= X8+8,
/*17*/    alldifferent([X11,X22,X33,X44,X55,X66,X77,X88]),

/*18*/    X18 #= X1+8,
/*19*/    X27 #= X2+7,
/*20*/    X36 #= X3+6,
/*21*/    X45 #= X4+5,
/*22*/    X54 #= X5+4,
/*23*/    X63 #= X6+3,
/*24*/    X72 #= X7+2,
/*25*/    X81 #= X8+1,
/*26*/    alldifferent([X18,X27,X36,X45,X54,X63,X72,X81]),

/*27*/    labeling([X1,X2,X3,X4,X5,X6,X7,X8]),
/*28*/    write([X1,X2,X3,X4,X5,X6,X7,X8]),nl,fail.

/*29*/    hetmani(_):-
/*30*/    write("To wszystko!").
```

Rozwiązanie ma postać identyczną jak dla przykładu `3_11_hetmani.ecl`: z pośród 92 możliwych rozwiązań przedstawimy tylko dwa pierwsze i dwa ostatnie:

```
[1, 5, 8, 6, 3, 7, 2, 4]
```

```
[1, 6, 8, 3, 7, 4, 2, 5]
.......................
[8, 3, 1, 6, 2, 5, 7, 4]
[8, 4, 1, 3, 6, 2, 7, 5]
To wszystko!
```

4.4.7 Siedem maszyn - siedem operacji

Dzielenie resursów pomiędzy operacje jest typową kombinatoryczną aplikacją, rozwiązywaną za pomocą języków *CLP*. Ilustruje to następujący prosty przykład:

Każda z siedmiu maszyn może wykonać dowolną z siedmiu operacji, jednak z różnym kosztem, co pokazano w tablicy 4.1.

Maszyny	Operacje						
	1	2	3	4	5	6	7
1	15	23	43	27	76	43	91
2	45	76	32	39	72	37	48
3	56	45	87	75	34	76	29
4	13	45	34	51	52	21	76
5	45	49	18	48	58	98	23
6	23	25	29	39	52	41	12
7	76	98	86	41	34	76	77

Tablica 4.1: Koszty operacji dla siedmiu maszyn

Należy dokonać rozdziału operacji pomiędzy maszyny tak, by całkowity koszt wykonania wszystkich operacji był mniejszy od 185. Można to zrobić za pomocą programu `4_10_7_maszyn_7_operacji.ecl`

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/     [O1,O2,O3,O4,O5,O6,O7]::1..7,
  % Np.: O2=4 oznacza, że maszyna 2 wykonuje operację 4
/*4*/     [K1,K2,K3,K4,K5,K6,K7]::0..100,
  % Np.: K2=39 oznacza, że maszyna 2 wykonuje operację z kosztem 39

/*5*/     alldifferent([O1,O2,O3,O4,O5,O6,O7]),
/*6*/     element(O1,[15,23,43,27,76,43,91],K1),
```

```
/*7*/     element(O2,[45,76,32,39,72,37,48],K2),
/*8*/     element(O3,[56,45,87,75,34,76,29],K3),
/*9*/     element(O4,[13,45,34,51,52,21,76],K4),
/*10*/    element(O5,[45,49,18,48,58,98,23],K5),
/*11*/    element(O6,[23,25,29,39,52,41,12],K6),
/*12*/    element(O7,[76,98,86,41,34,76,77],K7),

/*13*/    K1+K2+K3+K4+K5+K6+K7 #< 185,
/*14*/    labeling([K1,K2,K3,K4,K5,K6,K7]),
/*15*/    przedstaw_wyniki([O1,K1,O2,K2,O3,K3,O4,K4,
                  O5,K5,O6,K6,O7,K7],1),
/*16*/    K is K1+K2+K3+K4+K5+K6+K7,
/*17*/    write("Koszt = "),write(K),
/*18*/    L=[O1,K1,O2,K2,O3,K3,O4,K4,O5,K5,O6,K6,O7,K7],
/*19*/    write(L).

/*20*/  przedstaw_wyniki([],_):-
/*21*/    !.
/*22*/  przedstaw_wyniki([A,B|R],N):-
/*23*/    write("Maszyna "),write(N),write(" wykonuje operacje "),
            write(A),write(" przy koszcie "),write(B),write("."),nl,
/*24*/    M is N+1,
/*25*/    przedstaw_wyniki(R,M).
```

Tym razem zażądano wyświetlenia wszystkich rozwiązań za pomocą opcji `more` w menu głównym ECL^iPS^e, co daje:

```
Maszyna 1 wykonuje operacje 1 przy koszcie 15.
Maszyna 2 wykonuje operacje 4 przy koszcie 39.
Maszyna 3 wykonuje operacje 7 przy koszcie 29.
Maszyna 4 wykonuje operacje 6 przy koszcie 21.
Maszyna 5 wykonuje operacje 3 przy koszcie 18.
Maszyna 6 wykonuje operacje 2 przy koszcie 25.
Maszyna 7 wykonuje operacje 5 przy koszcie 34.
Koszt = 181
[O1,K1,O2,K2,O3,K3,O4,K4,O5,K5,O6,K6,O7,K7] =
[1, 15, 4, 39, 7, 29, 6, 21, 3, 18, 2, 25, 5, 34]

Maszyna 1 wykonuje operacje 1 przy koszcie 15.
Maszyna 2 wykonuje operacje 4 przy koszcie 39.
Maszyna 3 wykonuje operacje 2 przy koszcie 45.
Maszyna 4 wykonuje operacje 6 przy koszcie 21.
Maszyna 5 wykonuje operacje 3 przy koszcie 18.
Maszyna 6 wykonuje operacje 7 przy koszcie 12.
Maszyna 7 wykonuje operacje 5 przy koszcie 34.
Koszt = 184
```

```
[O1,K1,O2,K2,O3,K3,O4,K4,O5,K5,O6,K6,O7,K7] =
[1, 15, 4, 39, 2, 45, 6, 21, 3, 18, 7, 12, 5, 34]

Maszyna 1 wykonuje operacje 2 przy koszcie 23.
Maszyna 2 wykonuje operacje 6 przy koszcie 37.
Maszyna 3 wykonuje operacje 5 przy koszcie 34.
Maszyna 4 wykonuje operacje 1 przy koszcie 13.
Maszyna 5 wykonuje operacje 3 przy koszcie 18.
Maszyna 6 wykonuje operacje 7 przy koszcie 12.
Maszyna 7 wykonuje operacje 4 przy koszcie 41.
Koszt = 178
[O1,K1,O2,K2,O3,K3,O4,K4,O5,K5,O6,K6,O7,K7] =
[2, 23, 6, 37, 5, 34, 1, 13, 3, 18, 7, 12, 4, 41]

Maszyna 1 wykonuje operacje 4 przy koszcie 27.
Maszyna 2 wykonuje operacje 6 przy koszcie 37.
Maszyna 3 wykonuje operacje 7 przy koszcie 29.
Maszyna 4 wykonuje operacje 1 przy koszcie 13.
Maszyna 5 wykonuje operacje 3 przy koszcie 18.
Maszyna 6 wykonuje operacje 2 przy koszcie 25.
Maszyna 7 wykonuje operacje 5 przy koszcie 34.
Koszt = 183
[O1,K1,O2,K2,O3,K3,O4,K4,O5,K5,O6,K6,O7,K7] =
[4, 27, 6, 37, 7, 29, 1, 13, 3, 18, 2, 25, 5, 34]
```

4.4.8 Trzy maszyny - trzy z pięciu operacji

Nieco bardziej złożonym przykładem dzielenia resursów pomiędzy operacje jest następujący:

Każda z trzech maszyn może wykonać dowolną z pięciu operacji, jednak z różnym kosztem, co pokazano w tablicy 4.2.

Maszyny	Operacje				
	1	2	3	4	5
1	1	11	5	7	13
2	4	6	2	8	10
3	6	3	9	12	15

Tablica 4.2: Koszty operacji dla trzech maszyn

Należy dokonać rozdziału trzech z pięciu operacji pomiędzy maszyny tak, by całkowity koszt wykonania wszystkich operacji był mniejszy od kosztu progowego 10, a operacje były różne. Rozwiązanie tego zadania przedstawia program `4_11_3_maszyny_3_z_5_operacji.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/     [O1,O2,O3] :: 1..5,
% Lista numerów operacji zawiera numery trzech z pośród pięciu operacji.
% Np. O2 jest numerem operacji wykonywanej przez maszynę 2.

/*4*/     [K1,K2,K3] :: 1..10,
% Lista kosztów wybranych trzech z pośród pięciu operacji.
% Np. K2 jest kosztem wykonania operacji o numerze O2.

/*5*/     alldifferent([O1,O2,O3]),
/*6*/     element(O1,[1,11,5,7,13],K1),
% [1,11,5,7,13] - lista kosztów operacji wykonywanych przez maszynę 1.
/*7*/     element(O2,[4,6,2,8,10],K2),
/*8*/     element(O3,[6,3,9,12,15],K3),

/*9*/     K1+K2+K3 #=< 10,
/*10*/    labeling([K1,K2,K3]),
/*11*/    przedstaw_wyniki([O1,K1,O2,K2,O3,K3],1).

/*12*/    przedstaw_wyniki([],_).
/*13*/    przedstaw_wyniki([A,B|R],N):-
/*14*/       write("Maszyna "),write(N),write(" wykonuje operacje "),
/*15*/       write(A), write(" przy koszcie "),write(B),write(.),nl,
/*15*/       M is N+1,
/*16*/       przedstaw_wyniki(R,M).
```

Program generuje komunikat:

```
Maszyna 1 wykonuje operacje 1 przy koszcie 1.
Maszyna 2 wykonuje operacje 3 przy koszcie 2.
Maszyna 3 wykonuje operacje 2 przy koszcie 3.
```

przedstawiające jedno z możliwych siedmiu rozwiązań. Pozostałe można wyświetlić korzystając z opcji *more*.

4.4.9 Trzy maszyny - pięć operacji

Kolejną komplikację wprowadza następujący przykład:

Każda z trzech maszyn może wykonać dowolną z pięciu operacji, jednak z różnym kosztem, co pokazano w tablicy 4.3.

Maszyny	Operacje					
	1	2	3	4	5	6
M1	1	11	5	7	13	0
M12	1	11	5	7	13	0
M2	4	6	2	8	10	0
M22	4	6	2	8	10	0
M3	6	3	9	12	15	0
M32	6	3	9	12	15	0

Tablica 4.3: Koszty operacji dla maszyn i ich dubletów

Należy dokonać rozdziału pięciu operacji pomiędzy trzy maszyny tak, by całkowity koszt wykonania wszystkich operacji był mniejszy od pewnego kosztu progowego = 25, a operacje były różne. Dla uzyskania rozwiązania wprowadza się każdą maszynę dwa razy i wprowadza wszędzie pozorną operację (ostatnią na liście) o koszcie 0, co pokazano w tablicy 4.3. W ten sposób otrzymuje się rozwiązywany już rozdział z taką samą liczbą operacji co maszyn.

Rozwiązanie to generuje program `4_12_3_maszyny_5_operacji.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/     [M1,M2,M3,M12,M22,M32] :: 1..6,
/*4*/     [K1,K2,K3,K12,K22,K32] :: 0..16,
/*5*/     alldifferent([M1,M2,M3,M12,M22,M32]),

/*6*/     element(M1,[1,11,5,7,13,0],K1),
/*7*/     element(M12,[1,11,5,7,13,0],K12),
/*8*/     element(M2,[4,6,2,8,10,0],K2),
/*9*/     element(M22,[4,6,2,8,10,0],K22),
/*10*/    element(M3,[6,3,9,12,15,0],K3),
/*11*/    element(M32,[6,3,9,12,15,0],K32),

/*12*/    K1+K2+K3+K12+K22+K32 #=< 25,
/*13*/    labeling([K1,K2,K3,K12,K22,K32]),
/*14*/    L=[M1,K1,M2,K2,M3,K3,M12,K12,M22,K22,M32,K32],
```

```
/*15*/    write(L),nl,
/*16*/    przedstaw_wyniki(L,1),nl.

/*17*/  przedstaw_wyniki([],_):-
/*18*/    !.
/*19*/  przedstaw_wyniki([A,B|R],N):-
/*20*/    not(B = 0),
/*21*/    N =< 3,
/*22*/    !,
/*23*/    write("Maszyna "),write(N),write(" wykonuje operacje "),
/*24*/    write(A), write(" przy koszcie "),write(B),write(.),nl,
/*25*/    M is N+1,
/*26*/    przedstaw_wyniki(R,M).
/*27*/  przedstaw_wyniki([A,B|R],N):-
/*28*/    not(B = 0),
/*29*/    N > 3,
/*30*/    M is N - 3,
/*31*/    write("Maszyna "),write(M),write(" wykonuje operacje "),
/*32*/    write(A), write(" przy koszcie "),write(B),write(.),nl,
/*33*/    M is N+1,
/*34*/    przedstaw_wyniki(R,M).
/*35*/  przedstaw_wyniki([_,_|R],N):-
/*36*/    M is N+1,
/*37*/    !,
/*38*/    przedstaw_wyniki(R,M).
```

Program generuje komunikat:

```
[1, 1, 3, 2, 6, 0, 4, 7, 5, 10, 2, 3]
Maszyna 1 wykonuje operacje 1 przy koszcie 1.
Maszyna 2 wykonuje operacje 3 przy koszcie 2.
Maszyna 1 wykonuje operacje 4 przy koszcie 7.
Maszyna 2 wykonuje operacje 5 przy koszcie 10.
Maszyna 3 wykonuje operacje 2 przy koszcie 3.
```

4.5 Kombinatoryczne rozkładowanie

4.5.1 Pięć sal

Poniższy przykład jest odmianą znanej łamigłówki *Zebra* (należącej do folkloru *Prolog*u i *CLP*), której autorstwo jest przypisywane Lewisowi Carrollowi, autorowi "Alicji w Krainie Czarów":

Należy przyporządkować pięciu salom 5 kolorów, 5 dni tygodnia, 5 przedmiotów, 5 ocen przedmiotu i 5 technik prowadzenia przedmiotu. Na:

```
 - kolory sal (czerwona, zielona, niebieska, biała, żółta),
 - dni tygodnia (poniedziałek, wtorek, środa, czwartek, piątek),
 - przedmioty (fizyka, matematyka, informatyka, ekonomia, angielski),
 - oceny przedmiotów (nudne, trudne, ciekawe, bardzo_ciekawe,
             takie_sobie),
 - techniki prowadzenia (komputer, internet, magnetowid,
             kreda_tablica, wideoprojektor)
```

nakłada się następujące ograniczenia:

```
  (1)  Fizyka jest w czerwonej sali.
  (2)  Angielski jest prowadzony z wykorzystaniem magnetowidu.
  (3)  Matematyka jest w pierwszej sali z lewej strony.
  (4)  Zajęcia są nudne w sali żółtej.
  (5)  Zajęcia są ciekawe w sali obok sali, w której jest komputer.
  (6)  Matematyka jest w sali obok sali niebieskiej.
  (7)  Zajęcia takie_sobie są prowadzone za pomocą kredy i tablicy.
  (8)  Zajęcia bardzo_ciekawe są w czwartek.
  (9)  Informatyka jest ww wtorek.
  (10) Zajęcia z ekonomii są trudne.
  (11) Zajęcia są nudne w sali obok sali, gdzie jest internet.
  (12) W sali zielonej są zajęcia w piątek.
  (13) Sala zielona jest bezpośrednio na prawo od sali białej.
  (14) W sali środkowej są zajęcia w środę.

Należy wyznaczyć przyporządkowania kompletne:
1)Jakie zajęcia, w jakich salach, w jakie dni tygodnia,przy
  użyciu jakich technik i przy jakich ocenach są prowadzone?

Należy wyznaczyć  tylko przyporządkowania cząstkowe:
2a)Jakie zajęcia i w jakiej sali są  w poniedziałek?
```

2b) Jakie zajęcia i w jakiej sali są prowadzone za pomocą wideoprojektora?

Możliwym rozwiązaniem jest jeden z $(5!)^5 =\simeq 2.5 * 10^{10}$ sposobów rozdziału. Gdyby problem próbować rozwiązywać metodą przeglądu zupełnego, to przyjmując że generowanie i sprawdzenie pojedynczego sposobu rozdziału zajmuje 0.01 sek, rozwiązanie można by w najbardziej niekorzystnym przypadku wyznaczyć po 8 latach obliczeń. Odpowiedni program (`4_13_sale_5.ecl`) ma postać:

```
/*1*/   :-lib(ic).
/*2*/   top:-
/*3*/     Dni = [Poniedzialek,Wtorek,Sroda,Czwartek,Piatek],
/*4*/     Kolory = [Czerwona,Zielona,Niebieska,Biala,Zolta],
/*5*/     Przedmioty = [Fizyka,Matematyka,Informatyka,Ekonomia,
                          Angielski],
/*6*/     Oceny = [Nudne,Trudne,Ciekawe,BardzoCiekawe,TakieSobie],
/*7*/     Techniki = [Komputer,Internet,Magnetowid,KredaTablica,
                          Wideoprojektor],

/*8*/     Dni :: 1..5,
/*9*/     Kolory :: 1..5,
/*10*/    Przedmioty :: 1..5,
/*11*/    Oceny :: 1..5,
/*12*/    Techniki :: 1..5,

/*13*/    alldifferent(Dni),
/*14*/    alldifferent(Kolory),
/*15*/    alldifferent(Przedmioty),
/*16*/    alldifferent(Oceny),
/*17*/    alldifferent(Techniki),

%(1)  Fizyka jest w czerwonej sali:
/*18*/    Fizyka#=Czerwona,
%(2)  Angielski jest prowadzony z wykorzystaniem magnetowidu:
/*19*/    Angielski#=Magnetowid,
%(3)  Matematyka jest w pierwszej sali z lewej strony:
/*20*/    Matematyka is 1,
%(4)  Zajęcia są nudne w sali żółtej:
/*21*/    Nudne#=Zolta,
%(5)  Zajęcia są ciekawe w sali obok sali, w której stosowany jest
%     komputer:
/*22*/    obok(Ciekawe,Komputer,1),
%(6)  Matematyka jest w sali obok sali niebieskiej:
/*23*/    obok(Matematyka,Niebieska,1),
%(7)  Zajęcia takie sobie są prowadzone za pomocą kredy i tablicy:
/*24*/    TakieSobie#=KredaTablica,
```

```
%(8)  Zajęcia bardzo ciekawe są w czwartek:
/*25*/    BardzoCiekawe#=Czwartek,
%(9)  Informatyka jest w wtorek:
/*26*/    Informatyka#=Wtorek,
%(10) Zajęcia z ekonomii są trudne:
/*27*/    Ekonomia#=Trudne,
%(11) Zajęcia są nudne w sali obok sali gdzie jest Internet:
/*28*/    obok(Nudne,Internet,1),
%(12) W sali zielonej są zajęcia w piątek:
/*29*/    Zielona#=Piatek,
%(13) Sala zielona jest bezpośrednio na prawo od sali białej:
/*30*/    Zielona#=Biala+1,
%(14) W sali środkowej są zajęcia w środę:
/*31*/    Sroda#=3,


/*32*/    flatten([Dni, Kolory, Przedmioty,Oceny,Techniki], Lista),
/*33*/    labeling(Lista),nl,nl,


/*34*/write("Przyporzadkowania kompletne:"),nl,
/*35*/    write("Dni =        "),write(Dni),nl,
/*36*/    write("Kolory =     "), write(Kolory),nl,
/*37*/    write("Przedmioty = "),write(Przedmioty),nl,
/*38*/    write("Oceny =      "),write(Oceny),nl,
/*39*/    write("Techniki =   "),write(Techniki),nl,nl,


/*40*/write("Przyporzadkowania czesciowe:"),nl,
/*41*/    PrzedmiotyNames = [Fizyka-fizyka, Matematyka-matematyka,
              Informatyka-informatyka, Ekonomia-ekonomia,
                    Angielski-angielski],
/*42*/    memberchk(Poniedzialek-PoniedzialekDni, PrzedmiotyNames),
/*43*/    memberchk(Wideoprojektor-WideoprojektorTechniki,
                PrzedmiotyNames),
/*44*/    printf("%w jest w poniedzialek.\nl", [PoniedzialekDni]),
/*45*/    printf("%w jest prowadzona za pomocą wideoprojektora.",
                [WideoprojektorTechniki]).


/*46*/   obok(X,Y,Z):-
/*47*/     X+Z#=Y.


/*48*/   obok(X,Y,Z):-
/*49*/     X#=Y+Z.
```

Wyniki, ich graficzna interpretacja wraz z zastosowanymi listami pokazano na rysunku 4.1.

dni tygodnia(poniedziałek, wtorek, środa, czwartek, piątek),
kolory sal(czerwona, zielona, niebieska, biała, żółta),
przedmioty(fizyka, matematyka, informatyka, ekonomia, j_angielski),
oceny przedmiotów(nudne, trudne, ciekawe, bardzo_ciekawe, takie_sobie),
technika prowadzenia przedmiotów(komputer, internet, magnetowid, kreda_tablica, projektor)

Rozwiązanie:
Dni = [1, 2, 3, 4, 5]
Kolory = [3, 5, 2, 4, 1]
Przedmioty = [3, 1, 2, 5, 4]
Oceny = [1, 5, 2, 4, 3]
Techniki = [1, 2, 4, 3, 5]

poniedziałek	wtorek	środa	czwartek	piątek
żółta	niebieska	czerwona	biała	zielona
matematyka	informatyka	fizyka	angielski	ekonomia
nudne	ciekawe	takie_sobie	bardzo_ciekawe	trudne
komputer	internet	kreda_tablica	magnetowid	projektor

Rysunek 4.1: Wyniki dla 5 sal

4.5.2 Dziesięć sal

Powiększmy rozmiary problemu pięciu sal (patrz 4.5.1) do 10 sal: należy przyporządkować 10 salom 10 kolorów, 10 dni tygodnia,10 przedmiotów, 10 ocen przedmiotu i 10 technik prowadzenia przedmiotu. Na:

```
- kolory sal(czerwona, zielona, niebieska, biała, żółta,
          różowa,fioletowa,pomarańczowa,brązowa,szara),
- dni tygodnia-do południa i po południu(pn 8-12,wt 8-12,sr 8-12,
          czw 8-12,pt 8-12,pn 12-16,wt 12-16,sr 12-16,czw 12-16,
          pt 12-16),
- przedmioty(fizyka,matematyka,informatyka,ekonomia,j_angielski,
          chemia,niemiecki,historia,muzyka,elektronika),
- oceny przedmiotów(nudne,trudne,ciekawe,bardzo_ciekawe,takie_sobie,
         wyczerpujące,zabawne,b_popularne,rozśpiewane,absorbujące),
- technika prowadzenia przedmiotów(komputer,Internet,magnetowid,
         kreda_tablica,projektor,odczynniki,słowniki,mapa,fortepian,
         oscyloskop)
```

nakłada się następujące ograniczenia:

```
(1)  Fizyka jest w czerwonej sali.
```

```
  (2)  Angielski jest prowadzony z wykorzystaniem magnetowidu.
  (3)  Matematyka jest  w pierwszej sali z lewej strony.
  (4)  Zajęcia są nudne w sali żółtej.
  (5)  Zajęcia są ciekawe w sali obok sali, w której stosowany jest
            komputer.
  (6)  Matematyka jest w sali obok sali niebieskiej.
  (7)  Zajęcia takie_sobie są prowadzone za pomocą kredy_tablicy.
  (8)  Zajęcia bardzo_ciekawe są w czwartek.
  (9)  Informatyka jest w wtorek.
  (10) Zajęcia z ekonomii są trudne.
  (11) Zajęcia są nudne w sali obok sali gdzie jest sieć komputerowa.
  (12) W sali zielonej są zajęcia w piątek.
  (13) Sala zielona jest bezpośrednio na prawo od sali białej.
  (14) W sali środkowej są zajęcia w środę.
  (15) Po południu w różowej sali są zajęcia w poniedziałek.
  (16) Na chemii używa się odczynniki.
  (17) Zajęcia w poniedziałek po południu sa wyczerpujące.
  (18) Projektor jest w sali obok sali z odczynnikami.
  (19) Popołudniu po weekendzie  wolna jest tylko sala różowa.
  (20) W fioletowej sali sa słowniki.
  (21) Fioletowa sala jest obok różowej.
  (22) Niemiecki jest w sali obok sali, w której jest chemia.
  (23) We wtorek po południu zajęcia są zabawne.
  (24) Sala ze słownikami jest na lewo od sali pomarańczowej.
  (25) Zajęcia po niemieckim są bardzo popularne.
  (26) W środę po południu zajęcia są w sali pomarańczowej.
  (27) Zajęcia z historii wymagają map.
  (28) Fortepian jest w sali 9
  (29) Zajęcia z muzyki sa rozśpiewane.
  (30) Fortepian jest w brązowej sali.
  (31) Brązowa sale używana jest w czwartek po południu.
  (32) Na zajęciach z elektroniki słychać muzykę zza ścian.
  (33) Elektronika odbywa się z użyciem oscyloskopu
  (34) Zajęcia z oscyloskopem sa absorbujące.
  (35) W piątek po południu są zajęcia w szarej sali.

Należy wyznaczyć przyporządkowania kompletne:
1)Jakie zajęcia, w jakich salach, w jakie dni tygodnia,przy
  użyciu jakich technik i przy jakich ocenach są prowadzone?
```

Należy wyznaczyć tylko przyporządkowania cząstkowe:
2a) Jakie zajęcia i w jakiej sali są prowadzone w poniedziałek?
2b) Jakie zajęcia i w jakiej sali są prowadzone za pomocą wideoprojektora?

Wyznacza to program `4_14_sale_10.ecl`:

```
/*1*/   :-lib(ic).
/*2*/   top:-
/*3*/     Dni = [Poniedzialek_przed,Wtorek_przed,Sroda_przed,
                 Czwartek_przed,Piatek_przed,Poniedzialek_po,
                 Wtorek_po,Sroda_po,Czwartek_po,Piatek_po]
/*4*/     Kolory = [Czerwona, Zielona, Niebieska, Biala, Zolta,
              Rozowa,Fioletowa,Pomaranczowa,Brazowa,Szara],
/*5*/     Przedmioty = [Fizyka,Matematyka,Informatyka,Ekonomia,
              Angielski,Chemia,Niemiecki,Historia,Muzyka,
              Elektronika],

/*6*/     Oceny = [Nudne,Trudne, Ciekawe,BardzoCiekawe,
              TakieSobie,Wyczerpujace,Zabawne,B_popularne,
              Rozspiewane,Absorbujace],
/*7*/     Techniki = [Komputer,Internet,Magnetowid,KredaTablica,
              Wideoprojektor,Odczynniki,Slowniki,Mapy,Fortepian,
              Oscyloskop],

/*8*/     Dni :: 1..10,
/*9*/     Kolory :: 1..10,
/*10*/    Przedmioty :: 1..10,
/*11*/    Oceny :: 1..10,
/*12*/    Techniki :: 1..10,

/*13*/    alldifferent(Dni),
/*14*/    alldifferent(Kolory),
/*15*/    alldifferent(Przedmioty),
/*16*/    alldifferent(Oceny),
/*17*/    alldifferent(Techniki),

%(1)  Fizyka jest prowadzona w czerwonej sali:
/*18*/    Fizyka#=Czerwona,
%(2)  Angielski jest prowadzony z wykorzystaniem magnetowidu:
/*19*/    Angielski#=Magnetowid,
%(3)  Matematyka jest prowadzona w pierwszej sali z lewej strony:
/*20*/    Matematyka is 1,
%(4)  Zajęcia są nudne w sali żółtej:
/*21*/    Nudne#=Zolta,
%(5)  Zajęcia są ciekawe w sali obok sali, w ktorej stosowany
```

```
%     jest komputer:
/*22*/    obok(Ciekawe,Komputer,1),
%(6)  Matematyka odbywa się w sali obok sali niebieskiej:
/*23*/    obok(Matematyka,Niebieska,1),
%(7)  Zajęcia takie_sobie są prowadzone za pomocą kredy i tablicy:
/*24*/    TakieSobie#=KredaTablica,
%(8)  Zajęcia bardzo_ciekawe są w czwartek przed południem:
/*25*/    BardzoCiekawe#=Czwartek_przed,
%(9)  Informatyka jest w wtorek przed południem:
/*26*/    Informatyka#=Wtorek_przed,
%(10) Zajęcia z ekonomii sa trudne:
/*27*/    Ekonomia#=Trudne,
%(11) Zajęcia są nudne w sali obok sali gdzie jest Internet:
/*28*/    obok(Nudne,Internet,1),
%(12) W sali zielonej odbywają się zajęcia w piątek przed południem:
/*29*/    Zielona#=Piatek_przed,
%(13) Sala zielona jest bezposrednio na prawo od sali białej:
/*30*/    Zielona#=Biala+1,
%(14) W sali środkowej odbywają się zajęcia w środę przed południem:
/*31*/    Sroda_przed#=3,
%(15) Po południu w różowej sali są zajęcia w poniedziałek:
/*32*/    Rozowa#=Poniedzialek_po,
%(16) Na chemii używa się odczynników:
/*33*/    Chemia#=Odczynniki,
%(17) Zajęcia w poniedziałek po południu są wyczerpujące:
/*34*/    Poniedzialek_po#=Wyczerpujace,
%(18) Wideoprojektor jest w sali obok sali z odczynnikami:
/*35*/    obok(Wideoprojektor,Odczynniki,1),
%(19) Po weekendzie popołudniu wolna jest tylko sala różowa:
/*36*/    Poniedzialek_po#=Rozowa,
%(20) W fioletowej sali są słowniki:
/*37*/    Fioletowa#=Slowniki,
%(21) Fioletowa sala jest obok różowej:
/*38*/    obok(Fioletowa,Rozowa,1),
%(22) Niemiecki  jest w sali obok sali w której jest chemia:
/*39*/    obok(Niemiecki,Chemia,1),
%(23) We wtorek popołudniu zajęcia są zabawne:
/*40*/    Wtorek_po#=Zabawne,
%(24) Sala ze słownikami jest na lewo od sali pomarańczowej:
/*41*/    Pomaranczowa#=Slowniki+1,
%(25) Zajęcia następujące po niemieckim są bardzo popularne:
/*42*/    B_popularne#=Niemiecki+1,
%(26) W środę po południu zajęcia są w sali pomarańczowej:
/*43*/    Sroda_po#=Pomaranczowa,
%(27) Zajęcia z historii wymagają map:
/*44*/    Historia#=Mapy,
%(28) Fortepian jest w 9 sali:
/*45*/    Fortepian is 9,
%(29) Zajęcia z muzyki są rozśpiewane:
```

```
/*46*/    Muzyka#=Rozspiewane,
%(30) Fortepian jest w brązowej sali:
/*47*/    Fortepian#=Brazowa,
%(31) Brązowa sala używana jest w czwartek po południu:
/*48*/    Brazowa#=Czwartek_po,
%(32) Elektronika odbywa się obok sali, w ktorej uczy się muzyki:
/*49*/    obok(Elektronika,Muzyka,1),
%(33) Elektronika odbywa się z użyciem oscyloskopu:
/*50*/    Elektronika#=Oscyloskop,
%(34) Zajęcia z oscyloskopem są absorbujące:
/*51*/    Oscyloskop#=Absorbujace,
%(35) W piątek po południu są zajęcia w szarej sali:
/*52*/    Piatek_po#=Szara,

/*53*/    flatten([Dni, Kolory, Przedmioty, Oceny, Techniki], Lista),
/*54*/    labeling(Lista),

/*55*/write("Przyporzadkowania kompletne:"),nl,
/*56*/    write("Dni =        "),write(Dni),nl,
/*57*/    write("Kolory =     "), write(Kolory),nl,
/*58*/    write("Przedmioty = "),write(Przedmioty),nl,
/*59*/    write("Oceny =      "),write(Oceny),nl,
/*60*/    write("Techniki =   "),write(Techniki),nl,nl,

/*61*/write("Przyporzadkowania czesciowe:"),nl,
/*62*/    PrzedmiotyNames = [Fizyka-fizyka, Matematyka-matematyka,
            Informatyka-informatyka, Ekonomia-ekonomia,
            Angielski-angielski,Chemia-chemia,Niemiecki-niemiecki,
            Historia-historia,Muzyka-muzyka,Elektronika-elektronika],
/*63*/    memberchk(PoniedzialekDni,
            PrzedmiotyNames),
/*64*/    memberchk(Wideoprojektor-WideoprojektorTechniki,
            PrzedmiotyNames),
/*65*/    printf("W poniedzialek jest %w.\n", [PoniedzialekDni]),
/*66*/    printf("Za pomoca wideoprojektora jest prowadzona %w.\n",
            [WideoprojektorTechniki]).

/*67*/  obok(X,Y,Z):-
/*68*/    X+Z#=Y.

/*69*/  obok(X,Y,Z):-
/*70*/    X#=Y+Z.
```

Tym razem mamy cztery rozwiązania spełniające ograniczenia. Pokazano je na rysunkach 4.2 i 4.3.

kolory sal(czerwona, zielona, niebieska, biała, żółta,
różowa, fioletowa, pomarańczowa, brązowa, szara),
dni tygodnia-do południa i po południu(pn 8-12,wt 8-12,sr 8-12,czw 8-12,pt 8-12,
pn 12-16,wt 12-16,sr 12-16,czw 12-16,pt 12-16),
przedmioty(fizyka,matematyka,informatyka,ekonomia,j_angielski,
chemia,niemiecki,historia,muzyka,elektronika),
oceny przedmiotów(nudne, trudne, ciekawe, bardzo_ciekawe, takie_sobie,
wyczerpujące, zabawne, b_popularne, rozspiewane, absorbujace),
technika prowadzenia przedmiotów(komputer, internet, magnetowid, kreda_tablica, projektor,
odczynniki, slowniki, mapa, fortepian, oscyloskop)

Rozwiązanie 1:
Dni = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Kolory = [3, 5, 2, 4, 1, 6, 7, 8, 9, 10]
Przedmioty = [3, 1, 2, 5, 4, 6, 7, 8, 9, 10]
Oceny = [1, 5, 2, 4, 3, 6, 7, 8, 9, 10]
Techniki = [1, 2, 4, 3, 5, 6, 7, 8, 9, 10]

pn 8-12	wt 8-12	śr 8-12	czw 8-12	pt 8-12
żółta	niebieska	czerwona	biała	zielona
matematyka	informatyka	fizyka	angielski	ekonomia
nudne	ciekawe	takie_sobie	bardzo_ciekawe	trudne
komputer	internet	kreda_tablica	magnetowid	projektor

pn 12-16	wt 12-16	śr 12-16	czw 12-16	pt 12-16
różowa	fioletowa	pomarańczowa	brązowa	szara
chemia	niemiecki	historia	muzyka	elektronika
wyczerpujące	zabawne	b.popularne	rozśpiewane	absorbujące
odczynniki	słowniki	mapa	fortepian	oscyloskop

Rozwiązanie 2:
Dni = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Kolory = [3, 5, 2, 4, 1, 6, 7, 8, 9, 10]
Przedmioty = [3, 1, 2, 5, 8, 6, 7, 4, 9, 10]
Oceny = [1, 5, 2, 4, 3, 6, 7, 8, 9, 10]
Techniki = [1, 2, 8, 3, 5, 6, 7, 4, 9, 10]

pn 8-12	wt 8-12	śr 8-12	czw 8-12	pt 8-12
żółta	niebieska	czerwona	biała	zielona
matematyka	informatyka	fizyka	historia	ekonomia
nudne	ciekawe	takie_sobie	bardzo_ciekawe	trudne
komputer	internet	kreda_tablica	mapy	projektor

pn 12-16	wt 12-16	śr 12-16	czw 12-16	pt 12-16
różowa	fioletowa	pomarańczowa	brązowa	szara
chemia	niemiecki	angielski	muzyka	elektronika
wyczerpujące	zabawne	b.popularne	rozśpiewane	absorbujące
odczynniki	słowniki	magnetowid	fortepian	oscyloskop

Rysunek 4.2: Pierwsze dwa rozwiązania dla 10 sal

kolory sal(czerwona, zielona, niebieska, biała, żółta,
różowa, fioletowa, pomarańczowa, brązowa, szara),
dni tygodnia-do południa i po południu(pn 8-12,wt 8-12,sr 8-12,czw 8-12,pt 8-12,
pn 12-16,wt 12-16,sr 12-16,czw 12-16,pt 12-16),
przedmioty(fizyka,matematyka,informatyka,ekonomia,j_angielski,
chemia,niemiecki,historia,muzyka,elektronika),
oceny przedmiotów(nudne, trudne, ciekawe, bardzo_ciekawe, takie_sobie,
wyczerpujące, zabawne, b_popularne, rozspiewane, absorbujace),
technika prowadzenia przedmiotów(komputer, internet, magnetowid, kreda_tablica, projektor,
odczynniki, slowniki, mapa, fortepian, oscyloskop)

Rozwiązanie 3:
Dni = [1, 2, 3, 7, 5, 6, 4, 8, 9, 10]
Kolory = [3, 5, 2, 4, 1, 6, 7, 8, 9, 10]
Przedmioty = [3, 1, 2, 5, 4, 6, 7, 8, 9, 10]
Oceny = [1, 5, 2, 7, 3, 6, 4, 8, 9, 10]
Techniki = [1, 2, 4, 3, 5, 6, 7, 8, 9, 10]

pn 8-12	wt 8-12	śr 8-12	czw 8-12	pt 8-12
żółta	niebieska	czerwona	fioletowa	zielona
matematyka	informatyka	fizyka	niemiecki	ekonomia
nudne	ciekawe	takie_sobie	bardzo_ciekawe	trudne
komputer	internet	kreda_tablica	słowniki	projektor

pn 12-16	wt 12-16	śr 12-16	czw 12-16	pt 12-16
różowa	biała	pomarańczowa	brązowa	szara
chemia	angielski	historia	muzyka	elektronika
wyczerpujące	zabawne	b.popularne	rozśpiewane	absorbujące
odczynniki	magnetowid	mapa	fortepian	oscyloskop

Rozwiązanie 4:
Dni = [1, 2, 3, 7, 5, 6, 4, 8, 9, 10]
Kolory = [3, 5, 2, 4, 1, 6, 7, 8, 9, 10]
Przedmioty = [3, 1, 2, 5, 8, 6, 7, 4, 9, 10]
Oceny = [1, 5, 2, 7, 3, 6, 4, 8, 9, 10]
Techniki = [1, 2, 8, 3, 5, 6, 7, 4, 9, 10]

pn 8-12	wt 8-12	śr 8-12	czw 8-12	pt 8-12
żółta	niebieska	czerwona	fioletowa	zielona
matematyka	informatyka	fizyka	niemiecki	ekonomia
nudne	ciekawe	takie_sobie	bardzo_ciekawe	trudne
komputer	internet	kreda_tablica	słowniki	projektor

pn 12-16	wt 12-16	śr 12-16	czw 12-16	pt 12-16
różowa	biała	pomarańczowa	brązowa	szara
chemia	historia	angielski	muzyka	elektronika
wyczerpujące	zabawne	b.popularne	rozśpiewane	absorbujące
odczynniki	mapy	magnetowid	fortepian	oscyloskop

Rysunek 4.3: Kolejne dwa rozwiązania dla 10 sal

4.5.3 Wszystko dla Wszystkich

Ograniczenia mają często charakter warunkowy. Jak radzić sobie z nimi ilustruje następujący przykład:

Ważną siłą polityczną w Absurdolandii jest popularna partia "Wszystko dla Wszystkich". W jej centrali w każdy piątek omawia się delegacje zaplanowane na przyszły tydzień w związku z toczącą się kampanią wyborczą. W ostatni piątek postanowiono odwiedzić oddziały partii i uczestniczyć w zebraniach przedwyborczych w Dziurze Dolnej, Dziurze Górnej i Dziurze Niskiej, w poniedziałek, we wtorek względnie w środę. Odwiedzin mają dokonać działacze Bredzik i Bajdurczyk oraz działaczka Bajka-Bujalska. Każdy z działaczy ma odwiedzić jeden oddział i spotkać się z miejscowymi działaczami oraz wyborcami. W każdym dniu tylko jeden z nich może wyjechać, gdyż są to działacze niezbędni także w centrali. Ale kto z nich i dokąd ma się udać? Każdy z wymienionych działaczy ma specjalne życzenia:
1)Działacz Bredzik postanowił nigdy już nie pojechać do Dziury Dolnej, gdyż tam ostatnio częstowano go obiadem w przydrożnej szaszłykarni, gdzie również wręczono mu upominek w postaci kompletu chińskich długopisów, co oczywiście starannie ukrył przed partyjnymi kolegami.
2)Działacz Bajdurczyk chętnie pojedzie do Dziury Dolnej lub do Dziury Górnej, ale nie we wtorek, gdyż sponsorujący go Znany Przedsiębiorca, z którym zwykł spotykać się nad przypadkowo wybraną mogiłą na jednym z dziurodolnych lub dziurogórnych cmentarzy, tradycyjnie spędza wtorek z jedną ze swych przyjaciółek. Działacz Bajdurczyk przypisuje dużą wagę do swych spotkań ze Znanym Przedsiębiorcą, gdyż w ich trakcie otrzymuje plastikową reklamówkę z gotówką i spisem najważniejszych spraw do załatwienia, wraz z sugestiami jak je załatwić.
3)Do Dziury Górnej działacz Bajdurczyk też nie chce jechać w poniedziałek, gdyż w poniedziałki nieczynne są tamtejsze agencje towarzyskie.
4)Do Dziury Dolnej w poniedziałki nikt nie lubi jeździć, gdyż wtedy każdy tamtejszy bar i motel serwuje tradycyjnie reanimowane potrawy niedzielne.
5)Działacza Bredzika nie należy posyłać do Dziury Górnej, gdyż w trakcie ostatniego tam pobytu nie potrafił przekonywująco przedstawić tego punktu programu partii, w którym obiecuje ona państwową gwarancję kredytu bankowego każdemu bezrobotnemu, który dla potrzeb planowanej działalności gospodarczej zapragnie kupić nową limuzynę znanej marki "Luxus".
6)Do Dziury Niskiej może spokojnie jechać działacz Bredzik, ale prosi on, by nie było to w poniedziałek, gdyż ten dzień stanowi już tradycyjnie przedłużenie jego standardowego weekendu.

7)Działaczka Bajka-Bujalska nie powinna jechać do Dziury Niskiej, gdyż w trakcie ostatniego tam pobytu odmówiła prezesowi Dziuroniskiego Koła Partyjnego poparcia wniosku o przyznanie mu prestiżowego i wielce w Absurdolandii cenionego orderu "Virtuti Bałamuti", jako że sama nie została jeszcze tym orderem odznaczona.

Czy można znaleźć takie rozwiązanie dla delegacji, które by odpowiadało wszystkim warunkom i życzeniom?

Odpowiedni program (`4_15_delegacje.ecl`) ma postać:

```
/*1*/   :-lib(ic).

/*2*/   top:-
/*3*/     Miasta=[Delegacja_Bredzika,Delegacja_Bajdurczyka,
                  Delegacja_Bajki_Bujalskiej],
/*4*/     Miasta::1..3, % trzy odwiedzane miasta: 1 - Dziura Dolna,
                  2 - Dziura Górna, 3 -Dziura Niska
   %Np. jeżeli Delegacja_Bredzika=3, to do Dziury Niskiej pojedzie Bredzik.
/*5*/     alldifferent(Miasta),

/*6*/     Dni=[Poniedzialek,Wtorek,_],
/*7*/     Dni::1..3, % trzy odwiedzane miasta: 1 - Dziura Dolna,
   %Np. jeżeli Wtorek=3, to we wtorek będzie delegacja do Dziury Niskiej.
/*8*/     alldifferent(Dni),

   % 1)Działacz Bredzik nie pojedzie do Dziury Dolnej:
/*9*/     Delegacja_Bredzika #\= 1,

   % 2)Działacz Bajdurczyk chętnie pojedzie do Dziury Dolnej
           % lub do Dziury Górnej, ale nie we wtorek:
/*10*/    ograniczenie_2(Delegacja_Bajdurczyka,Wtorek),

   % 3)Do Dziury Górnej działacz Bajdurczyk też nie chce jechać
           % w poniedziałek:
/*11*/    ograniczenie_3(Delegacja_Bajdurczyka,Poniedzialek),

   % 4)Do Dziury Dolnej  w poniedziałki  nikt nie lubi jechać:
/*12*/    Poniedzialek #\= 1,

   % 5)Działacza Bredzika nie należy posyłać do Dziury Górnej:
/*13*/    Delegacja_Bredzika #\= 2,

   % 6)Do Dziury Niskiej może jechać działacz Bredzik,
   % ale nie w poniedziałek:
/*14*/    ograniczenie_6(Delegacja_Bredzika,Poniedzialek),
```

```
   % 7)Koleżanka Bajka-Bujalska nie powinna jechać do Dziury
           % Niskiej:
/*15*/    Delegacja_Bajki_Bujalskiej #\= 3,

/*16*/    write("Miasta = "),writeln(Miasta),
/*17*/    write("Dni = "),writeln(Dni),nl,

   % Koniec części programu rozwiązującej zadanie*/

   % Początek części programu piszącej rozwiązanie.
   % Istota problemu: znaleźć trójki (Nazwisko,Miejsce,Dzień)
   % o tych samych numerach

   % numer miejscowości na pozycji 1:
/*18*/    element(1,Miasta,Numer_miejscowosci_1),
   % numer dnia na pozycji Numer_miejscowosci_1:
/*19*/    element(Numer_dnia_1,Dni,Numer_miejscowosci_1),

   % numer miejscowości na pozycji 2:
/*20*/    element(2,Miasta,Numer_miejscowosci_2),
   % numer dnia na pozycji Numer_miejscowosci_2:
/*21*/    element(Numer_dnia_2,Dni,Numer_miejscowosci_2),

   % numer miejscowości na pozycji 3:
/*22*/    element(3,Miasta,Numer_miejscowosci_3),
   % numer dnia na pozycji Numer_miejscowosci_3:
/*23*/    element(Numer_dnia_3,Dni,Numer_miejscowosci_3),

%   translacja numeru miejscowości na nazwę miejscowości:
/*24*/    miejscowosc(Numer_miejscowosci_1,Nazwa_miejscowosci_1),
/*25*/    miejscowosc(Numer_miejscowosci_2,Nazwa_miejscowosci_2),
/*26*/    miejscowosc(Numer_miejscowosci_3,Nazwa_miejscowosci_3),

%   % translacja numeru dnia na nazwę dnia:
/*27*/    dzien(Numer_dnia_1,Nazwa_dnia_1),
/*28*/    dzien(Numer_dnia_2,Nazwa_dnia_2),
/*29*/    dzien(Numer_dnia_3,Nazwa_dnia_3),

   % łączenie elementów trójek:
/*30*/    write("Dzialacz Bredzik pojedzie do "),
            write(Nazwa_miejscowosci_1),write(Nazwa_dnia_1),nl,
/*31*/    write("Dzialacz Bajdurczyk pojedzie do "),
            write(Nazwa_miejscowosci_2),write(Nazwa_dnia_2),nl,
/*32*/    write("Dzialaczka Bajka-Bujalska pojedzie do "),
             write(Nazwa_miejscowosci_3),write(Nazwa_dnia_3),nl.

/*33*/  ograniczenie_2(Delegacja_Bajdurczyka,Wtorek):-
/*34*/    Delegacja_Bajdurczyka #= 1,
```

```
/*35*/    Wtorek #\= 1;
/*36*/    Delegacja_Bajdurczyka #= 2,
/*37*/    Wtorek #\= 2.
/*38*/  ograniczenie_2(_,_).

/*39*/  ograniczenie_3(Delegacja_Bajdurczyka,Poniedzialek):-
/*40*/    Delegacja_Bajdurczyka #= 2,
/*41*/    Poniedzialek #\= 2.
/*42*/  ograniczenie_3(_,_).

/*43*/  ograniczenie_6(Delegacja_Bredzika,Poniedzialek):-
/*44*/    Delegacja_Bredzika #= 3,
/*45*/    Poniedzialek #\= 3.
/*46*/  ograniczenie_6(_,_).

   % translacja numeru miejscowości na nazwę  miejscowości
/*47*/    miejscowosc(1,"Dziury Dolnej").
/*48*/    miejscowosc(2,"Dziury Gornej").
/*49*/    miejscowosc(3,"Dziury Niskiej").

   % translacja numeru dnia na nazwę dnia
/*50*/    dzien(1," w poniedzialek.").
/*51*/    dzien(2," we wtorek.").
/*52*/    dzien(3," w srode.").
```

Program generuje komunikat:

```
   Miasta = [3, 1, 2]
      Dni = [2, 3, 1]
Dzialacz Bredzik pojedzie do Dziury Niskiej we wtorek.
Dzialacz Bajdurczyk pojedzie do Dziury Dolnej w srode.
Dzialaczka Bajka-Bujalska pojedzie do Dziury Gornej w poniedzialek.
```

4.6 Obsługa danych

Rozwiązywanie bardziej złożonych problemów kombinatorycznych wymaga poszerzenia naszej wiedzy w zakresie sposobów przedstawiania danych i ich przetwarzania w systemie ECL^iPS^e CPS.

4.6.1 Struktury i tablice

Struktury są deklarowane za pomocą szablonów `local struct(++Struktura)`, np. typu:

```
:- local struct(osoba(nazwisko, adres, wiek)).
:- local struct(pracownik(o:osoba, zarobki)).
```

gdzie struktura `osoba` jest zagnieżdżona w strukturze `pracownik` za pośrednictwem pola `o`. Przykładem zastosowania powyższych struktur jest następujące polecenie wydane systemowi w trybie wiersza polecenia (patrz rozdział 0.3) oraz odpowiedź na to polecenie:

```
% to jest polecenie:
[eclipse 1]:
:- local struct(osoba(nazwisko, adres, wiek)).
:- local struct(pracownik(o:osoba, zarobki)).
Pracownik = pracownik with [nazwisko: "Jan Kowalski", wiek: 26,
            zarobki: 4000, adres: "Gliwice, Kormoranow 5"],
arg(nazwisko of pracownik, Pracownik, Nazwisko),
arg(wiek of pracownik, Pracownik, Wiek),
arg(zarobki of pracownik, Pracownik, Zarobki).
% to jest odpowiedź:
Pracownik = pracownik(osoba("Jan Kowalski",
            "Gliwice, Kormoranow 5", 26), 4000)
Nazwisko = "Jan Kowalski"
Wiek = 26
Zarobki = 4000
```

Ważną formą przedstawiania i przetwarzania danych są wielowymiarowe tablice (ang. *arrays*). Tablice są strukturami z funktorem [], obsługiwanymi za pomocą następujących predykatów:

1)1-wymiarową tablicę o czterech elementach można wyznaczyć za pomocą predykatu `dim(Tablica,[4])`, otrzymując wynik (w trybie wiersza poleceń) w postaci jednowymiarowej tablicy z czterema wolnymi zmiennymi:

```
% to jest polecenie:
[eclipse 2]: dim(Tablica,[4]).
```

```
% to jest odpowiedź:
Tablica = [](_169, _170, _171, _172)
```

2)2-wymiarową tablicę o wymiarach 3 x 2 można wyznaczyć następująco:

```
% to jest polecenie:
[eclipse 3]: dim(Tablica,[3,2]).
% to jest odpowiedź:
Tablica = []([](_181, _182), [](_178, _179), [](_175, _176))
```

3)Wymiary 2-wymiarowej tablicy można wyznaczyć następująco:

```
% to jest polecenie:
[eclipse 4]: Tablica = []([](a,b,c),[](d,e,f)),dim(Tablica,D).
% to jest odpowiedź:
D = [2, 3]
```

4)Predykat `dim/2` może również służyć do wyznaczenia liczby elementów jednowymiarowej tablicy:

```
% to jest polecenie:
[eclipse 5]: Tablica = [](a, b, c, d), dim(Tablica,D).
% to jest odpowiedź:
Tablica = [](a, b, c, d)
D = [4]
```

5)Jednowymiarowa tablica może mieć elementy będące listami. Liczba elementów takiej tablicy jest równa liczbie list, np.:

```
% to jest polecenie:
[eclipse 6]: Tablica=[]([5,7,1,20],[14,8,100,300],[2,20,50,12] ),
dim(Tablica,[M]).
% to jest odpowiedź:
Tablica = []([5, 7, 1, 20], [14, 8, 100, 300],
[2, 20, 50, 12])
M = 3
```

6)W przypadku listy można to wyznaczyć liczbę jej elementów za pomocą predykatu `length(?Lista, ?Dlugość_Listy)`. Np.

```
% to jest polecenie:
[eclipse 7]: length([a,b,c,d], Dlugosc_Listy).
% to jest odpowiedź:
Dlugosc_Listy = 4
```

Predykat `length/2` może również służyć do wyznaczenia listy. Np.:

```
% to jest polecenie:
[eclipse 8]: length(Lista, 4).
% to jest odpowiedź:
Lista = [_166, _168, _170, _172]
```

7)I-ty wiersza tablicy o M kolumnach można wyznaczyć za pomocą konstrukcji `Wiersz_I is Tablica(I,1..M]`, gdzie `Wiersz_I` jest listą. Np.:

```
% to jest polecenie:
[eclipse 9]: Tablica = []([](a,b,c),[](d,e,f)),
            Wiersz_Drugi is Tablica[2,1..3].
% to jest odpowiedź:
Tablica = []([](a, b, c), [](d, e, f))
Wiersz_Drugi = [d, e, f]
```

8)J-tą kolumnę tablicy o N wierszach można wyznaczyć za pomocą konstrukcji `Kolumna_J is Tablica(1..N,J]`, gdzie `Kolumna_J` jest listą. Np.:

```
% to jest polecenie:
[eclipse 10]: Tablica = []([](a,b,c),[](d,e,f)),
            Kolumna_Trzecia is Tablica[1..2,3].
% to jest odpowiedź:
Tablica = []([](a, b, c), [](d, e, f))
Kolumna_Trzecia = [c, f]
```

4.6.2 Jak wyłuskać elementy macierzy?

Jeżeli należy wykonywać operacje na kolejnych wierszach macierzy, przedstawiony powyżej sposób wyłuskania tych wierszy jest nieprzydatny. W tym celu

należy zastosować predykat:

```
arg(+Nr_wiersza, +Tablica_Macierz, -Tablica_Wiersz_Macierzy),
```

wyznaczający (w postaci tablicy) wiersz macierzy o zadanym numerze.

Ilustruje to przykład `4_16_wyznaczanie_elementow.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   tablica_macierz(Tablica_Macierz):-
/*3*/         Tablica_Macierz=[](
/*4*/         [](1,2,3,4,5),
/*5*/         [](6,7,8,9,10),
/*6*/         [](11,12,13,14,15)
/*7*/         ).
/*8*/   top:-
/*9*/         tablica_macierz(Tablica_Macierz),
/*10*/        arg(2,Tablica_Macierz,Tablica_Wiersz_Macierzy),
/*11*/        writeln("Tablica_Wiersz_Macierzy":Tablica_Wiersz_Macierzy),
/*12*/        Tablica_Wiersz_Macierzy=..[[]|Lista_Wiersz_Macierzy],
/*13*/        writeln("Lista_Wiersz_Macierzy":Lista_Wiersz_Macierzy),
/*14*/        element(3,Lista_Wiersz_Macierzy,Element_2_3),
/*15*/        writeln("Element_2_3":Element_2_3).
```

Program generuje komunikat:

```
Tablica_Wiersz_Macierzy : [](6, 7, 8, 9, 10)
Lista_Wiersz_Macierzy : [6, 7, 8, 9, 10]
Element_2_3 : 8
```

Jeżeli interesuje nas tylko konkretny element macierzy, można zastosować prostsze podejście, wywołując `Tablica_Macierz[współrzędne_elementu]`, jak poniżej:

```
Tablica_Macierz=[](
                        [](1,2,3,4,5),
                        [](6,7,8,9,10),
                        [](11,12,13,14,15)
                        ),
X is Tablica_Macierz[2,3].
```

Generowanym komunikatem jest:

```
Tablica_Macierz = [](1, 2, 3, 4, 5), [](6, 7, 8, 9, 10), [](11, 12, 13, 14, 15))
X = 8
```

4.6.3 Rekurencje a iteracje

W ECL^iPS^e *Prolog* - jak w każdym innym *Prolog*u - podstawą przetwarzania zmiennych są rekurencje. Np. wypisanie kolejnych elementów listy można uzyskać za pomocą prostej reguły rekurencyjnej `pisz_liste`, użytej w programie `4_17_pisz_liste.ecl`:

```
top :-
    pisz_liste([1,2,3]).

pisz_liste([X|Xs]):-
    writeln(X),
    pisz_liste(Xs).
pisz_liste([]).
```

Istota rekurencji polega na tym, że predykat `pisz_liste(_)` jest zdefiniowany przez samego siebie. Uruchomienie tego programu daje:

```
1
2
3
```

Rekurencyjne programowanie - jak mieliśmy okazję wielokrotnie się przekonać - funkcjonuje również w ECL^iPS^e CPS. Twórcy ECL^iPS^e zdecydowali się jednak na uzupełnienie rekurencji iteracjami, których istotą jest powtarzanie tego samego predykatu w pętli na zmieniających się danych. Iteracje praktycznie w *Prolog*ach nie są ani stosowane, ani spotykane. Twórcy ECL^iPS^e CPS kierowali się zapewne chęcią uczynienia go bliższym przyzwyczajeniom programistów języków proceduralnych. W wyniku utracono co nieco na deklaratywności ECL^iPS^e CPS, a tym samym na czytelności programów.

Podstawowym predykatem iteracyjnym jest `do/2` stosowany w postaci:

```
+definicja_iteracji(X) do +cel(X)
```

dla wykonywania `cel(X)` zgodnie z `definicją_iteracji(X)`. Dopuszczalne są następujące definicje iteracji:

1) `foreach(X,Lista) do cel(X)` iteruje `cel(X)` dla wszystkich `X` będących elementami listy `Lista`. `X` jest zmienną lokalną dla `cel(X)`. Np.:

```
% to jest polecenie:
[eclipse 1]: (foreach(X, [1,2,3]) do writeln(X)).
% to jest odpowiedź:
1
2
3
X = X
```

Odpowiedź jest więc identyczna jak dla predykatu `pisz_liste(_)`. Jednakże `foreach(X,Lista)` może być również stosowany do konstrukcji list:

```
% to jest polecenie:
[eclipse 2]: (foreach(X, [1,2,3]), foreach(Y,Lista) do Y is X+5).
% to jest odpowiedź:
X = X
Y = Y
Lista = [6, 7, 8]
```

Możliwość stosowania do konstrukcji list jest wspólna dla większości definicji iteracji; zdejmuje ona w jakimś sensie "brzemię proceduralności" z tych definicji. Skorzystano z niej w programie `4_19_iloczyn_skalarny.ecl` (patrz 4.6.4), wyznaczającym iloczyn skalarny dwóch wektorów przedstawionych w postaci list.

Iloczyn skalarny zostanie z kolei zastosowany w programie `5_6_plecak_1.ecl`, patrz 5.6.3.

2) `foreacharg(X,predykat(a,b,..)) do cel(X)` iteruje `cel(X)` dla wszystkich `X` będących argumentami predykatu `predykat(a,b,..)`. Np.:

```
% to jest polecenie:
[eclipse 3]: (foreacharg(X, s(p,q,r)) do writeln(X)).
% to jest odpowiedź:
p
```

```
q
r
X = X
```

Predykat `foreacharg(X,predykat(a,b,..))` nie może być stosowany do konstrukcji predykatu `predykat(a,b,..)` ze względu na niejednoznaczność tego pojęcia.

3) `foreacharg(X,predykat(a,b,..),I) do cel(X)` iteruje `cel(X)` dla wszystkich `X` będących elementami predykatu `predykat(a,b,..)`, udostępniając ponadto numery `I` pozycji `X` w predykacie. Np.:

```
% to jest polecenie:
[eclipse 3]: (foreacharg(X, s(p,q,r),I) do writeln(X),writeln(I)).
% to jest odpowiedź:
p
1
q
2
r
3
X = X
```

4) `param(Zmienna1,Zmienna2,...)` umożliwia wprowadzanie zmiennych do pętli `do`. Ilustruje to przykład wyznaczania wszystkich par elementów listy:

```
% to jest polecenie:
[eclipse 4]: Lista = [1,2,3],
( foreach(X, Lista), param(Lista) do
   ( foreach(Y,Lista), param(X) do
   write(X),write(" "),write(Y),nl
   )
).
% to jest odpowiedź:
1  1
1  2
1  3
2  1
2  2
```

```
2  3
3  1
3  2
3  3
Lista = [1, 2, 3]
X = X
Y = Y
```

Inny przykład zastosowania `param` dla pętli `for/3` to przykład rozwiązania zadania o hetmanach dla bardziej zwięzłej wersji korzystającej z prawidłowości zademonstrowanej w przykładzie `4_9_hetmani_raz_jeszcze.ecl`. Przykładem tym jest przykład `4_18_hetmani_ponownie.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   top:-
/*3*/       hetmani(_,_).

/*4*/   hetmani(N, Szachownica) :-
/*5*/       rozmiary_szachownicy(N),
/*6*/       dim(Szachownica, [N]),
/*7*/       Szachownica[1..N] :: 1..N,
/*8*/       (for(I,1,N), param(Szachownica,N) do
/*10*/   (for(J,I1,N), param(Szachownica,I) do +
/*11*/   Szachownica[I] #\= Szachownica[J],
/*12*/   Szachownica[I] #\= Szachownica[J]J-I, +
/*13*/   Szachownica[I] #\= Szachownica[J]I-J +
/*14*/   )
/*15*/   ),
/*16*/   labeling(Szachownica),
/*17*/   writeln(Szachownica).

/*18*/   rozmiary_szachownicy(4).
```

5)`count(I,Min,Max) do cel(I)` iteruje `cel(I)` dla wszystkich liczb całkowitych `I` z przedziału `[Min...Max]`. `I` jest zmienną lokalną dla `cel(I)`. Ilustruje to przykład tworzenia listy liczb całkowitych:

```
% to jest polecenie:
[eclipse 5]: (count(I,1,4), foreach(I,Lista) do true).
% to jest odpowiedź:
I = I
Lista = [1, 2, 3 ,4]
```

6) `for(I,MinExpr,MaxExpr)do cel(I)` iteruje `cel(I)` dla wszystkich zmiennych całkowitych `I` z przedziału `[MinExpr...MaxExpr]`. `I` jest zmienną lokalną dla `cel(I)`, a `MinExpr` i `MaxExpr` mogą być wyrażeniami arytmetycznymi. Predykat ten może służyć jedynie dla iteracji, tzn. `MaxExpr` musi być uziemione. Ilustruje to przykład tworzenia listy liczb całkowitych:

```
% to jest polecenie:
[eclipse 6]: (for(I,1,5), foreach(I,Lista) do true).
% to jest odpowiedź:
I = I
Lista = [1, 2, 3, 4, 5]
```

7) `for(I,MinExpr,MaxExpr,Delta)do cel(I)` iteruje `cel(I)` dla wszystkich zmiennych całkowitych `I` z przedziału `[MinExpr...MaxExpr]` z przyrostem `Delta`. `I` jest zmienną lokalną dla `cel(I)`, a `MinExpr` i `MaxExpr` mogą być wyrażeniami arytmetycznymi. Predykat ten może służyć jedynie dla iteracji, tzn. `MaxExpr` musi być uziemione. Ilustruje to przykład tworzenia listy liczb całkowitych:

```
% to jest polecenie:
[eclipse 7]: (for(I,1,5,2), foreach(I,Lista) do true).
% to jest odpowiedź:
I = I
Lista = [1, 3, 5]
```

8) `multifor(Lista,ListaMin,ListaMax)do cel(Lista)` jest uogólnieniem przypadku `5` dla `for/3` w sytuacji wymagającej iteracji dla większej liczby zmiennych. `multifor` iteruje `cel(Lista)` dla wszystkich zmiennych całkowitych będących elementami listy `Lista` z przedziału określonego listami `ListaMin` i `ListaMax`. `Lista` jest zmienną lokalną `cel(Lista)`, a `MinExpr` i `MaxExpr` mogą być listami zawierającymi taką samą liczbę wyrażeń arytmetycznych. Predykat ten może służyć jedynie dla iteracji, tzn. `MaxExpr` musi być uziemione. Jego stosowanie ilustruje następujący przykład:

```
% to jest polecenie:
[eclipse 8]: (multifor([I,J],[1,2],[2,4]) do writeln([I,J]),
                    K is I+J, writeln([K])).
% to jest odpowiedź:
[1, 2]
```

```
[3]
[1, 3]
[4]
[1, 4]
[5]
[2, 2]
[4]
[2, 3]
[5]
[2, 4]
[6]
I = I
J = J
K = K
```

Interesującym zastosowaniem `multifor/3` jest rozwiązywanie łamigłówek *Sudoku*, zob. program `4_20_sudoku.ecl` w rozdziale 4.7.1.

9) `multifor(Lista,ListaMin,ListaMax,ListaDelta) do cel(Lista)` jest uogólnieniem przypadku `6` dla `for/3` w sytuacji wymagającej iteracji dla większej liczby zmiennych z przyrostami określonymi przez `ListaDelta`. Jego stosowanie ilustruje następujący przykład:

```
% to jest polecenie:
[eclipse 9]: (multifor([I,J],[1,2],[2,5],[1,2]) do writeln([I,J]),
                    K is I+J, writeln([K])).
 % to jest odpowiedź:
[1, 2]
[3]
[1, 4]
[5]
[2, 2]
[4]
[2, 4]
[6]
I = I
J = J
K = K
```

10) `fromto(Pierwszy,We,Wy,Ostatni) do cel(We,Wy)` iteruje `cel(We,Wy)` poczynając od `We = Pierwszy` aż do `Wy = Ostatni`.
`We` i `Wy` są zmiennymi lokalnymi `celu`. Dla wszystkich iteracji z wyjątkiem pierwszej wartość `We` jest równa wartości `Wy` w poprzedniej iteracji. Ilustruje to przykład wyznaczania sumy listy liczb całkowitych:

```
% to jest polecenie:
[eclipse 10]: (foreach(X,[10,20,30]),
        fromto(0,We,Wy,Suma) do Wy is We+X).
% to jest odpowiedź:
X = X
We = We
Wy = Wy
Suma = 60
```

Odwracanie list można wykonać następująco:

```
% to jest polecenie:
[eclipse 11]: (foreach(X,[10,20,30]),
        fromto([],We,[X|We],Lista_Odwrotna) do true).
% to jest odpowiedź:
X = X
We = We
Lista_Odwrotna = [30, 20, 10]
```

Najbardziej finezyjne zastosowania `fromto/4` to takie, dla których argument `Pierwszy` zostaje ukonkretniony dopiero w wyniku iteracji. Sytuacja taka występuje dla najróżnorodniejszych filtracji zmiennych. Np.:

```
% to jest polecenie:
[eclipse 12]: (foreach(X,[5,3,8,1,4,6]),
    fromto(Lista,We,Wy,[]) do
        X>3 -> We=[X|Wy] ; Wy=We).
% to jest odpowiedź:
X = X
Lista = [5, 8, 4, 6]]
We = We
Wy = Wy
```

Przykład `4_21_hetmani_po_raz_ostatni.ecl` jest również przykładem, dla którego argument `Pierwszy` zostaje ukonkretniony dopiero w wyniku iteracji, patrz rozdział 4.7.2.

4.6.4 Iloczyn skalarny

Iloczynem skalarnym dwóch wektorów danych w postaci list:
`[a1,a2,...,an]`
`[b1,b2,...,bn]`
nazywa się wyrażenie:
`a1*b1 + a2*b2 + ... + an*bn.`
Można je wyznaczyć za pomocą programu `4_19_iloczyn_skalarny.ecl`:

```
/*1*/   :- lib(ic).

/*2*/   top:-
/*3*/       iloczyn_skalarny([1,2,3,4],[10,20,30,40],_).

/*4*/   iloczyn_skalarny(Lista_1,Lista_2,Iloczyn_skalarny):-
/*5*/       (foreach(V1, Lista_1),
/*6*/       foreach(V2, Lista_2),
/*7*/       foreach(Iloczyn,Iloczyn_lista)
/*8*/       do
/*9*/       Iloczyn is V1 * V2
/*10*/      ),
/*11*/      Iloczyn_skalarny #= sum(Iloczyn_lista),nl,
/*12*/      write("Iloczyn skalarny = "),writeln(Iloczyn_skalarny),nl.
```

Program daje w wyniku:

```
Iloczyn skalarny = 300
```

4.7 Kombinatoryczne przyporządkowanie - ciąg dalszy

4.7.1 Sudoku

Sudoku jest łamigłówką liczbową. Jej podstawą jest kwadratowa tablica składająca się z 81 komórek, rozmieszczonych w 9 rzędach po 9 komórek, i w 9 kolumnach po 9 komórek. Komórki te tworzą z kolei 9 kwadratów 9-komórkowych o wymiarach 3 x 3. W niektórych komórkach zostały już wpisane cyfry od 1 do 9. W pozostałe komórki należy wpisać również cyfry od 1 do 9. Należy to zrobić tak, aby w każdym wierszu tablicy, w każdej kolumnie tablicy i w każdym 9-komórkowym kwadracie poszczególne liczby występowały tylko raz. Program `4_20_sudoku.ecl` jest programem rozwiązującym łamigłówki *sudoku* z zastosowaniem predykatu `multifor/3`. Jest on nieco zmodyfikowaną wersją programu przedstawionego na witrynie [Schimpf-10]:

```
/*1*/   :- lib(ic).

/*2*/   top:-
/*3*/       write("Napisz numer zadania (1,2 lub 3):"),nl,
/*4*/       read_token(Numer, integer),
/*5*/       wyznacz(Numer).

/*6*/   wyznacz(Numer):-
/*7*/       zadanie(Numer, Tabela),
/*8*/       pisz_tabela(Tabela),
/*9*/       sudoku(Tabela),
/*10*/      pisz_tabela(Tabela).

/*11*/  sudoku(Tabela):-
/*12*/      Tabela[1..9,1..9] :: 1..9,
/*13*/      (for(I,1,9), param(Tabela)
/*14*/      do
/*15*/            Row is Tabela[I,1..9],
/*16*/            alldifferent(Row),
/*17*/            Col is Tabela[1..9,I],
/*18*/            alldifferent(Col)
/*19*/      ),
/*20*/      (multifor([I,J],1,9,3), param(Tabela)
/*21*/      do
/*22*/            (multifor([K,L],0,2),
```

```
/*23*/            param(Tabela,I,J),
/*24*/            foreach(X,Kwadrat)
/*25*/            do
/*26*/                X is Tabela[I+K,J+L]
/*26*/            ),
/*27*/      alldifferent(Kwadrat)
/*28*/      ),
/*29*/      term_variables(Tabela, Zmienne),
/*30*/      labeling(Zmienne).

/*31*/  pisz_tabela(Tabela):-
/*32*/      (for(I,1,9), param(Tabela)
/*33*/      do
/*34*/            (for(J,1,9), param(Tabela,I)
/*34*/            do
/*35*/                X is Tabela[I,J],
/*36*/                (var(X) -> write("  _") ; printf(" %2d", [X]))
/*37*/            ), nl
/*38*/      ), nl.

    zadanie(1, []([]( _, _, 2, _, 6, _, _, _, 3),
     []( _, _, _, 5, 9, _, _, 7, _),
     []( _, 6, _, _, _, 4, _, _, _),
     []( _, _, _, _, _, 1, _, 3, 7),
     []( _, 9, 7, _, _, _, 8, 1, _),
     [](4, 1, _, 7, _, _, _, _, _),
     []( _, _, _, 8, _, _, _, 9, _),
     []( _, 2, _, _, 7, 5, _, _, _),
     [](6, _, _, _, 4, _, 3, _, _))).

    zadanie(2, []([]( _, _, 5, _, 7, 4, _, 6, _),
     [](9, _, _, _, _, 3, _, _, _),
     []( _, _, 1, _, _, _, _, 3, 2),
     []( _, _, _, 9, _, _, _, _, 5),
     []( _, _, _, _, _, _, _, _, _),
     [](4, _, _, _, _, 6, _, _, _),
     [](8, 6, _, _, _, _, 7, _, _),
     []( _, _, _, 1, _, _, _, _, 8),
     []( _, 2, _, 8, 6, _, 5, _, _))).

    zadanie(3, []([]( _, _, _, _, _, 1, 2, _, _),
     []( _, _, _, _, _, _, 9, 6, _),
     []( _, _, _, 7, 4, 6, _, _, 8),
     []( _, 9, 3, _, 2, _, _, 1, _),
```

```
[](_, 8, _, _, _, _, _, 9, _),
[](_, 6, _, _, 5, _, 8, 2, _),
[](1, _, _, 5, 6, 7, _, _, _),
[](_, 3, 4, _, _, _, _, _, _),
[](_, _, 6, 3, _, _, _, _, _))).
```

Rozwiązując zadanie 3 otrzymuje się:

```
_  _  _  _  _  1  2  _  _
_  _  _  _  _  _  9  6  _
_  _  _  7  4  6  _  _  8
_  9  3  _  2  _  _  1  _
_  8  _  _  _  _  _  9  _
_  6  _  _  5  _  8  2  _
1  _  _  5  6  7  _  _  _
_  3  4  _  _  _  _  _  _
_  _  6  3  _  _  _  _  _

6  5  8  9  3  1  2  4  7
3  4  7  2  8  5  9  6  1
9  1  2  7  4  6  5  3  8
4  9  3  6  2  8  7  1  5
2  8  5  1  7  3  4  9  6
7  6  1  4  5  9  8  2  3
1  2  9  5  6  7  3  8  4
5  3  4  8  1  2  6  7  9
8  7  6  3  9  4  1  5  2
```

4.7.2 Hetmani po raz ostatni

W programie `4_21_hetmani_po_raz_ostatni.ecl` zastosowano fromto/4 do rozwiązania znanego już problemu hetmanów:

```
/*1*/   :- lib(ic).

/*2*/   top:-
/*3*/       czterej_hetmani(4,_).

/*4*/   czterej_hetmani(N, Szachownica):-
/*5*/       length(Szachownica, N),
/*6*/       Szachownica:: 1..N,
```

```
/*7*/       (fromto(Szachownica,[Pozycja1|Pozycje_dalsze],
                                Pozycje_dalsze,[])
/*8*/       do
/*9*/            (foreach(Pozycja2, Pozycje_dalsze),
/*10*/            param(Pozycja1),
/*11*/            count(Dystans,1,_)
/*12*/            do
/*13*/                Pozycja2 #\= Pozycja1,
/*14*/                Pozycja2 - Pozycja1 #\= Dystans,
/*15*/                Pozycja1 - Pozycja2 #\= Dystans
/*16*/            )
/*17*/       ),

/*18*/       labeling(Szachownica),
/*19*/       write("Szachownica = "),writeln(Szachownica).
```

Program generuje następujące odpowiedzi:

```
Szachownica = [2, 4, 1, 3]
Szachownica = [3, 1, 4, 2]
```

4.7.3 Niejawna deklaracja domen - jeszcze raz wykłady

Powróćmy do przykładu o wykładach wybieralnych z rozdziału 2.4.10. Przykład ten można rozwiązać za pomoca programu `4_22_wyklady.ecl`, w którym zastosowano niejawną deklarację domen:

```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/    Wyklady =
             [
             wyklad(1, _, _, _, _, _, _),
             wyklad(2, _, _, _, _, _, _),
             wyklad(3, _, _, _, _, _, _)
             ],

  %  Niejawna deklaracja domen  z linii  /*3*/  jest przeznaczona
  %  dla wszystkich predykatów ' wyklad/7)' z  listy 'Ograniczenia[]':

  %  Zgodnie z  definicją 'Wyklady', zmienne Andrzej, Barbara i  Krzysztof
  %  przyjmują wartości 1, 2 lub 3:
/*4*/    Ograniczenia =
             [
             wyklad(Andrzej, "Andrzej", _, _, _, _, _),
```

```
            wyklad(Barbara, "Barbara",_,_, _, _, _),
            wyklad(Krzysztof, "Krzysztof",_, _, _, _, _),

  %  Zgodnie z  definicją 'Wyklady', zmienne Inzynieria_Wiedzy,
  %  Modele_ekonometryczne i Sztuczna_inteligencja przyjmują wartości 1,2 lub 3:
            wyklad(Inzynieria_Wiedzy,_, "Inzynieria Wiedzy",_,_,_,_),
            wyklad(Modele_ekonometryczne _,"Modele ekonometryczna",_,_,_,_),
            wyklad(Sztuczna_inteligencja,_,"Sztuczna inteligencja",_,_,_,_),

    noindent  %  Zgodnie z  definicją 'Wyklady', zmienne Wtorek,  Sroda,
  %   i  Czwartek przyjmują wartości 1, 2 lub 3:
            wyklad(Wtorek, _, _, "wtorek", _, _, _),
            wyklad(Sroda, _, _, "sroda", _, _, _),
            wyklad(Czwartek, _, _, "czwartek", _, _, _),

  %  Zgodnie z  definicją 'Wyklady', zmienne G1400,  G1545,
  %   i  G1730 przyjmują wartości 1, 2 lub 3:
            wyklad(G1400, _, _, _, "14.00.", _, _),
            wyklad(G1545, _, _, _, "15.45", _, _),
            wyklad(G1730, _, _, _, "17.30", _, _),

  %  Zgodnie z  definicją 'Wyklady', zmienne S104,  SD3,
  %   i  SK2 przyjmują wartości 1, 2 lub 3:
            wyklad(S104, _, _, _, _, "104", _),
            wyklad(SD3, _, _, _, _, "D3", _),
            wyklad(SK2, _, _, _, _, "K2", _),

  %  Zgodnie z  definicją 'Wyklady', zmienne Przybylski,  Jankowski,
  %   i  Kowalski przyjmują wartości 1, 2 lub 3:
            wyklad(Przybylski, _, _, _, _, _, "Przybylski"),
            wyklad(Jankowski, _, _, _, _, _, "Jankowski"),
            wyklad(Kowalski, _, _, _, _, _, "Kowalski")
            ],

% 1) Andrzej będzie chodził na wykład profesora Przybylskiego:
/*5*/    Andrzej #= Przybylski,

% 2) Wtorkowy wykład nie rozpoczyna się o 14.00:
/*6*/    Wtorek #\= G1400,

% 3) Wykład "Inżynieria Wiedzy" nie rozpoczyna się o 17:30:
/*7*/    Inzynieria_Wiedzy #\= G1730,

% 4) Czwartkowy wykład rozpoczyna się 15:45:
/*8*/    Czwartek #= G1545,

% 5) Krzysztof będzie chodzil na wykład "Modele ekonometryczna":
/*9*/    Krzysztof #= Modele_ekonometryczne,
```

```
% 6) Barbara będzie chodziła na wykład wtorkowy:
/*10*/   Barbara #= Wtorek,

% 7) Wykład "Sztuczna inteligencja" będzie w sali D3:
/*11*/   Sztuczna_inteligencja #= SD3,

%8) Środowy wykład nie będzie w sali 104:
/*12*/   Sroda #\= S104,

% 9) Profesor Kowalski nie wykłada "Modeli ekonometrycznych":
/*13*/   Kowalski #\= Modele_ekonometryczne,

% 10) Profesor Jankowski nie wykłada w sali K2:
/*14*/   Jankowski #\= SK2,

/*14*/   uziemienie(Ograniczenia, Wyklady),
/*15*/   (foreach(Wyklad, Wyklady) do writeln(Wyklad)),!.

 %   Wszystkie zmienne z listy  'Ograniczenia[]  muszą zostać uziemione
 %   na wartościach  stalych  z listy 'Wyklady[]'

/*16*/  uziemienie([],_).
/*17*/  uziemienie([H|T],Wyklady) :-
/*18*/     member(H,Wyklady),
/*19*/     uziemienie(T,Wyklady).
```

Rozwiązanie ma postać:

```
wyklad(1, Andrzej, Inzynieria Wiedzy, sroda, 14.00, K2, Przybylski)
wyklad(2, Barbara, Sztuczna inteligencja, wtorek, 17.30, D3, Kowalski)
wyklad(3, Krzysztof, Modele ekonometryczna, czwartek, 15.45, 104, Jankowski)
```

4.7.4 Stabilne małżeństwa

Problem przyporządkowania zwany problemem *stabilnych małżeństw* najlepiej opisuje poniższy cytat ze znanej książki Wirtha ([Wirth-00]):

> *"Załóżmy, że dane są dwa zbiory (rozłączne) A i B o tej samej mocy n. Należy znaleźć zbiór n par (a; b) takich, że a ε A i b ε B oraz a i b spełniają pewne ograniczenia. Istnieje wiele różnych kryteriów ograniczających: jednym z nich jest 'reguła stabilnego małżeństwa'. Załóżmy, że A jest zbiorem mężczyzn, B zaś zbiorem kobiet. Każdy mężczyzna i każda kobieta mają ustalone preferencje dotyczące swoich partnerów. Jeśli n par zostało dobranych tak, że istnieje*

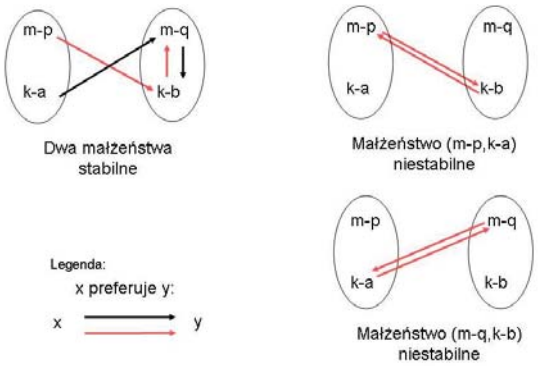
mężczyzna i kobieta, którzy nie stanowią małżeństwa, ale mężczyzna przedkłada tę kobietę nad aktualną żonę, a kobieta przedkłada tego mężczyznę nad aktualnego męża, to dobór małżeństw będziemy nazywać niestabilnym. Jeśli żadna taka para nie istnieje, to dobór będziemy nazywać stabilnym. Sytuacja taka jest charakterystyczna dla wielu podobnych problemów, w których trzeba przyjąć pewne ustalenia (dobór) na podstawie istniejących preferencji, np. wybór szkoły przez uczniów, dobór rekrutów do różnych jednostek wojska itp. Przykład z małżeństwami jest szczególnie intuicyjny...”
Niklaus Wirth, *”Algorytmy + struktury danych = programy.”*

Małżeństwo pomiędzy mężczyzną m (z pewnego zbioru mężczyzn) i kobietą k (z pewnego zbioru kobiet) jest więc stabilne wtedy i tylko wtedy gdy:

1. dla każdego mężczyzny m, który preferuje jakąś kobietę postronną kp ze zbioru kobiet bardziej niż swą żonę k, kobieta postronna kp preferuje swojego męża bardziej niż mężczyznę m, i

2. dla każdej kobiety k, która preferuje jakiegoś mężczyznę postronnego mp ze zbioru mężczyzn bardziej niż swego męża m, mężczyzna postronny mp preferuje swoją żonę bardziej niż kobietę k.

Definicje tę ilustruje przykład z rysunku 4.4.



Rysunek 4.4: Przykłady małżeństw stabilnych i niestabilnych

Jak widać, małżeństwo (`mężczyzna-p, kobieta-a`) jest niestabilne, gdyż `mężczyzna-p` preferuje `kobietę-b` bardziej niż swą żonę `kobietę-a`, `kobieta-b` zaś preferuje `mężczyznę-p` bardziej niż swego męża `mężczyznę-q`.

Zbiór małżeństw jest stabilny, jeżeli nie zawiera niestabilnych par.

Rozpatrzmy przykład z trzema kobietami (`kobieta_1`, `kobieta_2` i `kobieta_3`) oraz trzema mężczyznami (`mężczyzna_1`, `mężczyzna_2` i `mężczyzna_r`), których preferencje pokazano w tablicach 4.4 and 4.5. Jak widać, kolejność osób w rzędach obydwu tabel odpowiada malejącej preferencji: np. `kobieta_1` preferuje

Kobiety	Wysoka preferencja...Niska preferencja		
`kobieta_1`	`mężczyzna_2`	`mężczyzna_1`	`mężczyzna_3`
`kobieta_2`	`mężczyzna_3`	`mężczyzna_2`	`mężczyzna_1`
`kobieta_3`	`mężczyzna_1`	`mężczyzna_3`	`mężczyzna_2`

Tablica 4.4: Ranking mężczyzn wykonany przez kobiety

Mężczyźni	Wysoka preferencja...Niska preferencja		
`mężczyzna_1`	`kobieta_2`	`kobieta_1`	`kobieta_3`
`mężczyzna_2`	`kobieta_3`	`kobieta_2`	`kobieta_1`
`mężczyzna_3`	`kobieta_1`	`kobieta_3`	`kobieta_2`

Tablica 4.5: Ranking kobiet wykonany przez mężczyzn

najwyżej `mężczyznę_2`, jej drugim wyborem jest `mężczyzna_2`, a najniżej preferuje `mężczyzna_3`.
Intuicyjnie rzecz biorąc, dla problemu tego istnieją trzy zbiory stabilnych małżeństw:

1. Mężczyźni żenią się z najwyżej preferowanymi kobietami, kobiety wychodzą za mąż za najniżej preferowanych mężczyzn: `mężczyzna_1-kobieta_2`, `mężczyzna_2-kobieta_3`, `mężczyzna_3-kobieta_1`.

2. Kobiety wychodzą za mąż za najwyżej preferowanych mężczyzn, mężczyźni żenią się z najniżej preferowanymi kobietami: `mężczyzna_2-kobieta_1`, `mężczyzna_3-kobieta_2`, `mężczyzna_1-kobieta_3`.

3. Wszyscy małżonkowie mają partnerów średnio preferowanych: `mężczyzna_1 -kobieta_1`, `mężczyzna_2-kobieta_2`, `mężczyzna_3-kobieta_3`.

Dane zawarte w tablicach 4.4 i 4.5 należy w programie CLP-owym przedstawić w trochę mniej czytelnej postaci, mianowicie następującej:

```
   problem(1,
% 1=mężczyzna_1, 2=mężczyzna_2,   3=mężczyzna_3:
 /*kobieta_1:*/    []([](2, 1, 3),   % ranking mężczyzn  wykonany przez kobiety =
 /*kobieta_2:*/       [](3, 2, 1),   % RankPrzezKobiety
 /*kobieta_3:*/       [](1, 3, 2)),

% 1=kobieta_1, 2=kobieta_2, 3=kobieta_3
 /*mężczyzna_1:*/  []([](2, 1, 3),   % ranking kobiet wykonany przez mężczyzn =
 /*mężczyzna_2:*/     [](3, 2, 1),   % RankPrzezMezczyzn
 /*mężczyzna_3:*/     [](1, 3, 2))).
```

Problem wyznaczania zbiorów stabilnych małżeństw można rozwiązać za pomocą krótkiego lecz stosunkowo wymyślnego programu `4_23_stabilne` autorstwa Kjellerstranda ([Kjellerstrand-13]). Program ten wywołuje niestosowaną do tej pory bibliotekę *Propria* udostępniającą pewne zaawansowane techniki propagacji. Pominięcie tej biblioteki skutkuje komunikatem *instantiation fault* przy wczytywaniu danych. Program ten (`4_23_stabilne.ecl`) jest następujący (Maz = Mąż, Zona=Żona):

```
/*1*/  :-lib(ic).
/*2*/  :-lib(ic_global).
/*3*/  :-lib(ic_search).
/*4*/  :-lib(propia).

/*5*/  top :-
/*6*/     wszystkie_stabilne_pary(0),
/*7*/     wszystkie_stabilne_pary(1),
/*8*/     wszystkie_stabilne_pary(2),
/*9*/     wszystkie_stabilne_pary(3),
/*10*/    wszystkie_stabilne_pary(4),
/*11*/    wszystkie_stabilne_pary(5).

/*12*/ wszystkie_stabilne_pary(Problem) :-
/*13*/    printf("\nProblem %d\n", [Problem]), write("Stabline malzenstwa: "),
/*14*/    findall([Maz,Zona], stabilna_para(Problem,Maz,Zona),L),

   % Na odpowiadających sobie pozycjach list  'maz' i 'zona'
   %  są stabilne małżeństwa:
/*15*/    ( foreach([Maz,Zona], L) do
/*16*/    write("maz: "),write(Maz),nl,
/*17*/    write("zona   : "),write(Zona),nl,nl
/*18*/    ).
```

```
/*19*/ stabilna_para(Problem,maz,zona) :-
/*20*/    problem(Problem, RankPrzezKobiety,RankPrzezMezczyzn),

/*21*/    dim(RankPrzezKobiety,[LiczbaKobiet, LiczbaMezczyzn]),
/*22*/    dim(RankPrzezMezczyzn,[LiczbaMezczyzn, LiczbaKobiet]),

/*23*/    dim(zona,[LiczbaMezczyzn]),
/*24*/    Zona #:: 1..LiczbaKobiet,

/*25*/    dim(maz,[LiczbaKobiet]),
/*26*/    Maz #:: 1..LiczbaMezczyzn,

/*27*    ic_global: alldifferent(Zona),
/*28*    ic_global: alldifferent(Maz),

   % Rankingi są testowane dla wszystkich możliwych par mężczyzn i kobiet:
   % Jeżeli dla każdego mężczyzny M ze zbioru mężczyzn, który  preferuje jakąś
   %  kobietę postronną Kp ze zbioru kobiet bardziej niż swą żonę K,
   %  kobieta postronna Kp preferuje swojego męża bardziej niż mężczyznę M:

/*29*/    ( for(M,1,LiczbaMezczyzn) * for(Kp,1,LiczbaKobiet),
/*30*/     param(RankPrzezMezczyzn,RankPrzezKobiety,Zona,Maz) do
/*31*/    (RankPrzezMezczyzn[M,Kp] #< RankPrzezMezczyzn[M, Zona[M]]) =>
/*32*/    (RankPrzezKobiety[KP,maz[Kp]] #< RankPrzezKobiety[Kp,M])
/*33*/    ),

   %  i jeżeli dla każdej kobiety K ze zbioru kobiet, która  preferuje jakiegoś
   %  mężczyznę postronnego Mp ze zbioru mężczyzn bardziej niż swojego męża M,
   %  mężczyzna postronny Mp preferuje swoją żonę bardziej niż kobietę K:

/*34*/    ( for(W,1,LiczbaKobiet) * for(Mp,1,LiczbaMezczyzn),
/*35*/     param(RankPrzezMezczyzn,RankPrzezKobiety,Zona,Maz) do
/*36*/    (RankPrzezKobiety[W,Mp] #< RankPrzezKobiety[W,Maz[W]]) =>
/*37*/    (RankPrzezMezczyzn[Mp,Zona[Mp]] #< RankPrzezMezczyzn[Mp,W])
/*38*/    ),

   %   to małżeństwa są stabilne.

    % Pary Żona-Mąż są tworzone dla tych samych pozycji list  Maz i Zona: :
/*39*/    ( for(Z,1,LiczbaKobiet), param(Maz, Zona) do
/*40*/    Zona[Maz[Z]] #= Z
/*41*/    ),

   % Pary Mąż-Żona są tworzone dla tych samych pozycji list Maz i Zona:
/*42*/    ( for(M,1,LiczbaMezczyzn), param(Maz, Zona) do
/*43*/     Maz[Zona[M]] #= M
/*44*/    ),
```

```
   % spłaszczenie listy list [zona,maz] dla celów  labelingu:
/*45*/    term_variables([Zona,Maz],Vars),
/*46*/    labeling(Vars).

   problem(0,
   [](1, 2),   % RankPrzezKobiety
      [](1, 2)),

   [](2, 1),   % RankPrzezMezczyzn
      [](2, 1))).
```

Problem z [Wikipedia-13]:

```
   problem(1,
   [](2, 1, 3),   %  RankPrzezKobiety
      [](3, 2, 1),
      [](1, 3, 2)),

   [](2, 1, 3),   % RankPrzezMezczyzn
      [](3, 2, 1),
      [](1, 3, 2))).
```

Problem z [van Hentenryck-99]:

```
   problem(2,
   [](1, 2, 4, 3, 5),   % RankPrzezKobiety
       [](3, 5, 1, 2, 4),
       [](5, 4, 2, 1, 3),
       [](1, 3, 5, 4, 2),
       [](4, 2, 3, 5, 1)),

   [](5, 1, 2, 4, 3),   % RankPrzezMezczyzn
       [](4, 1, 3, 2, 5),
       [](5, 3, 2, 4, 1),
       [](1, 5, 4, 3, 2),
       [](4, 3, 2, 1, 5))).
```

Problem z [Kjellerstrand-13]:

```
   problem(3,
   [](7, 3, 8, 9, 6, 4, 2, 1, 5),   % RankPrzezKobiety
       [](5, 4, 8, 3, 1, 2, 6, 7, 9),
       [](4, 8, 3, 9, 7, 5, 6, 1, 2),
       [](9, 7, 4, 2, 5, 8, 3, 1, 6),
       [](2, 6, 4, 9, 8, 7, 5, 1, 3),
       [](2, 7, 8, 6, 5, 3, 4, 1, 9),
       [](1, 6, 2, 3, 8, 5, 4, 9, 7),
       [](5, 6, 9, 1, 2, 8, 4, 3, 7),
       [](6, 1, 4, 7, 5, 8, 3, 9, 2)),

   [](3, 1, 5, 2, 8, 7, 6, 9, 4),   % RankPrzezMezczyzn
       [](9, 4, 8, 1, 7, 6, 3, 2, 5),
```

```
    [](3, 1, 8, 9, 5, 4, 2, 6, 7),
    [](8, 7, 5, 3, 2, 6, 4, 9, 1),
    [](6, 9, 2, 5, 1, 4, 7, 3, 8),
    [](2, 4, 5, 1, 6, 8, 3, 9, 7),
    [](9, 3, 8, 2, 7, 5, 4, 6, 1),
    [](6, 3, 2, 1, 8, 4, 5, 9, 7),
    [](8, 2, 6, 4, 9, 1, 3, 7, 5))).
```

Problem z [Hunt-13]:

```
problem(4,
[]([](1,2,3,4),   % RankPrzezKobiety
    [](4,3,2,1),
    [](1,2,3,4),
    [](3,4,1,2)),

[]([](1,2,3,4),   % RankPrzezMezczyzn
    [](2,1,3,4),
    [](1,4,3,2),
    [](4,3,1,2))).
```

Problem z [Ahriz-13]:

```
problem(5,
[]([](1,5,4,6,2,3),   %RankPrzezKobiety
    [](4,1,5,2,6,3),
    [](6,4,2,1,5,3),
    [](1,5,2,4,3,6),
    [](4,2,1,5,6,3),
    [](2,6,3,5,1,4)),

[]([](1,4,2,5,6,3),   % RankPrzezMezczyzn
    [](3,4,6,1,5,2),
    [](1,6,4,2,3,5),
    [](6,5,3,4,2,1),
    [](3,1,2,4,5,6),
    [](2,3,1,6,5,4))).
```

Rozwiązania są następujące:

Problem 0:

```
maz:  [](2, 1)
zona: [](2, 1)
```

Problem 1:

```
maz:  [](2, 3, 1)
zona: [](3, 1, 2)

maz:  [](3, 1, 2)
zona: [](2, 3, 1)
```

```
maz:  [](1, 2, 3)
zona: [](1, 2, 3)
```

Problem 2:

```
maz:  [](4, 1, 2, 5, 3)
zona: [](2, 3, 5, 1, 4)

maz:  [](2, 1, 4, 5, 3)
zona: [](2, 1, 5, 3, 4)
```

```
maz:  [](2, 3, 4, 1, 5)
zona: [](4, 1, 2, 3, 5)
```

Problem 3:

```
maz:  [](7, 5, 9, 8, 3, 6, 1, 4, 2)
zona: [](7, 9, 5, 8, 2, 6, 1, 4, 3)

maz:  [](6, 5, 9, 8, 3, 7, 1, 4, 2)
zona: [](7, 9, 5, 8, 2, 1, 6, 4, 3)

maz:  [](6, 4, 9, 8, 3, 7, 1, 5, 2)
zona: [](7, 9, 5, 2, 8, 1, 6, 4, 3)
```

```
maz:  [](6, 1, 4, 8, 5, 9, 3, 2, 7)
zona: [](2, 8, 7, 3, 5, 1, 9, 4, 6)

maz:  [](6, 4, 1, 8, 5, 7, 3, 2, 9)
zona: [](3, 8, 7, 2, 5, 1, 6, 4, 9)

maz:  [](6, 1, 4, 8, 5, 7, 3, 2, 9)
zona: [](2, 8, 7, 3, 5, 1, 6, 4, 9)
```

Problem 4:

```
maz:  [](1, 4, 2, 3)
zona: [](1, 3, 4, 2)
```

```
maz:  [](1, 2, 4, 3)
zona: [](1, 2, 4, 3)
```

Problem 5:

```
maz:  [](1, 2, 6, 3, 5, 4)
zona: [](1, 2, 4, 6, 5, 3)

maz:  [](1, 2, 6, 3, 4, 5)
zona: [](1, 2, 4, 5, 6, 3)
```

```
maz:  [](1, 2, 4, 3, 6, 5)
zona: [](1, 2, 4, 3, 6, 5)
```

4.8 Kombinatoryczne szeregowanie

Kombinatoryczne szeregowanie polega na wyznaczeniu kolejności elementów pewnego zbioru w sposób spełniający *ograniczenia sąsiedztwa* tych elementów. Ograniczeniami sąsiedztwa nazywa się ograniczenia określające bliskie lub dalsze sąsiedztwo elementów, np. który element ma stać przy którym i z której strony, bezpośrednio lub w dalszej kolejności.

4.8.1 Szeregowanie karoserii samochodowych

Przykład ten jest okazją przedstawienia zastosowań ważnych predykatów globalnych:

1. Predykatu:

   ```
   sequence_total(+Min, +Max, +Low, +High, +K, +Vars, ++Values)
   ```

   gdzie liczba elementów listy różnych liczb całkowitych 'Values' jest zawarta pomiędzy nieujemną liczba całkowitą 'Low' i dodatnią liczba całkowitą 'High' dla wszystkich sekwencji 'K' liczb z listy liczb całkowitych 'Vars', a całkowita liczba wystąpień każdej liczby w liście 'Vars' jest zawarta pomiędzy nieujemną liczba całkowitą 'Min' i dodatnią liczbą całkowitą 'Max'. "Dziwność" tego predykatu wynika z jego przeznaczenia, którym jest generowanie sekwencji modelujących procesy na liniach montażowych.

2. Predykatu:

   ```
   occurrences(++Wartość, +Lista, ?N)
   ```

   który jest spełniony, jeżeli `Wartość` wystąpi `N` razy w liście `Lista`.

Szeregowanie polega na określeniu kolejności pewnych elementów, np. karoserii samochodow na linii montażowej na podstawie znajomości planu produkcji (tzn. zbioru samochodów, które należy w danym okresie - np. na zmianie - wykonać). ECL^iPS^e udostępnia szereg predykatów globalnych umożliwiających skuteczne rozwiązywanie problemów szeregowania. Rozpatrzmy następujący przykład:

W montowni samochodów samochody są na linii montażowej przesuwane przez kolejne stanowiska.

Na każdym ze stanowisk wykonuje się określony montaż, np. montowanie silnika, układu kierowniczego, okien dachowych, zmieniaczy CD, napędów foteli, automatycznych przekładni, asystentów parkowania itd.

W momencie gdy samochód pojawia sie na stanowisku, zespół monterów z tego stanowiska przesuwa się wraz z nim wykonując odpowiednie prace.

Prędkość linii montażowej jest dobierana tak, by monterzy zdążyli wykonać swe prace w czasie pobytu samochodu na stanowisku.

Np. jeżeli montaż napędów foteli wymaga 16 minut a nowa karoseria samochodowa pojawia się na linii produkcyjnej co 4 minuty, to (zakładając, że każdy model samochodu będzie miał napędy foteli) stanowisko montażu napędów foteli powinno mieć możliwość wykonania montażu dla $16/4 = 4$ samochodów.

Ponieważ jednak nie każdy model samochodu wymaga napędów foteli, możliwość wykonania montażu tych napędów jest mniejsza, co wynika z chęci zmniejszenia kosztów oprzyrządowania linii montażowej i kosztów robocizny. Jeżeli np. montaż napędów foteli jest wykonywany tylko przez 3 zespoły monterów, to na stanowisku montażu napędów foteli mogą się pojawiać conajwyżej 3 samochody wymagające takiego montażu z pośród kolejnych 4 montowanych samochodów. Gdyby tych samochodów było więcej, monterzy nie nadążyliby z montażem przed wysunięciem przez linię montażową nadmiarowych samochodów ze stanowiska montażu. Stosunek 3/4 nazywa się *ograniczeniem wydajności* stanowiska montażu napędów foteli. Ograniczenia jakościowe są wbudowane w linię montażową (w postaci oprzyrządowania stanowisk) i "wbudowane" w liczebność i kwalifikacje monterów (w postaci liczby monterów przeszkolonych do wykonywania określonego montażu). Efektywne wykorzystanie mocy produkcyjnej linii montażowej wymaga spełnienia wszystkich ograniczeń wydajnościowych. Zadaniem planisty linii montażowej będzie więc zapewnienie takiej sekwencji karoserii na linii, która spełnia wszystkie ograniczenia wydajnościowe. Sekwencja karoserii spełniająca wszystkie ograniczenia wydajnościowe będzie nazywana *sekwencją dopuszczalną*. Oczywiście, nie dla każdego programu produkcji istnieje sekwencja dopuszczalna: sytuacja taka wymaga zatrzymania linii produkcyjnej dla wykonania wszystkich czynności na przeciążonych stanowiskach. Może również zdarzyć się (jak w poniższym przykładzie), że liczba możliwych sekwencji dopuszczalnych jest ogromna.

Załóżmy, że linia montażowa służy do montażu 4 różnych modeli samochodów. Ponieważ modele różnią się montowanymi opcjami, nie każdy samochód wymaga montażu na każdym stanowisku.

Tabela 5.7 przedstawia plan produkcji 4 modeli (1, 2, 3 i 4) z zaznaczonymi opcjami i ograniczeniami wydajności.

Opcja	Ograniczenie wydajności	Modele			
		1	2	3	4
Okno dachowe	3/5	-	×	×	-
Zmieniacz CD	4/5	×	-	-	×
Automatyczna przekładnia	4/5	×	×	-	×
Napędy foteli	3/4	×	×	×	-
Asystent parkowania	1/2	×	-	×	-
Liczba produkowanych modeli		30	30	30	30

Tablica 4.6: Ograniczenia wydajnościowe dla przykładowego programu produkcji samochodów: x - opcja wymagana, - - opcja nie wymagana

Pojęcie *ograniczenia wydajnościowego* jest zilustrowane dla napędu foteli przykładem z rysunku 4.5.

Rysunek 4.5: Spełnienie ograniczenia wydajnościowego dla napędów foteli

Poniższy program `4_24_montowanie_samochodow.ecl` wyznacza jedną (z ogromnej liczby możliwych) sekwencję dopuszczalną dla 120 karoserii wprowadzanych na linie montażową. Program jest krótki i przejrzysty dzięki stosowaniu dwóch "mocnych" predykatów globalnych: `occurrences/3` i `sequence_total/7`:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(ic_global).

/*3*/  top:-
/*4*/    length(L,120),
/*5*/    L::1..4,
 % 1 - Model 1, 2 - Model 2,  3 - Model 3, 4 - Model 4
```

```
 % Ograniczenie liczby produkowanych modeli za pomocą 'occurrences/3':
 %          occurrences(++Value, +Vars, ?N)
 %  Liczba całkowita 'Value' występuje 'N' razy w liście liczb całkowitych 'Vars'.

  % Liczba 1 występuje 30 razy w liście 'L':
/*6*/    occurrences(1, L, 30),

  % Liczba 2 występuje 30 razy w liście 'L':
/*7*/    occurrences(2, L, 30),

  % Liczba 3 występuje 30 razy w liście 'L':
/*8*/    occurrences(3, L, 30),

  % Liczba 4 występuje 30 razy w liście 'L':
/*9*/    occurrences(4, L, 30),

 %  Ograniczenie  wydajności stanowisk za pomocą  'sequence_total/7':
 %        sequence_total(+Min, +Max, +Low, +High, +K, +Vars, ++Values)
 %  Liczba elementów listy liczb całkowitych 'Values' jest zawarta
 %  pomiędzy 'Low' i 'High' dla wszystkich sekwencji  'K' elementów listy liczb
 % całkowitych 'Vars', a całkowita liczba wystąpień każdego elementu w liscie
 % 'Vars' jest zawarta pomiędzy 'Min' i 'Max'.

 % Okno dachowe - co najmniej żadna a conajwyżej 3 z kolejnych 5 liczb z listy L
 % są liczbami z listy [2,3]; co najmniej 60 i co najwyżej 60 liczb w liście  L
 % są liczbami z listy [2,3]:
/*10*/   sequence_total(60, 60, 0, 3, 5, L, [2,3]),

 % Zmieniacz CD - co najmniej żadna a conajwyżej 4 z kolejnych 5 liczb z listy L
 % są liczbami z listy [1,3,4]; co najmniej 90 i co najwyżej 90 liczb w liście L
 % są liczbami z listy [1,3,4]:
/*11*/   sequence_total(90, 90, 0, 4,  5, L, [1,3,4]),

 % Automatyczna przekładnia - co najmniej żadna a conajwyżej 4  z kolejnych
 % 5 liczb z listy L są liczbami z listy [1,2,4]; co najmniej 90 i co najwyżej 90
 % liczb w liście  L są liczbami z listy [1,2,4]:
/*12*/   sequence_total(90, 90,0, 4,  5, L, [1,2,4]),

 % Napędy foteli - co najmniej żadna a conajwyżej 3  z kolejnych 4 liczb
 % z listy L są liczbami z listy [1,2,3]; co najmniej 90 ai co najwyżej 90 liczb
 % w liście L są  liczbami z listy [1,2,3]:
/*13*/   sequence_total( 90, 90, 0, 3,4, L, [1,2,3]),

 % Asystent parkowania - co najmniej żadna a conajwyżej 1 z kolejnych 2 liczb
 % z listy L są liczbami z listy [1,3]; co najmniej 60 i co najwyżej 60 liczb w
 % liście L są liczbami z listy [1,3]:
/*14*/   sequence_total( 60, 60, 0, 1, 2, L, [1,3]),
```

```
/*15*/   labeling(L),
/*16*/   write_list(L).

/*17*/ write_list([H|T]):-
/*18*/   write(H),write(", "),
/*19*/   write_list(T).
/*20*/   write_list([_]).
```

Rozwiązaniem jest:

```
  L=[
 1, 2, 1, 4, 3, 2, 1, 4, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1,
 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1,
 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1,
 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1,
 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1,
 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 3]
```

Aby np. sprawdzić spełnienie ograniczenia wydajnościowego dla napędów foteli, należy testować każdą sekwencje kolejnych 4 karoserii:

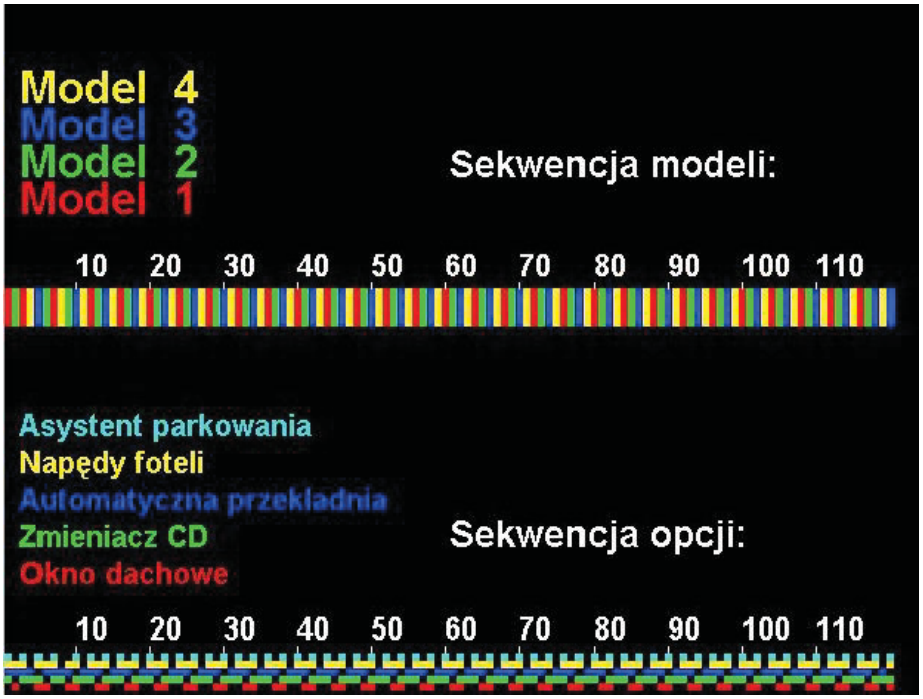
```
 1, 2, 1, 4,  W  porządku!
 2, 1, 4, 3,  W  porządku!
 1, 4, 3, 2,  W  porządku!
```

itd., dla każdego ograniczenia wydajnościowego. Zajmie to trochę czasu. Dlatego dobrze jest przedstawić sekwencje modeli i sekwencje opcji w postaci wykresu z rysunku 4.6, gdzie ograniczenia wydajnościowe przedstawiono różnokolorowymi blokami.

4.8.2 Szeregowanie na szpikulcu - szaszłyk Boba

Przykład ten jest kolejną okazją przedstawienia zastosowań ważnego predykatu globalnego `occurrences(++Wartość, +Lista, ?N)`. Przykład ten (znany w literaturze pod nazwą *Bob's Shish Kebab*) słynie jako przykład bardzo trudny (pięć gwiazdek w klasyfikacji Edmunda, zob. [Edmund-98]). Przykład jest następujący:

Bob i Patty zaprosili swych przyjaciół Javiera i Marię na szaszłyk. Dostępne były grilowane marynowane kostki wołowiny i cztery rodzaje jarzyn - grzyby, cebula, papryka i pomidory - które trzeba było nadziewać na szpikulce. Wszyscy uczestnicy spotkania mieli na swych szpikulcach po trzy kostki wołowiny i



Rysunek 4.6: Dopuszczalna sekwencja karoserii na linii montażowej

trzy różne - z wymienionych czterech - jarzyny. Każdy uczestnik spotkania nie przepadał za jedną z wymienionych czterech jarzyn i nie umieścił jej na swym szpikulcu. Nadzienia na szpikulcach będą numerowane od 1 (przy rączce) do 6 (przy ostrzu). Napisz program wyjaśniający kto co miał na szpikulcu, jeżeli wiadomo, że:
1. Na żadnym szpikulcu nie było dwóch kawałków wołowiny obok siebie.
2. Na żadnym szpikulcu wołowina nie była w tych samych trzech miejscach.
3. Na jednym szpikulcu pierwsze trzy elementy (o numerach odpowiednio 1, 2, i 3) to wołowina, papryka i grzyby; ten szpikulec nie był szpikulcem Javiera.
4. Na jednym szpikulcu były kostki wołowiny na miejscach 1, 3 i 5, a pomidor na miejscu 6.
5. Bob, który lubi cebulę i umieścił ją na swym szpikulcu, ma dwie inne jarzyny

na miejscach 4 i 5.
6. Na czterech szpikulcach na miejscu 5 była wołowina, grzyby, cebula i pomidor.
7. Każdy kawałek cebuli był bezpośrednio pomiędzy dwiema kostkami wołowiny.
8. Papryka nie była nigdzie bezpośrednio pomiędzy dwiema kostkami wołowiny.
9. Maria nie lubi grzybów i nie umieściła ich na swym szpikulcu.
10.Co najmniej dwa szpikulce miały tą samą jarzynę w tym samym miejscu co najmniej raz.

Przykład ten zawiera sporo *negatywnych* reguł, tzn. reguł stwierdzających, że coś nie jest prawdą. Reguły takie są najczęściej źródłem kłopotów. Jak sobie z tym poradzić pokazuje program `4_25_szaszlyk_Boba.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  :-lib(ic_global).

/*3*/  top:-

  %Bi - element na miejscu i-tym szpikulca Boba
  %Pi - element na miejscu i-tym szpikulca Pati
  %Ji - element na miejscu i-tym szpikulca Javiera
  %Mi - element na miejscu i-tym szpikulca Marii
  %   1-grzyby, 2-papryka, 3-cebula, 4-pomidor, 5-wołowina

/*4*/       Bob=[B1,B2,B3,B4,B5,B6,B7],
/*5*/       Bob::1..5,
/*6*/       Patty=[P1,P2,P3,P4,P5,P6,P7],
/*7*/       Patty::1..5,
/*8*/       Javier=[J1,J2,J3,J4,J5,J6,J7],
/*9*/       Javier::1..5,
/*10*/      Maria=[M1,M2,M3,M4,M5,M6,M7],
/*11*/      Maria::1..5,

  %    Np.  J3 = 5 oznacza, że Javier ma wołowinę na miejscu 3 swego szpikulca.

  %   Zakłada się dla uproszczenia notacji, że miejsce 7 na każdym szpikulcu
  %   jest "rezerwowane" dla nielubianego warzywa, pominiętego przez każdego z
  %   uczestników spotkania przy kompletowaniu szpikulca:
/*12*/      [B7,P7,J7,M7]::1..4,
/*13*/      ic_global: alldifferent([B7,P7,J7,M7]),

  %    Ograniczenie 1 - na żadnym szpikulcu nie było dwóch kawałków wołowiny
  %    obok siebie:
/*14*/      ograniczenie_1([Bob,Patty,Javier,Maria]),
```

```
  %    Ograniczenie 2 - na żadnym szpikulcu wołowina nie była w tych samych
  %    trzech miejscach:
/*15*/      ograniczenie_2([Bob,Patty,Javier,Maria]),

  %    Ograniczenie 3 - na jednym szpikulcu pierwsze trzy elementy (o numerach
  %    odpowiednio 1, 2 i 3) to  wołowina, papryka i grzyby; ten szpikulec nie
  %    był szpikulcem Javiera:
/*16*/      ograniczenie_3(Bob,Patty,Maria),

  %    Ograniczenie 4 - na jednym szpikulcu były kostki wołowiny na miejscach
  %    1, 3 i 5, a pomidor na miejscu 6:
/*17*/      ograniczenie_4(Bob,Patty,Javier,Maria),

  %    Ograniczenie 5 -  Bob, który lubi cebulę i umieścił ją na swym szpikulcu,
  %    ma dwie inne jarzyny na miejscach 4 i 5:
/*18*/      ograniczenie_5([_,_,_,B4,B5,_,_]),

  %    Ograniczenie 6 - na czterech szpikulcach na miejscu 5 była  wołowina,
  %    grzyby, cebula i pomidor:
/*19*/      ograniczenie_6(Bob,Patty,Javier,Maria),

  %    Ograniczenie 7 - każdy kawałek cebuli był bezpośrednio pomiędzy
  %    dwiema kostkami wołowiny:
/*20*/      ograniczenie_7([Bob,Patty,Javier,Maria]),

  %    Ograniczenie 8 - papryka nie była nigdy bezpośrednio pomiędzy
  %    dwiema kostkami wołowiny:
/*21*/      ograniczenie_8([Bob,Patty,Javier,Maria]),

  %    Ograniczenie 9 - Maria nie lubi grzybów i nie umieściła  ich na swym
  %    szpikulcu:
/*22*/      ograniczenie_9(Maria),

  %    Ograniczenie 10 - co najmniej dwa szpikulce miały tę samą jarzynę
  %    w tym samym miejscu co najmniej raz:
/*23*/      ograniczenie_10([Bob,Patty,Javier,Maria]),

  %     Każdy szpikulec miał 3 kostki wołowiny:
/*24*/      occurrences(5, [B1,B2,B3,B4,B5,B6], 3),
/*25*/      occurrences(5, [P1,P2,P3,P4,P5,P6], 3),
/*26*/      occurrences(5, [J1,J2,J3,J4,J5,J6], 3),
/*27*/      occurrences(5, [M1,M2,M3,M4,M5,M6], 3),

  %     Każdy szpikulce miał po kawałku z trzech jarzyn,
  %     czwarta jarzyna została pominięta, tzn. znalazła się na miejscu 7:
/*28*/      occurrences(1, [B1,B2,B3,B4,B5,B6,B7], 1),
/*29*/      occurrences(1, [P1,P2,P3,P4,P5,P6,P7], 1),
/*30*/      occurrences(1, [J1,J2,J3,J4,J5,J6,J7], 1),
```

```
/*31*/      occurrences(1, [M1,M2,M3,M4,M5,M6,M7], 1),

/*32*/      occurrences(2, [B1,B2,B3,B4,B5,B6,B7], 1),
/*33*/      occurrences(2, [P1,P2,P3,P4,P5,P6,P7], 1),
/*34*/      occurrences(2, [J1,J2,J3,J4,J5,J6,J7], 1),
/*35*/      occurrences(2, [M1,M2,M3,M4,M5,M6,M7], 1),

/*36*/      occurrences(3, [B1,B2,B3,B4,B5,B6,B7], 1),
/*37*/      occurrences(3, [P1,P2,P3,P4,P5,P6,P7], 1),
/*38*/      occurrences(3, [J1,J2,J3,J4,J5,J6,J7], 1),
/*39*/      occurrences(3, [M1,M2,M3,M4,M5,M6,M7], 1),

/*40*/      occurrences(4, [B1,B2,B3,B4,B5,B6,B7], 1),
/*41*/      occurrences(4, [P1,P2,P3,P4,P5,P6,P7], 1),
/*42*/      occurrences(4, [J1,J2,J3,J4,J5,J6,J7], 1),
/*43*/      occurrences(4, [M1,M2,M3,M4,M5,M6,M7], 1),

/*44*/      labeling([B1,B2,B3,B4,B5,B6,B7,P1,P2,P3,P4,P5,P6,P7,
                J1,J2,J3,J4,J5,J6,J7,M1,M2,M3,M4,M5,M6,M7]),

/*45*/      write("Szpikulec Boba:    "),przepisz(B1),write("  "),przepisz(B2),
                write("  "), przepisz(B3),write("  "), przepisz(B4),
                write("  "),przepisz(B5),write("  "), przepisz(B6),nl,

/*46*/      write("Szpikulec Patty:   "),przepisz(P1),write("  "),przepisz(P2),
                write("  "),przepisz(P3),write("  "),przepisz(P4),
                write("  "),przepisz(P5),write("  "),przepisz(P6),nl,

/*47*/      write("Szpikulec Javiera: "),przepisz(J1),write("  "),przepisz(J2),
                write("  "),przepisz(J3),write("  "),przepisz(J4),
                write("  "),przepisz(J5),write("  "),przepisz(J6),nl,

/*48*/      write("Szpikulec Marii:   "),przepisz(M1),write("  "),przepisz(M2),
                write("  "),przepisz(M3),write("  "),przepisz(M4),
                write("  "),przepisz(M5),write("  "),przepisz(M6),nl,nl.

/*49*/  przepisz(1):-write("grzyby").
/*50*/  przepisz(2):-write("papryka  ").
/*51*/  przepisz(3):-write("cebula   ").
/*52*/  przepisz(4):-write("pomidor  ").
/*53*/  przepisz(5):-write("wolowina    ").

  %    Ograniczenie 1 - na żadnym szpikulcu nie było dwóch kawałków wołowiny obok[2]
  %    siebie:
/*54*/  ograniczenie_1([H|T]):-
/*55*/      testuj_1(H),
```

[2]Czytelnik wybaczy powtarzanie werbalizacji ograniczeń, ale ów nadmiar ułatwia zrozumienie programu.

```
/*56*/      ograniczenie_1(T).
/*57*/  ograniczenie_1([]).

/*58*/  testuj_1([A,B,C,D,E,F,_]):-
/*59*/      ~dwa_kawalki_wolowiny_obok_siebie(A,B),
/*60*/      ~dwa_kawalki_wolowiny_obok_siebie(B,C),
/*61*/      ~dwa_kawalki_wolowiny_obok_siebie(C,D),
/*62*/      ~dwa_kawalki_wolowiny_obok_siebie(D,E),
/*63*/      ~dwa_kawalki_wolowiny_obok_siebie(E,F).

/*64*/  dwa_kawalki_wolowiny_obok_siebie(X,Y):-
/*65*/      X#=5,Y#=5.

  %    Ograniczenie 2 - na żadnym szpikulcu wołowina nie była w tych samych trzech
  %    miejscach:
/*66*/  ograniczenie_2([Bob,Patty,Javier,Maria]):-
/*67*/      testuj_2(Bob,Patty),testuj_2(Bob,Javier),testuj_2(Bob,Maria),
/*68*/      testuj_2(Patty,Javier),testuj_2(Patty,Maria),testuj_2(Javier,Maria).

/*69*/  testuj_2([B1,B2,B3,B4,B5,B6,_],[P1,P2,P3,P4,P5,P6,_]):-
/*70*/      ~w_tych_samych_miejscach(B1,B3,B5,P1,P3,P5),
/*71*/      ~w_tych_samych_miejscach(B1,B3,B6,P1,P3,P6),
/*72*/      ~w_tych_samych_miejscach(B1,B4,B6,P1,P4,P6),
/*73*/      ~w_tych_samych_miejscach(B2,B4,B6,P2,P4,P6).

/*74*/  w_tych_samych_miejscach(X1,X2,X3,Y1,Y2,Y3):-
/*75*/      X1#=5,
/*76*/      X2#=5,
/*77*/      X3#=5,
/*78*/      Y1#=X1,
/*79*/      Y2#=X2,
/*80*/      Y3#=X3.

  %    Ograniczenie 3 - na jednym szpikulcu pierwsze trzy elementy (o numerach
  %    odpowiednio 1, 2 i 3) to wołowina, papryka i grzyby; ten szpikulec nie był
  %    szpikulcem Javiera:
/*81*/  ograniczenie_3([B1,B2,B3,_,_,_,_],[P1,P2,P3,_,_,_,_],[M1,M2,M3,_,_,_,_]):-
/*82*/      (
/*83*/      (B1#=5, B2#=2, B3#=1);
/*84*/      (P1#=5, P2#=2, P3#=1);
/*85*/      (M1#=5, M2#=2, M3#=1)
/*86*/      ).

  %    Ograniczenie 4 - na jednym szpikulcu były kostki wołowiny na
  %    miejscach 1, 3 i 5, a pomidor na miejscu 6:
/*87*/  ograniczenie_4(Bob,Patty,Javier,Maria):-
/*88*/      (kostki_wolowiny_i_pomidor_na_miejscach_1_3_5_6(Bob);
/*89*/      kostki_wolowiny_i_pomidor_na_miejscach_1_3_5_6(Patty);
/*90*/      kostki_wolowiny_i_pomidor_na_miejscach_1_3_5_6(Javier);
```

```
/*91*/      kostki_wolowiny_i_pomidor_na_miejscach_1_3_5_6(Maria)).

/*92*/  kostki_wolowiny_i_pomidor_na_miejscach_1_3_5_6([X1,_,X3,_,X5,X6,_]):-
/*93*/      X1#=5, X3#=5, X5#=5, X6#=4.

  %    Ograniczenie 5 - Bob, który lubi cebulę i umieścił ją na swym szpikulcu,
  %    ma dwie inne jarzyny na miejscach 4 i 5:
/*94*/  ograniczenie_5([_,_,_,B4,B5,_,_]):-
/*95*/      B4#\=5, B5#\=5, B4#\=B5, B4#\=3, B5#\=3.

  %    Ograniczenie 6 - na czterech szpikulcach, na miejscu 5
  %    była  wołowina, grzyby, cebula i pomidor:
/*96*/  ograniczenie_6([_,_,_,_,B5,_,_],[_,_,_,_,P5,_,_],[_,_,_,_,J5,_,_],
                       [_,_,_,_,M5,_,_]):-
/*97*/      (
/*98*/      (B5#=5; P5#=5; J5#=5; M5#=5),
/*99*/      (B5#=1; P5#=1; J5#=1; M5#=1),
/*100*/     (B5#=3; P5#=3; J5#=3; M5#=3),
/*101*/     (B5#=4; P5#=4; J5#=4; M5#=4)
/*102*/     ).

  %    Ograniczenie 7 - każdy kawałek cebuli był bezpośrednio pomiędzy
  %    dwiema kostkami wołowiny:
/*103*/ ograniczenie_7([Bob,Patty,Javier,Maria]):-
/*104*/     testuj_7(Bob),testuj_7(Patty),
/*105*/     testuj_7(Javier),testuj_7(Maria).

/*106*/ testuj_7([A,B,C,_,_,_,_]):-
/*107*/     A#=5,B#=3,C#=5.
/*108*/ testuj_7([_,B,C,D,_,_,_]):-
/*109*/     B#=5,C#=3,D#=5.
/*110*/ testuj_7([_,_,C,D,E,_,_]):-
/*111*/     C#=5,D#=3,E#=5.
/*112*/ testuj_7([_,_,_,D,E,F,_]):-
/*113*/     D#=5,E#=3,F#=5.
/*114*/ testuj_7([_,_,_,_,_,_,G]):-
/*115*/     G#=3.

  %    Ograniczenie 8 - papryka nie była nigdy bezpośrednio pomiędzy
  %    dwiema kostkami wołowiny:
/*116*/ ograniczenie_8([Bob,Patty,Javier,Maria]):-
/*117*/     testuj_8(Bob),testuj_8(Patty),
/*118*/     testuj_8(Javier),testuj_8(Maria).

/*119*/ testuj_8([A,B,C,D,E,F,_]):-
/*120*/     ~papryka_pomiędzy_dwoma_kostkami_wolowiny(A,B,C),
/*121*/     ~papryka_pomiędzy_dwoma_kostkami_wolowiny(B,C,D),
/*122*/     ~papryka_pomiędzy_dwoma_kostkami_wolowiny(C,D,E),
/*123*/     ~papryka_pomiędzy_dwoma_kostkami_wolowiny(D,E,F).
```

```
/*124*/ papryka_pomiędzy_dwoma_kostkami_wolowiny(X,Y,Z):-
/*125*/     X#=5,Y#=2,Z#=5.

  %    Ograniczenie 9 - Maria nie lubi grzybów i nie umieściła ich
  %    na swym szpikulcu:
/*126*/ ograniczenie_9([M1,M2,M3,M4,M5,M6,M7]):-
/*127*/     M7#=1,
/*128*/     M1#\=1, M2#\=1, M3#\=1,
/*129*/     M4#\=1, M5#\=1, M6#\=1.

  %    Ograniczenie 10 - co najmniej dwa szpikulce miały tę samą jarzynę
  %    w tym samym miejscu co najmniej raz:
/*130*/  ograniczenie_10([Bob,Patty,Javier,Maria]):-
/*131*/     (
/*132*/     testuj_10(Bob,Patty);
/*133*/     testuj_10(Bob,Javier);
/*134*/     testuj_10(Bob,Maria);
/*135*/     testuj_10(Patty,Javier);
/*136*/     testuj_10(Patty,Maria);
/*137*/     testuj_10(Javier,Maria)
/*138*/     ).

/*139*/ testuj_10([X1,X2,X3,X4,X5,X6,_],[Y1,Y2,Y3,Y4,Y5,Y6,_]):-
/*140*/     (
/*141*/     (X1#\=5,X1#=Y1);
/*142*/     (X2#\=5,X2#=Y2);
/*143*/     (X3#\=5,X3#=Y3);
/*144*/     (X4#\=5,X4#=Y4);
/*145*/     (X5#\=5,X5#=Y5);
/*146*/     (X6#\=5,X6#=Y6)
/*147*/     ).
```

Komunikat przedstawia unikatowe rozwiązanie:

```
Szpikulec Boba:    wolowina   cebula    wolowina   papryka    grzyby    wolowina
Szpikulec Patty:   wolowina   papryka   grzyby     wolowina   pomidor   wolowina
Szpikulec Javiera: wolowina   cebula    wolowina   grzyby     wolowina  pomidor
Szpikulec Marii:   papryka    wolowina  pomidor    wolowina   cebula    wolowina
```

4.8.3 Szeregowanie cykliczne - awantura kolacyjna

Często zmienne są *cykliczne*, jak np. dni tygodnia, miesiące roku, miejsca wokół okręgu. Ich obsługa wymaga pewnej uwagi, co pokazuje poniższy przykład:

Pan i Pani Davis zaprosili swych przyjaciół, Pana i Panią Astor, Pana i Panią Blake oraz Pana i Panią Crane na kolację podawaną na ich ulubionym, eleganckim sześciokątnym stole w stylu retro. Niestety, miła atmosfera przyjęcia popsuła się, gdy uczestnicy nieostrożnie poruszyli tematy związane z aktualną polityką. W trakcie ożywionej wymiany opinii:
1) Pani Astor została obrażona przez pana Blake, który siedział z jej lewej strony.
2) Pan Blake został obrażony przez panią Crane, która siedziała naprzeciwko.
3) Pani Blake została obrażona przez panią Astor, która siedział obok niej.
4) Gospodyni (pani Davis) została obrażona przez jedyną osobę siedzącą pomiędzy dwoma panami.
Wiedząc ponadto że:
5) Gospodyni była jedyną osobą siedzącą pomiędzy parą małżonków i
6) Pani Davis siedziała naprzeciwko pana Davisa,
należy określić, kto siedział gdzie i kto obraził gospodynię. Problem można rozwiązać za pomocą programu `4_26_awantura_kolacyjna.ecl`:

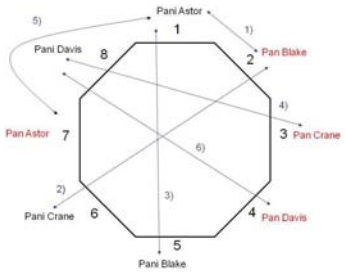
```
/*1*/   :- lib(ic).
/*2*/   top :-
/*3*/       Miejsca = [Pani_Astor, Pan_Astor, Pan_Blake, Pani_Blake,
                         Pani_Crane, Pan_Crane, Pani_Davis, Pan_Davis],
/*4*/       Miejsca :: 1..8,
 % Miejsca są ponumerowane jak to pokazano na rysunku 4.7.
 % Sens zmiennych: jeżeli np. Pan_Astor = 7, to Pan Astor siedzi na miejscu 7
 % Osobę siedzącą na jednym miejscu można ustalić:
/*5*/       Pani_Astor = 1,

 % % Każda osoba zajmuje tylko jedno miejsce:
/*6*/       alldifferent(Miejsca),

 % 1) Pani Astor została obrażona przez pana Blake'a siedzącego z jej lewej strony:
/*7*/       Pan_Blake = 2,

 % 2) Pan Blake został obrażony przez panią Crane, która siedziała naprzeciwko:
/*8*/       naprzeciw(2,Pani_Crane),

 % 3) Pani Blake siedziała naprzeciwko pani Astor
/*9*/       naprzeciw(Pani_Blake,1),

 % 4) Gospodyni została obrażona przez jedyną osobę siedzącą między dwoma panami:
/*10*/      (member(Pozycja_chama,[Pani_Blake,Pani_Crane,Pan_Crane]),
                pomiedzy(Pan_Astor,Pozycja_chama,2);
/*11*/       member(Pozycja_chama,[Pani_Blake,Pani_Crane]),
                pomiedzy(Pan_Astor,Pozycja_chama,Pan_Crane);
```

```
/*12*/       member(Pozycja_chama,[Pani_Blake,Pani_Crane,Pan_Crane,Pani_Davis]),
                 pomiedzy(Pan_Astor,Pozycja_chama,Pan_Davis);
/*13*/       member(Pozycja_chama,[Pan_Astor,Pani_Blake,Pani_Crane,Pan_Davis]),
                 pomiedzy(2,Pozycja_chama,Pan_Crane);
/*14*/       member(Pozycja_chama,[Pan_Astor,Pani_Blake,Pani_Crane,Pan_Crane]),
                 pomiedzy(2,Pozycja_chama,Pan_Davis);
/*15*/       member(Pozycja_chama,[Pan_Astor,Pani_Blake,Pani_Crane]),
                 pomiedzy(Pan_Crane,Pozycja_chama,Pan_Davis)),

 % 5) Gospodyni (Pani_Davis) była jedyną osobą siedzącą pomiedzy parą małżonków:
/*16*/      (pomiedzy(Pani_Blake, Pani_Davis, 2);
/*17*/       pomiedzy(1, Pani_Davis, Pan_Astor);
/*18*/       pomiedzy(Pani_Crane, Pani_Davis, Pan_Crane)),

 % 6) Pani Davis siedziała naprzeciwko pana Davisa:
/*19*/      naprzeciw(Pani_Davis,Pan_Davis),

/*20*/      labeling([Pani_Astor, Pan_Astor, Pan_Blake, Pani_Blake,
                      Pani_Crane, Pan_Crane, Pani_Davis, Pan_Davis]),

/*21*/      writeln([Pani_Astor, Pan_Astor, Pan_Blake, Pani_Blake,
                     Pani_Crane, Pan_Crane, Pani_Davis, Pan_Davis]),
/*22*/      Lista_nazwisk=["Pani_ Astor","Pan_ Astor","Pan_ Blake","Pani_ Blake",
                    "Pani_ Crane","Pan_ Crane","Pani_ Davis","Pan_ Davis"],
/*23*/      Lista_niejsc=[Pani_Astor, Pan_Astor, Pan_Blake, Pani_Blake,
                     Pani_Crane, Pan_Crane, Pani_Davis, Pan_Davis],
/*24*/      szukaj_chama(Lista_nazwisk,Lista_niejsc,Pozycja_chama,Nazwisko_chama),

/*25*/   write("Pani Astor siedziala na miejscu "),write("1"),writeln("."),
/*26*/   write("Pan Astor siedzial na miejscu "),write(Pan_Astor),writeln("."),
/*27*/   write("Pani Blake siedziala na miejscu "),write(Pani_Blake),writeln("."),
/*28*/   write("Pan Blake siedzial na miejscu "),write("2"),writeln("."),
/*29*/   write("Pani Crane siedziala na miejscu "),write(Pani_Crane),writeln("."),
/*30*/   write("Pan Crane siedzial na miejscu "), write(Pan_Crane),writeln("."),
/*31*/   write("Pani Davis siedziala na miejscu "),write(Pani_Davis),writeln("."),
/*32*/   write("Pan Davis siedzial na miejscu "),write(Pan_Davis),writeln("."),
/*33*/   writeln("Gospodyni (Pani Davis) została obrazona przez osobe siedzaca"),
/*34*/   write(" na miejscu "),write(Pozycja_chama),write(", ktora byl "),
/*35*/   write(Nazwisko_chama),writeln("."),nl.

/*36*/   next_to(A,B):-
/*37*/      B #= A + 1;
/*38*/      A #= B + 1.

/*39*/   obok(8,1).
/*40*/   obok(1,8).

/*41*/   pomiedzy(A,X,B):-
/*42*/      X #= A + 1,
```

```
/*43*/      X #= B - 1;
/*44*/      X #= A - 1,
/*45*/      X #= B + 1.

/*46*/   pomiedzy(7,8,1).
/*47*/   pomiedzy(1,8,7).
/*48*/   pomiedzy(8,1,2).
/*49*/   pomiedzy(2,1,8).

/*50*/   naprzeciw(A,B):-
/*51*/      B #= A + 4;
/*52*/      A #= B + 4.

/*53*/   szukaj_chama([_|T_nazwiska],[H_pozycja|T_pozycja],X,Cham):-
/*54*/      not(X = H_pozycja),
/*55*/      szukaj_chama(T_nazwiska,T_pozycja,X,Cham).
/*56*/   szukaj_chama([H_nazwisko|_],[H_pozycja|_],X,Cham):-
/*57*/      X = H_pozycja,
/*58*/      Cham = H_nazwisko.
```

Rozwiązaniem (rysunek 4.7) jest: `[1, 7, 2, 5, 6, 3, 8, 4]`

```
Pani Astor siedziala na miejscu 1.    Pan Astor siedzial na miejscu 7.
Pani Blake siedziala na miejscu 5.    Pan Blake siedzial na miejscu 2.
Pani Crane siedziala na miejscu 6.    Pan Crane siedzial na miejscu 3.
Pani Davis siedziala na miejscu 8.    Pan Davis siedzial na miejscu 4.
Gospodyni (Pani Davis) zostala obrazona przez
osobe siedzaca na miejscu 3 ktora był Pan Crane.
```



Rysunek 4.7: Rozwiązanie dla awantury kolacyjnej

4.9 Zadania

Antyki z nefrytu [3]

Wan Li, znany sprzedawca chińskich antyków, miał ostatnio znakomity miesiąc: dokonał sprzedaży czterem klientom z całego świata - Finlandii, Włoch, Japonii i Stanów Zjednoczonych - którzy byli skłonni dużo zapłacić za upatrzone antyki. Cztery antyki były rzadkimi figurkami z nefrytu (klamra do paska, smok, konik polny i koń), każdy wyrzeźbiony z innego koloru nefrytu (ciemno-zielonego, jasno-zielonego, czerwonego i białego). Każdy przedmiot datowany był z innej chińskiej dynastii (Ching, Ming, Sung i Tang). Napisać program, który łączy każdą figurkę z jej kolorem i dynastią oraz podać kraj pochodzenia każdego z nabywców, jeśli:
1. Rzadki biały smok (którego nie kupił Amerykanin) nie pochodzi z dynastii Sung.
2. Wspaniała klamra do paska (która nie posiadała żadnego z odcieni zieleni) została wykonana w 618 r. n.e. dla cesarza dynastii Tang.
3. Trzy z figurek były: jedna kupiona przez Fina (która nie była smokiem), druga z dynastii Ching (która nie trafiła do nabywcy z Japonii) i trzecia jasno-zielona (która nie była konikiem polnym).
4. Amerykanka nie zdecydowała się ani na konika polnego ani na przedmiot z dynastii Sung, gdyż żadne z nich nie pasowało do jej domowego wystroju.

Wykłady [4]

W zeszłym tygodniu w auli szkoły odbyła się seria wykładów proszonych, jeden każdego dnia (od poniedziałku do piątku). Żaden z nich nie był wyjątkowo interesujący (wybór studiów, higiena, sztuka nowoczesna, odżywianie i metody uczenia się), ale uczniowie uznali, że wszystko co odciągnie ich od czwartej godziny lekcyjnej jest godne uwagi. Wykłady były prowadzone przez dwie wykładowczynie o imionach Alicja i Barbara, i trzech wykładowców o imionach Karol, Dariusz i Edward; nazywali się Fabrycy, Garber, Haller, Imieliński i Jankowiak. Napisz program wyznaczający dzień wykładu i przedmiot każdego wykładowcy, jeżeli wiadomo, że:
1.Alicja wykładała w poniedziałek.
2.Karol nie wykładał swojego przedmiotu (higieny) w piątek.
3.Dietetyk Imieliński wykładał o odżywianiu.

[3] Przykład z http://brownbuffalo.sourceforge.net/
[4] Przykład z http://www.f1compiler.com/default.html

4.Sztukę nowoczesną wykładał mężczyzna.
5.Pani Jankowiak i wykładowca metod uczenia się wykładali w kolejnych dniach po sobie, w takim porządku lub odwrotnie.
6.Haller wykładał jakiś czas po Edwardzie.
7.Dariusz Fabrycy wykładał jakiś czas przed wykładem o sztuce nowoczesnej.

Posiedzenie rady miejskiej

Na ostatnim posiedzeniu rady miejskiej każdy członek (pan Akerman, pani Baird, pan Chatham, pani Duval i pan Etting) głosował nad pięcioma wnioskami (o numerach od 1 do 5) w sposób następujący:
1. Każdy wniosek uzyskał inną liczbę głosów popierających.
2. Podczas całego głosowania oddano o trzy głosy popierające więcej niż głosów sprzeciwu.
3. Żadnych dwóch członków rady nie głosowało w identyczny sposób nad wszystkimi pięcioma wnioskami.
4. Dwie panie częściej głosowały różnie dla tego samego wniosku, aniżeli tak samo.
5. Pan Chatham nigdy nie zagłosował za sąsiadującymi ze sobą wnioskami.
6. Pan Akerman i Pani Baird zagłosowali za wnioskiem 4.
7. Wniosek 1 otrzymał o dwa głosy popierające więcej niż wniosek 2.
8. Wiosek 3 otrzymał dwa razy więcej głosów popierających niż wniosek 4.
Napisz program wyjaśniający, kto jak głosował.

Konkurs DJ-ów

Na ostatnim festiwalu muzycznym czterech DJ-ów uczestniczyło w konkursie miksowania. Każdy z nich miał na koszuli numer (1,2,3 lub 4), a ich konsole były różnych kolorów. DJ Skinf Lint wygrał konkurs, i tylko jeden DJ miał na koszuli numer odpowiadający pozycji, którą zajął. DJ Slam Dunk miał na koszuli numer 1. DJ z numerem 2 miał czerwoną konsolę, a DJ Jam Jar nie miał żółtej konsoli. DJ który zajął miejsce ostatnie miał niebieską konsolę. DJ Park'n Ride pokonał DJ-a Slam Dunk. DJ z numerem 1 miał zieloną konsolę, a DJ, które zajął drugie miejsce, miał numer 3. Napisz program wyjaśniający, kto zajął które miejsce, jaki miał numer na koszuli i jakiego koloru była jego konsola.

Dyplomata, polityk i szpieg

Z pośród trzech osób (Alek, Bolek i Czarek), jedna jest dyplomatą, jedna politykiem, a jedna szpiegiem[5]. Dyplomata zawsze mówi prawdę, polityk zawsze kłamie, a szpieg albo kłamie, albo mówi prawdę. Alek powiedział: "Czarek jest politykiem." Bolek powiedział: "Alek jest dyplomatą." Czarek powiedział: "Jestem szpiegiem." Napisz program wyjaśniający, kto jest dyplomatą, kto politykiem a kto szpiegiem.

Magiczny kwadrat

Dany jest kwadrat o wymiarach 3×3:
A B C
D E F
G H I
Napisz program uziemiający zmienne A,B,...,I tak, by suma elementów dla każdego wiersza, każdej kolumny i każdej przekątnej była taka sama.

Rozwiązanie:

```
2  7  6
9  5  1
4  3  8
```

Suma

Napisz program zastępujący wszystkie litery takimi cyframi, by następująca suma była poprawna:

```
   AND
    TO
   ALL
     A
  GOOD
-------
 NIGHT
```

Różne litery odpowiadają różnym cyfrom, jednakowe litery odpowiadają jednakowym cyfrom, a cyfry na najbardziej znaczących pozycjach nie mogą być zerami.

Rozwiązanie:

```
[A, N, D, T, O, L, G, I, H] = [5, 1, 3, 6, 7, 8, 9, 0, 4]
```

[5] zob. http://www.mathsisfun.com/puzzles.

Dyżury

Siedmiu lekarzy ma dyżury kolejno w ciągu całego tygodnia. Każdy ma dyżur innego dnia. Arek ma dyżur następnego dnia po dyżurze Darka. Czarek ma dyżur dzień później niż Bolek. Edek ma dyżur szóstego dnia przed dyżurem Franka. Czwartek, który jest dniem dyżuru Grzegorza, znajduje się akurat pośrodku między dniami dyżurów Czarka i Darka. Napisz program wyjaśniający, kto i kiedy ma dyżur.

Rozwiązanie:

Pon	Wt	Śr	Czw	Pią	Sob	Niedz
Edek	Bolek	Czarek	Grzegorz	Darek	Arek	Franek

Tablica 4.7: Rozkład dyżurów

Książki

Osiem zaprzyjaźnionych par małżeńskich spotkało się na kiermaszu książek. Dla zaoszczędzenia wydatków, każda para kupuje tylko jedną książkę, a mianowicie taką, którą również chętnie przeczyta inna para, i natychmiast wymieniają się kupionymi książkami. Pary mają takie samo nazwisko, zawód i samochód. Wszystkie pary mają swój ulubiony kolor. Ponadto wiadomo, że:

1. Danka Czerny i jej mąż pracują jako menedżerowie sprzedaży.
2. Książka "Lodołamacz" została zakupiona przez parę, która jeździ Fiatem i lubi kolor czerwony.
3. Piotr i jego żona Wiktoria lubią kolor brązowy.
4. Staszek Kogut i jego żona Hanna lubią kolor biały.
5. Janka Kowal i jej mąż pracują jako menedżerowie hurtowni i jeżdżą Fordem.
6. Monika i jej mąż Olek pożyczyli książkę "Stulecie kłamców".
7. Mateusz i jego żona lubią kolor różowy i zakupili książkę "Towarszysz Szmaciak".
8. Irka i jej mąż Tolek pracują jako księgowi.
9. Książka "Polactwo" została pożyczona przez parę, która jeździ Volkswagenem.
10. Państwo Piwowar są menedżerami multipleksu i zakupili książkę "Eko-

nomia w jednej lekcji".
11. Pan i Pani Stolarz są lekarzami i pożyczyli książkę „Cywilizacja bizantyńska".
12. Paweł i jego żona lubią kolor zielony.
13. Zosia Komar i jej mąż lubią kolor niebieski.
14. Rysiek z żoną kupili książkę „Cywilizacja bizantyńska" i jeżdżą Toyotą.
15. Jakaś para kupiła książkę „Rosja w roku 1839" i pożyczyła książkę „Towarszysz Szmaciak".
16. Para, która jeździ Peugotem lubi kolor fioletowy.
17. Para, która pracuje jako nauczyciele pożyczyła książkę „Rosja w roku 1839".
18. Para, która pracuje jako rolnicy jeżdżą Volvo.
19. Pamela i jej mąż jeżdżą Renault i kupili książkę „Stulecie kłamców"
20. Pamela i jej mąż pożyczyli książkę, którą kupili państwo Zając.
21. Robert i jego żona lubią kolor żółty i pożyczyli książkę „Obłęd '44".
22. Państwo Sawa pracują jako klienci-kontrolerzy.
23. Książka „Obłęd '44" została zakupiona przez parę, która jeździ Skodą.
Napisz program, który dla każdej z par określi imię i nazwisko, markę samochodu, ulubiony kolor, kupioną książkę i pożyczoną książkę.

Rozwiązanie:
Danka i Mateusz Czerny pracują jako menedżerowie sprzedaży, ich samochód to Volkswagen, lubią kolor różowy, kupili książkę "Towarszysz Szmaciak", a pożyczyli książkę "Polactwo".

Wiktoria i Piotr Stolarz pracują jako lekarze, ich samochód to Skoda, lubią kolor brązowy, kupili książkę "Obłęd '44", a pożyczyli książkę "Cywilizacja bizantyńska".

Hanna i Staszek Kogut pracują jako rolnicy, ich samochód to Volvo, lubią kolor biały, kupili książkę "Rosja w roku 1839", a pożyczyli książkę "Towarszysz Szmaciak".

Janka i Robert Kowal pracują jako menedżerowie hurtowni, ich samochód to Ford, lubią kolor żółty, kupili książkę "Polactwo", a pożyczyli książkę "Obłęd '44".

Monika i Olek Piwowar pracują jako menedżerowie multipleksu, ich samochód to Peugot, lubią kolor fioletowy, kupili książkę "Ekonomia w jednej lekcji", a pożyczyli książkę "Stulecie kłamców".

Irka i Tolek Zając pracują jako księgowi, ich samochód to Fiat, lubią kolor czerwony, kupili książkę "Lodołamacz", a pożyczyli książkę "Ekonomia w jednej lekcji".

Zosia i Rysiek Komar pracują jako nauczyciele, ich samochód to Toyota, lubią kolor niebieski, kupili książkę "Cywilizacja bizantyńska", a pożyczyli książkę "Rosja w roku 1839".

Pamela i Paweł Sawa pracują jako klienci-kontrolerzy, ich samochód to Renault, lubią kolor zielony, kupili książkę "Stulecie kłamców", a pożyczyli książkę "Lodołamacz".

Dinner[6]

Zeszłego weekendu, pięciu przyjaciół zebrało się na kolacji w ich ulubionej restauracji *Steak and seafood*. Każdy z przyjaciół (dwóch panów: Georg i Oliver, i trzy panie imieniem Colleen, Patti i Teresa) zamówiło różne dania główne (krab, befsztyk wołowy, żeberka, krewetki, lub polędwica wołowa), i różnego rodzaju ziemniaki (pieczone, frytki, lyońskie, puree lub zapiekane). Na popitkę każdy z przyjaciół wybrał jeden z następujących napojów: piwo imbirowe, mrożona herbata, lemoniada, piwo korzenne, woda. Napisz program wyznaczający imiona i nazwiska uczestników kolacji, ich dania główne, rodzaj ziemniaków i rodzaj napoju, jeżeli wiadomo, że:
1) Jedyna osoba o tych samych pierwszych literach imienia i nazwiska zamówiła żeberka.
2) Osoba o nazwisku Petroski i osoba, która zamówiła krewetki są osobami z których jedna miała ziemniaki po lyońsku, a druga piwo korzenne.
3) Osoba, która zamówiła befsztyk wołowy, nie zamówiła lemoniady.
4) Osoba, która miała zapiekane ziemniaki (ale bez polędwicy wołowej) nie piła wody.
5) Pierwsza litera imienia osoby, która miała piwo korzenne, jest taka sama jak pierwsza litera nazwiska Georga.
6) Teresa nie zamówiła wody.
7) Osoby, które zamówiły krewetki i pieczone ziemniaki, są odmiennej płci.
8) Pierwsza litera imienia pani, która zamówiła kraba, jest taka sama jak pierwsza litera nazwiska osoby, która wybrała ziemniaki puree.
9) Pierwsza litera imienia osoby, która zamówiła lemoniadę, jest taka sama jak pierwsza litera nazwiska osoby, która zamówiła ziemniaki lyońskie.
10) Osoba o nazwisku Chiasson (ale nie jest to Patti) nie zamówiła frytek ani ziemniaków lyońskich.
11) Osoba o nazwisku Truang (która nie zamówiła frytek ani ziemniaków puree) nie wybrała piwa imbirowego.
12) Colleen zamówiła albo befsztyk wołowy albo polędwicę wołową.

Rozwiązanie:

Colleen Petroski: befsztyk wołowy, ziemniaki puree, piwo korzenne.

George Chiasson: polędwica wołowa, ziemniaki pieczone, lemoniada

Oliver Orlando: żeberka, frytki, woda.

Patti Truang: krab, ziemniaki zapiekane, mrożona herbata.

Teresa Gold: krewetki, ziemniaki lyońskie, piwo imbirowe.

Wybór zupy

Każdy z sześciu przyjaciół z Szkoły Kucharskiej jest obecnie cenionym szefem w jakiejś renomowanej restauracji. Co kilka tygodni przyjaciele spotykają się w celu wymiany doświadczeń i delektowania się swymi kulinarnymi kreacjami. Na ostatnim spotkaniu każdy zjawił się z odmienną, specjalnie na tę okazję przygotowana zupą, którą serwowano przy sześciokątnym stole o miejscach ponumerowanych 1, 2,...,6, zgodnie z kierunkiem ruchu wskazówek zegara. Wiadomo, że:

1. Gloria (która pracuje albo w Apple Orchard Inn albo w Hennigan's Place) przygotowała albo francuską zupę cebulową albo grochówkę.
2. Osoba, która przygotowała zupę jarzynową zajmowała miejsce o niższym numerze aniżeli Marvin.
3. Osoba o nazwisku Anderson siedziała dokladnie naprzeciwko szefa pracującego w Michel's Cafe.
4. Marvin i osoba, która przygotowała zupę szparagową są osobami, z których jedna siedzi na miejscu 5, a druga siedzi na przeciwko szefa, który zrobił rosół z kury z makaronem.
5. Norville siedział obok osoby szefującej w Country Kitchen.
6. Quincy i szef pracujący w Village Smorgasbord są osobami, z których jedna ma nazwisko Dugan, a druga nie siedzi naprzeciwko szefa który zrobił zupę szparagową.
7. Szef pracujący w Pine Cove Restaurant i szef siedzący na miejscu 4 sa osobami, z których jedna ma na nazwisko Anderson, a druga nie przygotowała zupy jarzynowej.
8. Osoba o nazwisku Burns (pracująca w Apple Orchard Inn) nie przygotowała grochówki.
9. Sześcioma szefami są Jenna, szef o nazwisku Dugan, szef pracujący w Pine Cove Restaurant, osoba która przygotowała krem z małży, szef siedziący na miejscu 3 i szef siedzący naprzeciwko osoby o nazwisku Dugan.
10. Isabela i osoba o nazwisku Friedman to osoby, z których jedna pracuje w Michel's Cafe, a druga przygotowała rosół z kury z makaronem.
11. Marvin nie przygotował kremu z małży.
12. Jenna (która siedzi na miejscu o nieparzystym numerze) siedzi obok osoby o nazwisku Caruso.
13. Szef pracujący dla Pine Cove Restaurant nie zajmuje miejsca o numerze 6.

Napisz program wyznaczający imię i nazwisko każdego szefa, numer jego

miejsca przy stole, nazwę miejsca pracy oraz nazwę zupy. którą częstował przyjaciół.

Rozwiązanie:
Miejsce 1, Isabela Earle, Country Kitchen, rosół z kury z makaronem
Miejsce 2, Norville Anderson, Pine Cove Restaurant, zupa jarzynowa
Miejsce 3, Gloria Burns, Apple Orchard Inn, francuska zupa cebulowa
Miejsce 4, Marvin Dugan, Village Smorgasbord, grochówka
Miejsce 5, Jenna Friedman, Michel's Cafe, zupa szparagowa
Miejsce 6, Quincy Caruso, Hennigan's Place, krem z małży

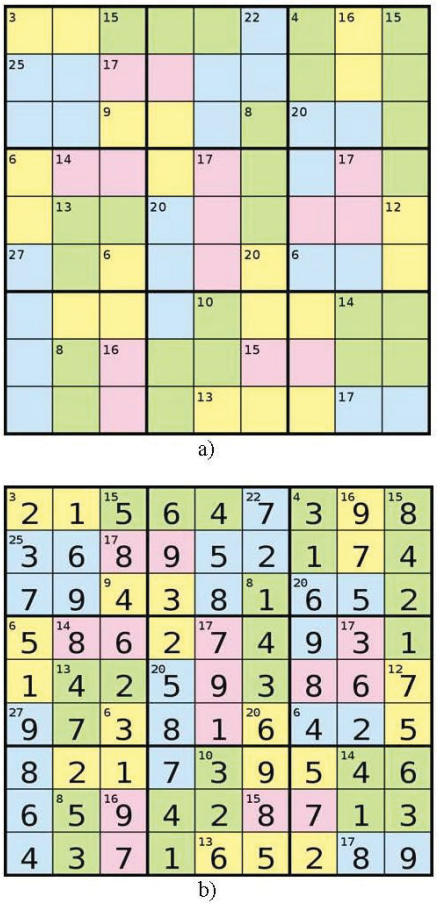
Historyczne domy

Każdego roku klub *Old Houses* sponsoruje *Wycieczkę po Historycznych Domach*, w której pięć starych domów jest (za zgoda właścicieli) otwartych dla publiczności. Tego roku pięć domów jest przy różnych ulicach (Azalea Drive, Crepe Myrtle Court, Jasmine Boulevard, Magnolia Street i Oleander Road). Każdy został wybudowany w innym roku (1860, 1870, 1890, 1900 i 1920). Napisz program wyznaczający porządek, w którym wycieczka zwiedzi tych pięć domów (identyfikując je po ulicach) i wyznaczający roczniki domów, jeżeli wiadomo, że:
1. Dom na Jasmine Boulevard jest o 20 lat starszy niż dom na Azalea Drive.
2. Trzeci dom na wycieczce został wybudowany w 1860.
3. Wycieczka zwiedziła dom na Magnolia Street trochę czasu przed domem wybudowanym w 1890.
4. Wycieczka zwiedziła dom na Oleander Road (który nie byl wybudowany jako ostatni z piątki) nieco przed domem zwiedzanym przy Jasmine Boulevard, który to dom byl zwiedzany nieco przed domem wybudowanym w 1900.

Zabójcze Sudoku [7]

Napisz program rozwiązujący *Zabójcze Sudoku* przedstawione na rysunku 4.8a). Dodatkowym ograniczeniem jest, by suma liczb z sąsiadujących pól o jednakowych kolorach była równa liczbie w górnym lewym rogu skrajnie lewej kratki pola. Rysunek 4.8b) przedstawia rozwiązanie tego problemu.

[7] Przykład z http://en.wikipedia.org/wiki/Killer sudoku



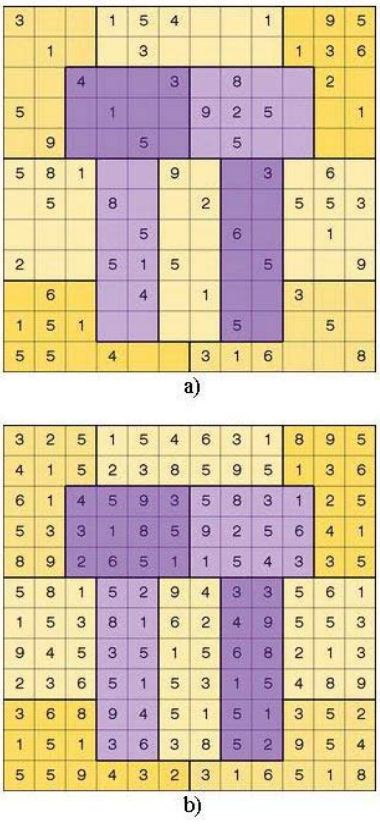
Rysunek 4.8: Temat a) i rozwiązanie b) dla Zabójczego Sudoku

Pi-Day Sudoku [8]

Napisz program rozwiązujący Pi-Day Sudoku z rysunku 4.9a). Każdy wiersz, każda kolumna i każdy zaznaczony nieregularny obszar musi zawierać pierwsze dwanaście liczb pi, z uwzględnieniem powtórzeń: 3.14159265358. A więc każdy wyróżniony obszar musi zawierać dwie jedynki, dwie trójki,

[8] Przykład z http://www.brainfreezepuzzles.com/main/piday2008.html

trzy piatki i nie może zawierać siódemki. Rozwiązanie problemu przedstawia rysunek 4.9b).



Rysunek 4.9: Temat a) i rozwiązanie b) dla Pi-Day Sudoku

Rozdział 5

CLP z predykatami elementarnymi dla rozwiązań optymalnych

5.1 Uwagi ogólne

Programy komputerowe rozwiązujące (dla celów wspomagania decyzji) skomplikowane zagadnienia kombinatoryczne, w tym również dokonujące optymalizacji kombinatorycznej, były tradycyjnie opracowywane w ramach badań operacyjnych pod nazwą *programowanie całkowitoliczbowe* . Wyróżnia się ono kodowaniem wszystkich zmiennych kombinatorycznych za pomocą zmiennych binarnych o wartościach `0` i `1`. Tego typu kodowanie można również stosować dla *problemów optymalnego spełniania ograniczeń* (POSO), aczkolwiek nie jest ono zalecane ze względu na duży wzrost liczby zmiennych potrzebnej do opisu problemu.

W ramach programowania całkowitoliczbowego wyróżnić można dwa podejścia metodologiczne:

1. Stosowanie istniejących solwerów (np. dla całkowitoliczbowego programowania liniowego). Zaletą tego podejścia jest korzystanie z mocnych, wypróbowanych algorytmów solwera, co skraca czas i obniża koszt rozwiązania. Wadą zaś jest to, że rozwiązanie wymaga transformacji rozwiązywanego problemu do *postaci kanonicznej* akceptowalnej przez solwer. Transforma-

cja ta jest źródłem *przepaści semantycznej* pomiędzy pierwotnym i rozwiązywanym problemem: transformacja czyni problem słabo czytelnym ze względu na potrzebę wprowadzenia dużej liczby zmiennych pomocniczych, często nie mających bezpośredniego odzwierciedlenia w pierwotnym problemie. Wadą jest również to, że wymogi solwera mogą bardzo utrudnić lub wręcz uniemożliwić wbudowanie w program dodatkowej wiedzy o problemie, mogącej znacznie ułatwić rozwiązanie problemu.

2. Opracowanie specyficznego ("szytego na miarę") algorytmu rozwiązującego problem w sposób optymalny i kodowanie tego algorytmu za pomocą języka proceduralnego właściwego dla obliczeń arytmetykochłonnych (np. FORTRAN, Pascal, C). Zalety tego podejścia to możliwość wbudowania w program wszelkiej wiedzy o pierwotnym problemie i możliwość uzyskania dużej efektywności programu. Wadami zaś bardzo długi czas opracowania i testowania algorytmu oraz programu, konieczność zatrudniania wysoko kwalifikowanych i opłacanych specjalistów i związana z tym ograniczona modyfikowalność programu, często niemożliwa bez zaangażowania twórców programu.

Ze względów natury dydaktycznej podejście programowania całkowitoliczbowego (zwane w dalszym ciągu podejściem badań operacyjnych zostanie zilustrowane szeregiem przykładów wyróżnionych skrótem `BO`).

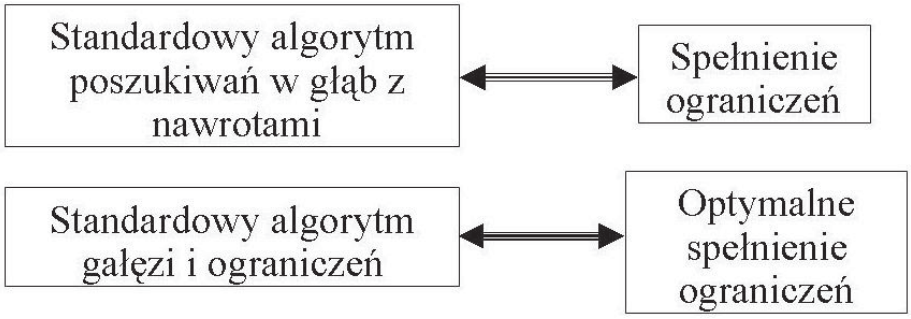
Społeczność *CLP* opracowała odmienne, deklaratywne podejście do optymalizacji kombinatorycznej, nie wymagające transformacji pierwotnych zmiennych całkowitoliczbowych problemu do postaci zmiennych binarnych, często trudnych do interpretowania. Podejście to, zwane w dalszym ciągu podejściem *CLP*, będzie w dalszym ciągu podejściem preferowanym, gdyż jest ono pozbawione większości wad programowania całkowitoliczbowego. Przykłady ilustrujące to podejście zostaną wyróżnione skrótem `CLP`.

5.2 Gałęzie i ograniczenia

Podstawową metodą optymalizacji, stosowaną w tym i w następnym rozdziale, będzie metoda *gałęzi i ograniczeń* (ang. *branch and bound*). Z metodą tą (w wersji standardowej) zetknęliśmy się już w rozdziale 2.3.1. Dobrze będzie jednak przyjrzeć się jej bliżej.

Zaczniemy od stwierdzenia, że tym, czym dla *problemów spełnienia ograniczeń* był standardowy algorytm poszukiwań w głąb z nawrotami, tym dla

problemów optymalnego spełnienia ograniczeń jest standardowy algorytm *gałęzi i ograniczeń*, patrz rysunek 5.1.



Rysunek 5.1: Analogia pomiędzy standardowymi algorytmami *poszukiwań w głąb z nawrotami* a *gałęzi i ograniczeń*

Zasadnicza różnica pomiędzy standardowymi poszukiwaniami w głąb a standardowymi *gałęziami i ograniczeniami* polega na tym, że dla tego ostatniego, w trakcie przeszukiwania drzewa poszukiwań, jest testowane dodatkowe ograniczenie pomiędzy aktualnie wyznaczoną wartością wskaźnika jakości (AWWWJ) a dotychczas najlepszą wartością wskaźnika jakości (DNWWJ). Dla przypadku minimalizacji, która jest standardem w większości platform *CLP*, można to sformułować następująco:

- jeżeli `AWWWJ < DNWWJ`, to `DNWWJ` zostaje przypisana wartość `AWWJ`, zbiór wartości zmiennych decyzyjnych dla `DNWWJ` zostaje usunięty i zastąpiony zbiorem wartości zmiennych decyzyjnych dla `AWWWJ`;
- jeżeli `AWWWJ > DNWWJ`, nic się nie zmienia;
- jeżeli `AWWWJ = DNWWJ`, spotykane są dwa rozwiązania: 1)nic się nie zmienia - ten sposób jest stosowany w ECL^iPS^e CLP i na innych platformach *CLP*; 2)zostaje przechowany zbiór wartości zmiennych decyzyjnych dla wyznaczonej wartości `DNWWJ`: tak właśnie postąpiono w przykładzie `2_9_konf_opt.pl`.

Z powyższego wynika, że *gałęzie i ograniczenia* nie są ściśle mówiąc konkretnym algorytmem optymalizacji, lecz bardzo ogólną metodą wszelakiej optymalizacji kombinatorycznej, dopuszczającą wskaźniki jakości będące nieliniowymi

funkcjami zmiennych decyzyjnych oraz ograniczenia wyrażane również za pomocą nieliniowych funkcji zmiennych decyzyjnych. Praktycznie jednak - ze względów natury numerycznej - obecnie spotykane platformy *CLP* (w tym również ECL^iPS^e) umożliwiają stosowanie metody *gałęzi i ograniczeń* wyłącznie dla liniowych wskaźników jakości.

5.3 Udoskonalone *gałęzie i ograniczenia*

Z podobieństw istniejących pomiędzy metodą *gałęzi i ograniczeń* a metodą *poszukiwań w głąb ze standardowymi nawrotami* wynika, że tą pierwszą można udoskonalić wprowadzając mechanizmy omówione w rozdziałach 3.2.4 i 3.2.5, tzn. *sprawdzanie krok w przód* i *sprawdzanie dwóch kroków w przód*. Rozważania na temat heurystyk poszukiwań z rozdziału 3.3 są również ważne dla metody *gałęzi i ograniczeń*.

5.3.1 Optymalne ustawienie hetmanów - standardowe *gałęzie i ograniczenia*

Dla lepszego zrozumienia sposobów doskonalenia metody *gałęzi i ograniczeń* rozpatrzmy jej wersję standardową dla bardzo prostego problemu optymalizacji bezpiecznego ustawienia hetmanów na szachownicy o wymiarach 4×4. Należy ustawić 4 hetmanów na szachownicy w taki sposób, by uzyskać minimum wskaźnika jakości:

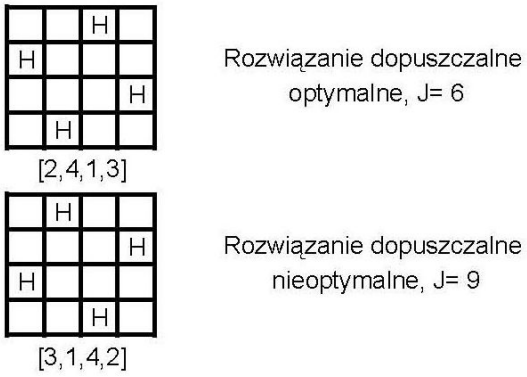
```
J = 1*X1+0*X2+1*X3+1*X4,
```

gdzie - jak poprzednio - `Xi` jest numerem wiersza, w którym hetman jest w `i`-tej kolumnie. Rysunek 5.2 przedstawia dwa dopuszczalne rozwiązania tego problemu, z których jedno jest optymalne.

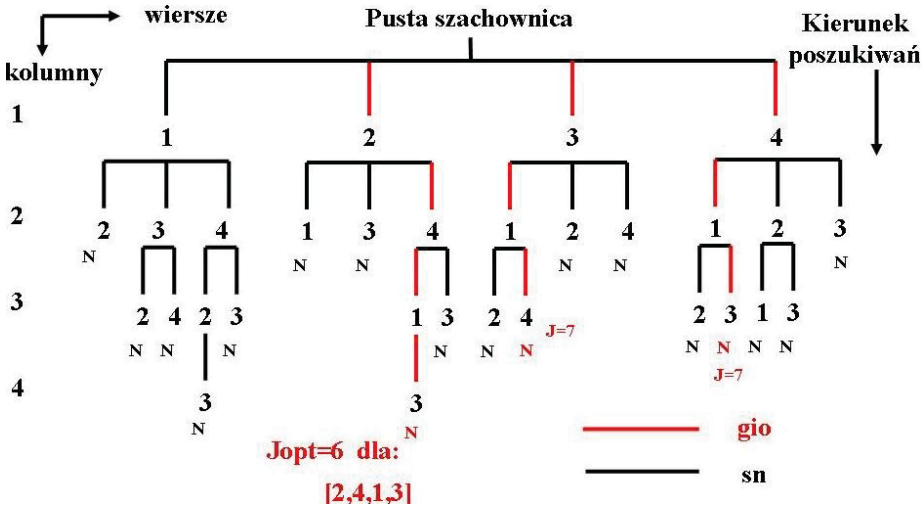
Gałęzie i ograniczenia można scharakteryzować wymieniając sytuacje, w których jest inicjowany nawrót. Dla standardowej wersji nawrót jest inicjowany:

- w wyniku uzyskania gorszej wartości wskaźnika jakości (*gałęzie i ograniczenia - gio*);
- w wyniku naruszenia ograniczenia (*standardowy nawrót - sn*).

Ilustruje to drzewo poszukiwań optymalnej konfiguracji czterech hetmanów z rysunku 5.3.



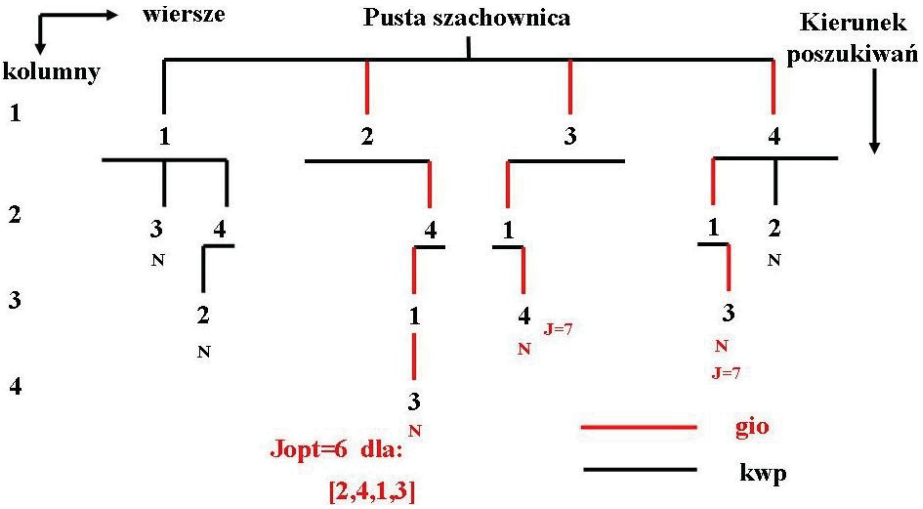
Rysunek 5.2: Dwa rozwiązania dopuszczalne dla 4 hetmanów



Rysunek 5.3: Drzewo poszukiwań metodą standardową *gałęzi i ograniczeń* dla 4 hetmanów

5.3.2 Optymalne ustawienie hetmanów - *gałęzie i ograniczenia* oraz *sprawdzanie krok w przód*

Dla *gałęzi i ograniczeń oraz sprawdzania krok w przód* nawrót jest inicjowany w następujących sytuacjach:



Rysunek 5.4: Drzewo poszukiwań metodą *gałęzi i ograniczeń oraz sprawdzania krok w przód* dla 4 hetmanów

- w wyniku uzyskania gorszej wartości wskaźnika jakości (*gałęzie i ograniczenia - gio*);
- w wyniku uzyskania pustej dziedziny (*sprawdzanie krok w przód - kw*p).

Ilustruje to drzewo poszukiwań optymalnej konfiguracji czterech hetmanów z rysunku 5.4.

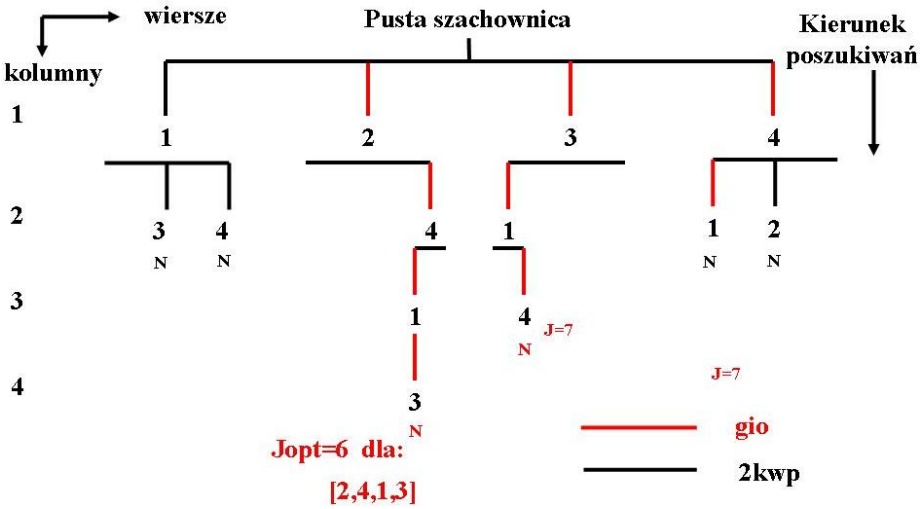
5.3.3 Optymalne ustawienie hetmanów - *gałęzie i ograniczenia* oraz *sprawdzanie dwóch kroków w przód*

Dla *gałęzi i ograniczeń oraz sprawdzania dwóch kroków w przód* nawrót jest inicjowany w następujących sytuacjach:

- w wyniku uzyskania gorszej wartości wskaźnika jakości (*gałęzie i ograniczenia - gio*);
- w wyniku braku wartości dopuszczalnych w dziedzinach niepustych (*sprawdzanie dwóch kroków w przód - 2kwp*);

- w wyniku uzyskania pustej dziedziny (*sprawdzanie krok w przód*).

Ilustruje to drzewo poszukiwań optymalnej konfiguracji czterech hetmanów z rysunku 5.5.



Rysunek 5.5: Drzewo poszukiwań metodą *gałęzi i ograniczeń oraz sprawdzania dwóch kroków w przód* dla 4 hetmanów

W ostatnim przypadku drzewo poszukiwań ma tylko `6` liści w porównaniu z `18` liśćmi drzewa poszukiwań dla standardowej metody gałęzi i ograniczeń. W przypadkach bardziej złożonych drzew poszukiwań korzyści ze stosowania różnorodnych modyfikacji standardowej metody gałęzi i ograniczeń mogą być znacznie bardziej poważne.

5.4 Podstawowe predykaty

W rozdziale tym (oraz w następnym) będą stosowane dwa bardzo ważne predykaty standardowe

1. `bb_min/3` dla realizacji metody *gałęzi i ograniczeń* z biblioteki `lib(branch_and_bound)`

2. `search/6` dla przeszukiwania przestrzeni stanu w ramach metody *gałęzi i ograniczeń* lub w ramach poszukiwań rozwiązań dopuszczalnych, z biblioteki `lib(ic)`.

Obszerny opis obydwu predykatów jest przedstawiony w dokumentacji ECL^iPS^e, patrz rysunek 5. Ze względu na ich znaczenie dla rozważanych przykładów - robiąc odstępstwo od zasady sformułowanej w rozdziale 0.3 - przedstawimy niektóre właściwości tych predykatów.

5.4.1 Predykat 'bb min/3'

Predykat `bb_min/3` inicjuje poszukiwania metodą *gałęzi i ograniczeń*. W najprostszej wersji ma postać:

```
bb_min(+szukaj, ?Wskaźnik_jakości, ?Opcje)
```

gdzie:

- `szukaj` jest predykatem zależnym od zmiennych decyzyjnych problemu, dokonującym takiego ich uziemiania, które zmierza do minimalizacji `Wskaźnika_jakości`. Predykatem tym może być `labeling/1` lub `search/6`;

- `Wskaźnik_jakości` jest zmienną minimalizowaną na drodze uziemiania zmiennych decyzyjnych;

- `Opcje` (ważniejsze) mogą być następujące:

  - `strategy` z następującymi pod-opcjami:

    * `strategy=continue` oznacza, że po wyznaczeniu rozwiązania należy kontynuować poszukiwania ze znalezionym rozwiązaniem. Jest to pod-opcja domyślna;

    - `strategy=restart` albo `strategy=step` oznacza, że po wyznaczeniu rozwiązania należy rozpocząć poszukiwania od początku przyjmując wyznaczoną wartość wskaźnika jakości jako nowe ograniczenie górne tego wskaźnika;
    - `strategy=dichotomic` oznacza, że po znalezieniu rozwiązania dzieli się pozostały (niezbadany) zakres zmian wskaźnika jakości na dwa zakresy i poszukuje się rozwiązania w dolnym zakresie. Jeżeli się to nie uda, dzieli się pozostały górny zakres na dwa podzakresy, itd. Nowe ograniczenie wskaźnika jakości albo punkt podziału są wyznaczane przy uwzględnieniu parametrów `delta` i `factor`, patrz niżej.

- `from: ograniczenie_dolne_wskaźnika_jakości`. Domyślnie jest `from:-1.0Inf`;
- `to: ograniczenie_górne_wskaźnika_jakości`. Domyślnie jest `to:+1.0Inf`;
- `delta` jest minimalną oczekiwaną bezwzględną poprawą wskaźnika jakości z kroku na krok. Domyślnie jest `delta:1`.
- `factor` jest minimalną oczekiwaną względną poprawą wskaźnika jakości z kroku na krok, odniesioną do dolnego ograniczenia tego wskaźnika. Domyślnie jest brak ograniczenia.
- `timeout` jest maksymalną liczbą sekund pracy jednostki centralnej komputera przy wyznaczaniu rozwiązania. Domyślnie jest brak ograniczenia.

Dla predykatu `bb_min/3` istnieje wersja uproszczona `minimize/2` o postaci:

```
minimize(+szukaj, ?Wskaźnik_jakości),
```

stosowana z powodzeniem dla prostych problemów optymalizacji w przypadku opcji domyślnej.

Twórcy ECL^iPS^e nie dali użytkownikowi (chyba słusznie) możliwości wyboru jednej z opcji poszukiwań opisanych w rozdziale 5.3. Wypada założyć, że zastosowano najefektywniejszą z posiadanych.

5.4.2 Predykat 'search/6'

Predykat ten jest uogólnieniem predykatu `labeling/1`, dokonującego uziemienia zmiennych (tzn. dokonującego poszukiwań), przedstawionego w rozdziale 3.2.1. W wersji współpracującej z biblioteką *ic* ma następującą postać:

```
search(+Lista,++Arg,++Wybieranie_zmiennych,+Wybieranie_wartości,
           ++Metoda,+Opcje)
```

gdzie:

- `Lista` jest listą zmiennych decyzyjnych (jeżeli `Arg = 0`) lub termów (jeżeli `Arg > 0`);
- `Arg` jest liczbą całkowitą, równą `0` jeżeli `Lista` jest listą zmiennych, lub większą od `0`, jeżeli `Lista` jest listą termów o arności większej niż `Arg`; w tym ostatnim przypadku `Arg` wskazuje wybrany argument termu;
- `Wybieranie_zmiennych` (zob. również 3.3) oznacza jedną z poniżej zdefiniowanych *heurystyk wyboru zmiennych* do uziemiania:
  - `input_order` - wybrany zostaje pierwszy element listy;
  - `first_fail` - wybrany zostaje element o najmniejszej dziedzinie;
  - `anti_first_fail` - wybrany zostaje element o największej dziedzinie;
  - `smallest` - wybrany zostaje element o najmniejszej wartości;
  - `largest` - wybrany zostaje element o największej wartości;
  - `occurrence` - wybrany zostaje element występujący w największej liczbie ograniczeń;
  - `most_constrained` - wybrany zostaje element o najmniejszej dziedzinie. Gdy takich elementów jest więcej, wybrany zostaje element występujący w największej liczbie ograniczeń;
  - `max_regret` - wybrany zostaje element o maksymalnej różnicy pomiędzy najmniejszą i drugą z kolei najmniejszą wartością w dziedzinie;
- `Wybieranie_wartości` (zob. również 3.3) oznacza jedną z poniżej zdefiniowanych *heurystyk wyboru wartości* dla zmiennych wyznaczonych za pomocą opcji `Wybieranie_zmiennych`:

  - `indomain` - wybrany zostaje porządek według wzrastających wartości, patrz rozdział 3.2. W przypadku porażki testowana wartość nie jest usuwana z dziedziny;
  - `indomain_min` - wybrany zostaje porządek leksykograficzny. W przypadku porażki testowana wartość jest usuwana z dziedziny;
  - `indomain_max` - wybrany zostaje porządek według malejących wartości. W przypadku porażki testowana wartość jest usuwana z dziedziny;
  - `indomain_middle` - wybrany zostaje porządek od środka dziedziny;
  - `indomain_median` - wybrany zostaje porządek od wartości mediany dziedziny;
  - `indomain_split` - dziedzina jest połowiona, poddziedzina w której wystąpi porażka zostaje usunięta. Pozostała jest ponownie połowiona;
  - `indomain_random` - wybrany zostaje porządek losowy;
  - `indomain_interval` - jeżeli dziedzina składa się z kilku przedziałów, dokonuje się najpierw wyboru przedziału. W przypadku pojedynczego przedziału stosuje się `indomain_split`;

- `Metoda` oznacza jedną z dziesięciu metod poszukiwań, z których podstawowymi są:
  - `complete` - wszystkie możliwe uziemienia zmiennych są testowane;
  - `bbs(Liczba_nawrotów)` - poszukiwania z ograniczoną liczbą nawrotów;
  - `sbds` - wszystkie możliwe uziemienia są testowane z pominięciem symetrycznych części drzewa poszukiwań;

- `Opcje` oznaczają jedną z czterech możliwych opcji. Najbardziej podstawowymi są następujące:
  - `backtrack(-N)` zwraca liczbę nawrotów wykonanych w trakcie poszukiwań;
  - `nodes(++N)` ustala górną granicę liczby węzłów drzewa poszukiwań, odwiedzanych w trakcie poszukiwań;

5.5 Prosty przykład optymalizacji

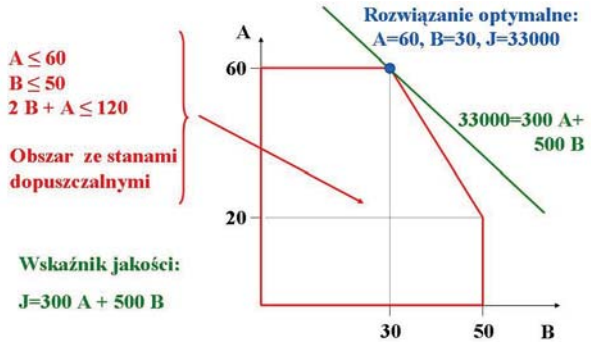
Podstawowe elementy POSO ilustruje następujący prosty przykład montowni komputerów, która ma na magazynie 60 płyt głównych typu A, 50 płyt głównych typu B oraz 120 dysków twardych. Komputery z płytami głównymi A dają zysk 300 JP i wymagają po jednym dysku twardym, komputery z płytami głównymi B dają zysk 500 JP i wymagają dwóch dysków twardych. Ile wyprodukować komputerów różnych typów, by zmaksymalizować zysk? Czytelnicy znający programowanie całkowitoliczbowe łatwo rozpoznają problem jako należący do tej klasy. Jego graficzne rozwiązanie przedstawia rysunek 5.6. Odpowiedni program 5_1_PL.ecl ma postać:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).
 /*3*/   top :-
/*4*/       Plyty=[A,B],
/*5*/       Plyty :: 1..60,
/*6*/       Zysk :: 30000..40000,
/*7*/       A #=< 60,
/*8*/       B #=< 50,
/*9*/       A + 2*B #=< 120,
/*10*/      Zysk #= 300*A + 500*B,
/*11*/      Ujemny_zysk #= - Zysk,
/*12*/      minimize(labeling([A,B]),Ujemny_zysk),
/*13*/      writeln("Maksymalny zysk":Zysk),nl,
/*14*/      write("A_opt = "), write(A), nl,
/*15*/      write("B_opt = "), write(B),nl.
```

Ponieważ problem wymaga maksymalizacji, a dysponujemy predykatem dla minimalizacji, wykonuje się ją dla ujemnego zysku. Komunikat pokazujący kolejne, coraz to lepsze rozwiązania z drzewa poszukiwań, jest następujący:

```
Found a solution with cost -30100 Found a solution with cost -30400
Found a solution with cost -30700 Found a solution with cost -31000
Found a solution with cost -31100 Found a solution with cost -31200
Found a solution with cost -31300 Found a solution with cost -31400
Found a solution with cost -31500 Found a solution with cost -31600
Found a solution with cost -31700 Found a solution with cost -31800
Found a solution with cost -31900 Found a solution with cost -32000
```

```
Found a solution with cost -32100 Found a solution with cost -32200
Found a solution with cost -32300 Found a solution with cost -32400
Found a solution with cost -32500 Found a solution with cost -32600
Found a solution with cost -32700 Found a solution with cost -32800
Found a solution with cost -32900 Found a solution with cost -33000
Found no solution with cost -40000.0 .. -33001.0
Maksymalny zysk :  33000
A_opt = 60 B_opt = 30
```



Rysunek 5.6: Graficzne rozwiązanie prostego przykładu optymalizacji

Problemy optymalizacji omawiane poniżej zostaną przedstawione zgodnie z klasyfikacją z rozdziału 1.7.

5.6 Optymalne konfigurowanie

5.6.1 System 3-elementowy - podejście BO

Spróbujemy obecnie najpierw rozwiązać zadanie konfigurowania optymalnego z rozdziału 2.3.1 za pomocą podejścia *BO*, możliwego do stosowania również na platformie ECL^iPS^e. Odpowiedni program `5_2_konfiguracja_BO.ecl` zawiera zmienne `A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2`, równe `1`, jeżeli odpowiedni element jest częścią konfiguracji, a równe `0` w przypadku przeciwnym. Program ma postać:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).

/*3*/   top :-
/*4*/       Konf=[A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2],
/*5*/       Konf :: 0..1,
   %  Sens zmiennych: jeżeli np. A_1 = 0, to konfiguracja nie ma elementu A_1
/*6*/       Koszt:: 1..3000,

 % Linie /*7*/ - /*9*/  deklarują wymogi konfiguracyjne systemu:
/*7*/       A_1 + A_2 + A_3 #= 1,       % konfiguracja ma jeden element typu A
/*8*/       B_1 + B_2 + B_3 + B_4 #= 1, % itp.
/*9*/       C_1 + C_2 #= 1,             % itp.
 % Linie /*10*/ - /*15*/  deklarują ograniczenia kompatybilności elementów systemu:
/*10*/      C_1 + A_2 #=< 1,   % C_1 i A_2 są niekompatybilne
/*11*/      B_2 + C_2 #=< 1,   % itp.
/*12*/      C_2 + B_3 #=< 1,   % itp.
/*13*/      B_4 + A_2 #=< 1,   % itp.
/*14*/      B_3 + A_1 #=< 1,   % itp.
/*15*/      A_3 + B_3 #=< 1,   % itp.

/*16*/      Koszt #= A_1 * 1900 + A_2 * 750 + A_3 * 900 +
                   B_1 * 300 + B_2 * 500 + B_3 * 450 +B_4 * 600 +
                           C_1 * 700 + C_2 * 850,

/*17*/      bb_min(labeling(Konf),Koszt,bb_options with [strategy:step]),

/*18*/      write("Minimalny koszt konfiguracji: "),write(Koszt),nl,nl,
/*19*/      write("Optymalna konfiguracja:"),nl,
/*20*/      write([A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2]),nl,
/*21*/      write(["A_1","A_2","A_3","B_1","B_2","B_3","B_4","C_1","C_2"]),nl,
/*22*/      pisz_konfiguracja([A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2],
              ["A_1","A_2","A_3","B_1","B_2","B_3","B_4","C_1","C_2"]),nl,nl,
/*23*/      fail.

/*24*/  top :-
/*25*/      write("To wszystkie rozwiazania optymalne.").

/*26*/  pisz_konfiguracja([H1|T1],[H2|T2]):-
/*27*/      H1 is 1, write(H2),write("  "),
/*28*/      pisz_konfiguracja(T1,T2).

/*29*/  pisz_konfiguracja([H1|T1],[_|T2]):-
/*30*/      H1 is 0,
/*31*/      pisz_konfiguracja(T1,T2).
/*32*/  pisz_konfiguracja([],[]).
```

Program ten daje następujące rozwiązanie:

```
Found a solution with cost 2350
Found a solution with cost 2200
Found a solution with cost 2100
Found a solution with cost 2050
Found a solution with cost 1900
Found no solution with cost 1.0 .. 1899.0

Minimalny koszt konfiguracji:1900
Optymalna konfiguracja:
[0, 0, 1, 1, 0, 0, 0, 1, 0]
[A_1, A_2, A_3, B_1, B_2, B_3, B_4, C_1, C_2]
A_3  B_1  C_1
To wszystkie rozwiazania optymalne.
```

Jak widać, `fail` w linii `/*23*/` nie umożliwił wyznaczenia drugiego rozwiązania optymalnego, o którego istnieniu wiemy (patrz `2_9_konf_opt.pl`). Jest to dosyć istotne ograniczenie platformy ECL^iPS^e. Aby wyznaczyć wszystkie rozwiązania optymalne, należy skorzystać ze znajomości minimalnego kosztu wyznaczonej konfiguracji dla zawężenia dziedziny kosztów (patrz wiersz `/*5*/` poniższego programu), a następnie wyznaczyć wszystkie rozwiązania o tym koszcie.

Ilustruje to program `5_3_konfiguracje_wszystkie_BO.ecl`:

```
/*1*/   :- lib(ic).

/*2*/   top :-
/*3*/       Konf=[A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2],
/*4*/       Konf :: 0..1,
/*5*/       Koszt is 1900,

/*6*/       A_1 + A_2 + A_3 #= 1,
/*7*/       B_1 + B_2 + B_3 + B_4 #= 1,
/*8*/       C_1 + C_2 #= 1,
/*9*/       C_1 + A_2 #=< 1,
/*10*/      B_2 + C_2 #=< 1,
/*11*/      C_2 + B_3 #=< 1,
/*12*/      B_4 + A_2 #=< 1,
/*13*/      B_3 + A_1 #=< 1,
/*14*/      A_3 + B_3 #=< 1,

/*15*/      Koszt #=  A_1 * 1900 + A_2 * 750 + A_3 * 900 +
                B_1 * 300 + B_2 * 500 + B_3 * 450 + B_4 * 600 +
```

```
                        C_1 * 700 + C_2 * 850,
/*16*/      labeling(Konf),

/*17*/      writeln("Minimalny koszt konfiguracji": Koszt),nl,nl,
/*18*/      write("Wynik poszukiwań:"),nl,
/*19*/      write([A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2]),nl,
/*20*/      write(["A_1","A_2","A_3","B_1","B_2","B_3","B_4", "C_1","C_2"]),nl,
/*21*/      write("Optymalna konfiguracja:"),nl,
/*22*/      pisz_konfiguracja([A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2],
            ["A_1","A_2","A_3","B_1","B_2","B_3","B_4","C_1","C_2"]),nl,nl,
/*23*/      fail.

/*24*/   top :-
/*25*/      write("To wszystkie rozwiazania optymalne.").

/*26*/   pisz_konfiguracja([H1|T1],[H2|T2]):-
/*27*/      H1 is 1, write(H2),write("  "),
/*28*/      pisz_konfiguracja(T1,T2).

/*29*/   pisz_konfiguracja([H1|T1],[_|T2]):-
/*30*/      H1 is 0,
/*31*/      pisz_konfiguracja(T1,T2).
/*32*/   pisz_konfiguracja([],[]).
```

Program ten generuje następujący komunikat:

```
Minimalny koszt konfiguracji:1900
Wynik poszukiwan:
[0, 0, 1, 1, 0, 0, 0, 1, 0]
[A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2]
Optymalna konfiguracja:
A_3  B_1  C_1

Minimalny koszt konfiguracji:1900
Wynik poszukiwan:
[0, 1, 0, 1, 0, 0, 0, 0, 1]
[A_1,A_2,A_3,B_1,B_2,B_3,B_4,C_1,C_2]
Optymalna konfiguracja:
A_2  B_1  C_2

To wszystkie rozwiazania optymalne.
```

5.6.2 System 3-elementowy - podejście CLP

Obecnie problem optymalnej konfiguracji zostanie rozwiązany przy zastosowaniu podejścia *CLP* za pomocą programu `5_4_konfiguracja_CLP.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).

/*4*/   top:-
   %  CA - koszt elementu A
   %  NA - indeks elementu A; jeżeli np. NA = 2, elementem konfiguracji będzie A_2
/*4*/        NA :: 1..3,
/*5*/        NB :: 1..4,
/*6*/        NC :: 1..2,
/*7*/        [CA,CB,CC] :: 300..1900,
/*8*/        Koszt :: 1800..2600,
/*9*/        element(NA,[1900,750,900],CA), % dla NA = 2 jest CA = 750
/*10*/       element(NB,[300,500,450,600],CB),
/*11*/       element(NC,[700,850],CC),
/*12*/       ~ niekompatybilne_NB_NC(NB,NC),
/*13*/       ~ niekompatybilne_NA_NB(NA,NB),
/*14*/       ~ niekompatybilne_NA_NC(NA,NC),
%     Tilda z  dowolonym predykatem (np. ~Cel) jest efektywnym operatorem negacji,
%     czekającym aż Cel zostanie uziemiony.

/*15*/       Koszt #= CA + CB + CC,
/*16*/       bb_min(labeling([NA,NB,NC]),Koszt,bb_options with [strategy:step]),

/*17*/       writeln("Optymalna konfiguracja:"),
/*18*/       write("("),write("A"),write(NA),write(","),
/*19*/       write("B"),write(NB),write(","),write("C"),write(NC),writeln(")"),
/*20*/       write("w cenie "),write(Koszt), writeln("."),nl,fail.

/*21*/  top:-
/*22*/       writeln("To wszystkie optymalne konfiguracje!").

/*23*/  niekompatybilne_NA_NB(2,4).
/*24*/  niekompatybilne_NA_NB(1,3).
/*25*/  niekompatybilne_NA_NB(3,3).

/*26*/  niekompatybilne_NA_NC(2,1).

/*27*/  niekompatybilne_NB_NC(2,2).
/*28*/  niekompatybilne_NB_NC(3,2).
```

Program generuje komunikat:

```
Found a solution with cost 1900
Found no solution with cost 1800.0 .. 1899.0
Optymalna konfiguracja:
(A2,B1,C2)
w cenie 1900.

To wszystkie optymalne konfiguracje!
```

Aby uzyskać wszystkie optymalne rozwiązania, stosowany jest ten sam trik jak dla przykładu `5_3_konfiguracje_wszystkie_BO.ecl`. Zrobiono to w programie `5_5_konfiguracje_wszystkie_CLP.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).

/*3*/   top:-
      %  CA - koszt elementu A
      %  NA - indeks elementu A
/*4*/        NA :: 1..3,
/*5*/        NB :: 1..4,
/*6*/        NC :: 1..2,
/*7*/        [CA,CB,CC] :: 300..1900,
/*8*/        Koszt is 1900,
/*9*/        element(NA,[1900,750,900],CA),
/*10*/       element(NB,[300,500,450,600],CB),
/*11*/       element(NC,[700,850],CC),
/*12*/       ~niekompatybilne_NB_NC(NB,NC),
/*13*/       ~niekompatybilne_NA_NB(NA,NB),
/*14*/       ~niekompatybilne_NA_NC(NA,NC),
/*15*/       Koszt #= CA + CB + CC,
/*16*/       labeling([NA,NB,NC]),
/*17*/       writeln("Optymalna konfiguracja:"),
/*18*/       write("("),write("A"),write(NA),write(","),
/*19*/       write("B"),write(NB),write(","),write("C"),write(NC),writeln(")"),
/*20*/       write("w cenie "),write(Koszt), writeln("."),nl,fail.

/*21*/  top:-
/*22*/       writeln("To już wszystkie optymalne konfiguracje!").

/*23*/  niekompatybilne_NA_NB(2,4).
/*24*/  niekompatybilne_NA_NB(1,3).
/*25*/  niekompatybilne_NA_NB(3,3).
/*26*/  niekompatybilne_NA_NC(2,1).
/*27*/  niekompatybilne_NB_NC(2,2).
/*28*/  niekompatybilne_NB_NC(3,2).
```

Program generuje komunikat::

```
Optymalna konfiguracja:  (A2,B1,C2) w cenie 1900.
Optymalna konfiguracja:  (A3,B1,C1) w cenie 1900.

To już wszystkie optymalne konfiguracje!
```

5.6.3 Problem plecakowy 1

Problem plecakowy jest klasycznym problemem konfigurowania rozwiązywanym w ramach optymalizacji kombinatorycznej. Jego nazwa pochodzi od takiego problemu wyboru przedmiotów z pewnego zbioru, by ich sumaryczna wartość była maksymalna i jednocześnie mieściły się w plecaku przemytnika. Najprostszą wersją problemu plecakowego jest jednowymiarowy problem plecakowy, który można rozwiązać korzystając z predykatu `iloczyn_skalarny/3`, jak to pokazano w przykładzie `5_6_plecak_1.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(branch_and_bound).

/*3*/  top:-
/*4*/       plecak([52,23,35,15,7],[100,60,70,15,15],60,[_,_,_,_,_]).

/*5*/  plecak(Rozmiary,Wartosci,Wielkosc_plecaka,[X1,X2,X3,X4,X5]):-
/*6*/       X = [X1,X2,X3,X4,X5],
/*7*/       X :: 0..1,
/*8*/       iloczyn_skalarny(Rozmiary,X,Rozmiar),
/*9*/       Rozmiar #=< Wielkosc_plecaka,
/*10*/      iloczyn_skalarny(Wartosci,X,Wartosc),
/*11*/      Koszt #= -Wartosc,
/*12*/      minimize(labeling(X),Koszt),nl,
/*13*/      Wartosc is -Koszt,
/*14*/      write("Wartosc = "),writeln(Wartosc),
/*15*/      write("Plecak = "), writeln(X),
/*16*/      write("Rozmiar ="),writeln(Rozmiar).

/*17*/ iloczyn_skalarny(Lista_1,Lista_2,Iloczyn_skalarny):-
/*18*/      (
/*19*/      foreach(V1, Lista_1),
/*20*/      foreach(V2, Lista_2),
/*21*/      foreach(Iloczyn,Iloczyn_lista)
```

```
/*22*/      do
/*232*/      Iloczyn = V1 * V2
/*24*/      ),
/*25*/      Iloczyn_skalarny #= sum(Iloczyn_lista).
```

Program generuje komunikat:

```
Found a solution with cost 0
Found a solution with cost -15
Found asolution with cost -30
Found a solution with cost -70
Found a solution with cost -85
Found a solution with cost -100
Found a solution with cost -130
Found no solution with cost -260.0 .. -131.0
Wartosc = 130
Plecak = [0, 1, 1, 0, 0]
Rozmiar = 58,
```

z którego wynika, że optymalny załadunek plecaka to przedmioty z pozycji drugiej i trzeciej, o rozmiarach odpowiednio 23 i 35, o łącznych rozmiarach 58 i wartości 130.

5.6.4 Urzeczowione ograniczenia

Często trzeba, by spełnienie ograniczenia powodowało, że wyróżniona zmienna dwuwartościowa (`Wskaźnik`) przyjmuje wartość `1`, w przeciwnym przypadku zaś wartość `0`. `Wskaźnik` ów może być użyteczny dla kolejnych ograniczeń. Można to uzyskać stosując `urzeczowione ograniczenia` (ang. *reified constraints*). Ilustrują to następujące przykłady:

```
% to jest polecenie 1:
[eclipse 1]: Liczba = 0, #>(Liczba,0,Wskaznik).
% to jest odpowiedź:
Liczba = 0 Wskaznik = 0,
```

co oznacza, że nieprawdą jest `Liczba > 0`.

```
% to jest polecenie 2:
[eclipse 2]: Liczba = 6, #>(Liczba,0,Wskaznik).
% to jest odpowiedź:
Liczba = 6
Wskaznik = 1,
```

co oznacza, że prawdą jest `Liczba > 0`.

Wszystkie predykaty elementarne można przedstawić w postaci urzeczowionej. Np. rozpatrzmy ograniczenie implikacji:

```
+ograniczenie(X) => +ograniczenie(Y),
```

dla którego spełnione '`ograniczenie(X)`' implikuje spełnienie '`ograniczenie(Y)`', które przykładowo funkcjonuje następująco:

```
% to jest polecenie:
[eclipse 3]: X is 9, Y is 8, X#<10 => Y+2#<12.
% to jest odpowiedź:
X = 9
Y = 8
Yes
```

Powyżej założono, że `X` i `Y` zostały uziemione przez poprzednią część programu. Ograniczenie to można przedstawić w postaci urzeczowionej jako:

```
% to jest polecenie:
[eclipse 4]: X is 9, Y is 8, =>(X#<10,Y+2#<12,Wskaznik).
% to jest odpowiedź:
X = 9
Y = 8
Wskaznik = 1
```

W przykładzie poniżej implikacja nie jest prawdą:

```
% to jest polecenie 1:
[eclipse 5]: X is 9, Y is 8, =>(X#<10,Y+2#>15,Wskaznik).
% to jest odpowiedź:
X = 9
```

```
Y = 8
Wskaznik = 0
```

Implikacja może być prawdą dla każdej wartości z dziedziny, np. w przykładzie:

```
% to jest polecenie 2:
[eclipse 6]: X is 12, Y::14..18,=>(X#=5,Y+2#>15,Wskaznik).
% to jest odpowiedź:
X = 12
Y = Y{14 .. 18}
Wskaznik = 1,
```

gdzie `Wskaźnik` jest określony jednoznacznie.

Natomiast gdy implikacja jest prawdziwa tylko dla pewnych wartości z dziedziny, np. dla przykładu:

```
% to jest polecenie 3:
[eclipse 7]: X is 12, Y::12..17,=>(X#=12,Y+2#>15,Wskaznik).
% to jest odpowiedź:
X = 12
Y = Y{12 .. 17}
Wskaznik = Wskaznik{[0, 1]},
```

to `Wskaznik` pozostaje nieokreślony.

5.6.5 Ograniczenia dla zbiorów

ECL^iPS^e CPS wyróżnia się możliwością stosowania ograniczeń dla dziedziny zbiorów liczb całkowitych. Wymaga to zastosowania biblioteki `ic_sets`.
Zbiorami liczb całkowitych są uporządkowane i pozbawione powtórzeń listy liczb całkowitych, np.:

```
Zbior_Czterech_Liczb = [41,42,43,44],   Zbior_Pusty =[]
```

Podkreśla się, że listy zawierające liczby niecałkowite, listy nieuporządkowane lub listy z powtórzeniami, nie mogą być przetwarzane przez predykaty biblioteki `ic_sets`.

Definiuje się również pojęcie *zmiennych zbiorowych* jako zmiennych przyjmujących wartości będące zbiorami liczb całkowitych. Zmienne takie można deklarować np. następująco:

```
Zmienna_Zbiorowa :: []..[1,2,3,4,5,6]
```

gdzie zbiór pusty `[]` jest *dolnym ograniczeniem* `Zmiennej_Zbiorowej`, a zbiór `[1,2,3,4,5,6]` jej *górnym ograniczeniem*. Przetestujmy niektóre jej właściwości:

```
% to jest polecenie:
[eclipse 1]:
        :-lib(ic_sets).
        Zmienna_Zbiorowa :: []..[1,2,3,4,5,6],
        Zmienna_Zbiorowa = [3,2,1].
% to jest odpowiedź:
[eclipse 2]:
No (0.00s cpu)

% to jest polecenie:
[eclipse 3]:
        :-lib(ic_sets).
        Zmienna_Zbiorowa :: []..[1,2,3,4,5,6],
        Zmienna_Zbiorowa = [1,4,6].
% to jest odpowiedź:
[eclipse 4]:
Zmienna_Zbiorowa = [1, 4, 6]
Yes (0.00s cpu)

% to jest polecenie:
[eclipse 5]:
        :-lib(ic_sets).
        Zmienna_Zbiorowa :: []..[1,2,3,4,5,6],
        Zmienna_Zbiorowa = [].
% to jest odpowiedź:
[eclipse 6]:
Zmienna_Zbiorowa = []
Yes (0.00s cpu)
```

W ECL^iPS^e CPS zbiór pusty `[]` nie jest elementem dziedziny zbioru, jeżeli nie został *explicite* zadeklarowany w dziedzinie:

```
% to jest polecenie:
[eclipse 7]:
        :-lib(ic_sets).
        Zmienna_Zbiorowa :: [4]..[5,6,7],
        Zmienna_Zbiorowa = [].
% to jest odpowiedź:
[eclipse 8]:
No (0.00s cpu)
```

Zbiór pusty `[]` jest natomiast podzbiorem każdego zbioru. Np.:

```
% to jest polecenie:
[eclipse 9]:
        :-lib(ic_sets).
       [4,5,6,7] includes Zmienna_Zbiorowa,
        Zmienna_Zbiorowa = [].
% to jest odpowiedź:
[eclipse 8]:
X = X{([] .. [4, 5, 6, 7]) : _358{0 .. 4}},
```

gdzie zmienna `_358` przedstawia zakres liczb kardynalnych zbioru `X`.

Dolne ograniczenie nie jest "automatycznie" elementem podzbioru zawierającego dodatkowo elementy ograniczenia górnego, jak to ilustruje poniższy przykład:

```
% to jest polecenie:
[eclipse 9]:
        :-lib(ic_sets).
        Zmienna_Zbiorowa :: [4]..[5,6,7],
        Zmienna_Zbiorowa = [4,5].
% to jest odpowiedź:
[eclipse 8]:
No (0.00s cpu)
```

Aby dolne ograniczenie mogło być elementem podzbioru zawierającego dodatkowo elementy ograniczenia górnego, należy dolne ograniczenie wprowadzić do

górnego ograniczenia:

```
% to jest polecenie:
[eclipse 10]:
       :-lib(ic_sets).
        Zmienna_Zbiorowa :: [4]..[4,5,6,7],
        Zmienna_Zbiorowa=[4,7].
% to jest odpowiedź:
[eclipse 11]:
Zmienna_Zbiorowa = [4, 7]
Yes (0.00s cpu)
```

Ważnym ograniczeniem łączącym zbiory i tablice jest:

```
    weight(?Zbior, ++Tablica_Wag, ?Waga_Zbioru),
```

gdzie `Waga_Zbioru` jest sumą tych wag z `Tablicy_Wag`, które w tej tablicy znajdują się na pozycjach będących elementami zbioru `Zbioru`. Ilustruje to program `5_7_waga_zbioru_1.ecl`:

```
/*1*/  :- lib(ic_sets).

/*2*/   top:-
/*3*/       Zbior = [1,3],
/*4*/       weight(Zbior,[](10,20,30,40,50),Waga_Zbioru),
/*5*/       write("Waga zbioru = "),write(Waga_Zbioru),nl.
```

generujący komunikat:

```
         Waga zbioru = 40
```

Predykatu `weight/3` można użyć do wyznaczania zbiorów o wagach spełniających pewne warunki. Przedstawia to program `5_8_waga_zbioru_2.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_sets).

/*3*/   top:-
/*4*/       ic_sets:(Set :: [].. [1, 2, 3, 4, 5]),
/*5*/       weight(Set,[](10,20,30,40,50),Waga_Zbioru),
```

```
/*6*/       Waga_Zbioru #=< 50,
/*7*/       Waga_Zbioru #>= 35,
/*8*/       insetdomain(Set,_,_,_),
/*9*/       write("Waga zbioru = "),write(Waga_Zbioru),
/*10*/      write("    Zbior = "),write(Set),nl.
```

Predykat `insetdomain/4` jest "zbiorowym" odpowiednikiem predykatu `indomain/1` dla dziedzin liczb całkowitych.

Program ten generuje szereg rozwiązań:

```
Waga zbioru = 40    Zbior = [1, 3]
Waga zbioru = 50    Zbior = [1, 4]
Waga zbioru = 50    Zbior = [2, 3]
Waga zbioru = 40    Zbior = [4]
Waga zbioru = 50    Zbior = [5]
```

5.6.6 Problem plecakowy 2

Jednowymiarowy problem plecakowy można również rozwiązać korzystając ze zbiorów i predykatu `weight/3`, jak to pokazano w przykładzie `5_9_plecak_2.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_sets).
/*3*/  :- lib(branch_and_bound).

/*4*/   top:-
/*5*/       wektor_rozmiarow(Rozmiary),
/*6*/       wektor_wartosci(Wartosci),
/*7*/       wielkosc_plecaka(Wielkosc_plecaka),
/*8*/       ic_sets:(Zbior :: [].. [1, 2, 3, 4, 5]),
/*9*/       weight(Zbior,Rozmiary,Rozmiar),
/*10*/      Rozmiar #=< Wielkosc_plecaka,
/*11*/      weight(Zbior,Wartosci,Wartosc),
/*12*/      Koszt #= -Wartosc,

/*13*/      minimize(insetdomain(Zbior,decreasing,_,_),Koszt),

/*14*/      write("Wartosc = "),writeln(Wartosc),
/*15*/      write("Plecak = "), writeln(Zbior),
/*16*/      write("Rozmiar ="),writeln(Rozmiar).
```

```
/*17*/  wektor_rozmiarow([](52,23,35,15,7)).
/*18*/  wektor_wartosci([](100,60,70,15,15)).
/*19*/  wielkosc_plecaka(60).
```

Program generuje komunikat:

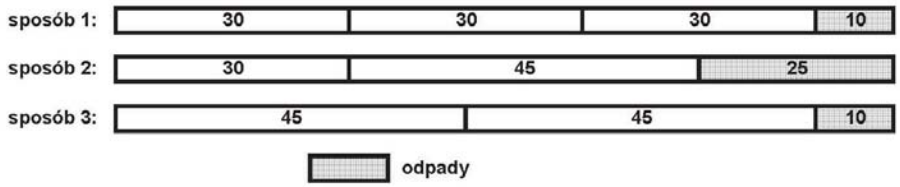
```
Found a solution with cost -90
Found a solution with cost -100
Found a solution with cost -115
Found a solution with cost -130
Found no solution with cost -260.0 .. -131.0
Wartosc = 130
Plecak = [2, 3]
Rozmiar =58,
```

z którego wynika, że optymalny załadunek plecaka to przedmioty `2` i `3`, o łącznych rozmiarach `58` i wartości `130`.

5.6.7 Jak ciąć optymalnie?

Różnorodność problemów konfigurowania jest duża. Kolejną ilustracja tej różnorodności jest problem optymalnego cięcia prętów:

Należy pociąć pręty o długości 1 m tak, by otrzymać 36 prętów o długości 30 cm i 24 pręty o długości 45 cm, minimalizując odpady powstałe przy cięciu. Przykłady możliwych sposobów cięcia pokazuje rysunek 5.7, gdzie możliwe odpady oznaczono odcieniem szarym.



Rysunek 5.7: Sposoby cięcia 1-metrowego pręta

Sposoby cięcia minimalizujące straty wyznacza program `5_10_cut.ecl`, gdzie zmienne `Sp_1`, `Sp_2` i `Sp_3` oznaczają odpowiednio liczbę prętów ciętych odpo-

wiednio sposobem 1, sposobem 2 i sposobem 3.

```
/*1*/  :-lib(ic).
/*2*/  :-lib(branch_and_bound).

/*3*/  top :-
/*4*/  Zmienne = [Sp_1,Sp_2,Sp_3],
/*4*/  Zmienne :: 0..60,

/*5*/  3*Sp_1 + 1*Sp_2 + 0*Sp_3 #>=36,
/*6*/  0*Sp_1 + 1*Sp_2 + 2*Sp_3 #>= 24,

/*7*/  Koszt #= 10*Sp_1 + 25*Sp_2 + 10*Sp_3,

/*8*/  minimize(search(Zmienne,0,first_fail,indomain,complete,[]),Koszt),
/*9*/  writeln("Zmienne":Zmienne ),
/*10*/ writeln("Koszt":Koszt).
```

Program generuje następujący komunikat:

```
Found a solution with cost 900
Found a solution with cost 835
Found a solution with cost 770
Found a solution with cost 705
Found a solution with cost 640
Found a solution with cost 595
Found a solution with cost 540
Found a solution with cost 495
Found a solution with cost 440
Found a solution with cost 395
Found a solution with cost 340
Found a solution with cost 295
Found a solution with cost 240
Found no solution with cost 0.0 .. 239.0

Zmienne : [12, 0, 12]
Koszt : 240
```

A więc 12 prętów należy ciąć sposobem 1 i również 12 prętów należy ciąć sposobem 3.

5.6.8 Powołujemy komisję parlamentarną

Często spotykanym problemem konfigurowania jest problem wyznaczeniu najmniejszego zbioru zawierającego elementy innych zbiorów. Możliwość minimalizacji wynika stąd, że zbiory te mają po części jednakowe elementy. Ilustruje to następujący przykład:

Rządząca w Absurdolandii koalicja popularnych partii "Najpierw Człowiek!" oraz "Raj na Ziemi" stoi wobec zadania wyłonienia zespołu czterech parlamentarzystów do tworzonej (pod naciskiem opinii publicznej) Komisji Parlamentarnej, której zadaniem będzie zbadanie wpływu nieformalnych struktur lobbistycznych na przebieg procesów legislacyjnych. Każda z partii koalicyjnych wytypowała po pięciu parlamentarzystów, z których należy wyłonić zespół czterech parlamentarzystów, obejmujący reprezentacje wszystkich nurtów myśli politycznej i społecznej, kultywowanych w obydwu partiach. Bliższa analiza wykazała, że wytypowani wstępnie parlamentarzyści (którzy - w trosce o ochronę danych osobowych - zostaną zakonspirowani numerami 1, 2, 3,..., 10) reprezentują wszystkie nurty tej myśli. Co więcej, szczęśliwym trafem niektórzy parlamentarzyści mają tak znaczący potencjał intelektualny, że stać ich było na poważny wkład do szeregu nurtów myśli politycznej i społecznej. Słusznie uznano, że powinni oni w Komisji reprezentować wszystkie bliskie im nurty. Z przeprowadzonych analiz wynika rozkład przynależności wytypowanych parlamentarzystów do obydwu partii oraz poszczególnych nurtów myśli politycznej i społecznej, pokazany w tablicy 5.1.

Czy uda się jednak wyłonić zespół czterech parlamentarzystów reprezentujących obydwie partie i wszystkie nurty? Aby odpowiedzieć na to pytanie, postanowiono wyznaczyć najpierw minimalną liczbę parlamentarzystów, reprezentującą wszystkie nurty myśli politycznej i społecznej. Użyto do tego program o nazwie `5_11_komisja.ecl`, dla którego inspiracją był program przedstawiony w witrynie [Kjellerstrand-13]:

```
/*1*/  :-lib(ic).
/*2*/  :-lib(branch_and_bound).

/*3*/  top :-
 % Najpierw zostanie wyznaczona wartość optymalna (Minimum),
 % potem zostaną wyznaczone wszystkie rozwiązania dla tej wartości.
/*4*/  writeln("Wyznaczanie rozwiazania optymalnego:"),
/*5*/  dane(Kto_gdzie),
/*6*/  wybierz_ze_zbiorow(Kto_gdzie, Minimum,_),
/*7*/  writeln("\nWyznaczanie wszystkich rozwiazan optymalnych:"),
```

Parlamentarzyści	Partie koalicji
1, 2, 3, 4, 5	Najpierw Człowiek!
6, 7, 8, 9, 10	Raj na Ziemi
	Nurty myśli politycznej i społecznej
3, 8, 9	Agentura 1
1, 6, 7	Agentura 2
3, 4	Mafia 1
2, 6	Mafia 2
7, 10	Biznesmeni od Hazardu
3, 6	Handlarze Złomu
7	Producenci Hulajnóg
2	Hodowcy Krokodyli
5, 10	Pożyteczni Idioci

Tablica 5.1: Wytypowani parlamentarzyści i wspierane przez nich nurty myśli politycznej i społecznej

```
/*8*/  findall(X, wybierz_ze_zbiorow(Kto_gdzie,  Minimum,X), L),
/*9*/  length(L, Len),
/*10*/ printf("Wyznaczono %d rozwiazania optymalne.\n", [Len]).

/*11*/ wybierz_ze_zbiorow(Kto_gdzie, Minimum, X) :-
/*12*/ dim(Kto_gdzie,[Liczba_grup,Liczba_parlamentarzystow]),
 % Tworzenie listy parlamentarzystów:
/*13*/ dim(X,[Liczba_parlamentarzystow]),
/*14*/ X :: 0..1,

 % wybór parlamentarzystów z każdej grupy:
/*14*/ (
/*15*/ for(I,1,Liczba_grup),
/*16*/ param(Liczba_parlamentarzystow,X,Kto_gdzie)
/*17*/ do
/*18*/   (
/*19*/   for(J,1,Liczba_parlamentarzystow),
/*20*/   fromto(0,We,Wy,Suma),
/*21*/   param(X,Kto_gdzie,I) do
/*22*/   Wy #= We + X[J]*Kto_gdzie[I,J]
/*23*/   ),
/*24*/ Suma #>= 1
/*25*/ ),

 % Minimalizowanie liczby parlamantarzystów:
```

```
/*26*/ flatten_array(X, Zmienne),
/*27*/ Z #= sum(Zmienne),
/*28*/ Z #= Minimum,

 % Zależnie od tego, czy Minimum jest - albo nie jest -
 % uziemione, wyznacza się wszystkie rozwiązania minimalne
 % albo wyznacza się jedno rozwiązanie minimalne:
/*29*/ (
/*30*/ ground(Minimum)
/*31*/ ->
/*32*/ search(Zmienne,0,first_fail,indomain,complete, [])
/*33*/ ;
/*34*/ minimize(search(Zmienne,0,first_fail,indomain,
             complete,[]),Z)
/*35*/ ),
/*36*/ writeln("Minimalna liczba parlamentarzystow":Z),
/*37*/ writeln("Wytypowani parlamentarzysci":X).

/*38*/ dane([](
   [](1, 1, 1, 1, 1, 0, 0, 0, 0, 0),  % Najpierw Człowiek!
   [](0, 0, 0, 0, 0, 1, 1, 1, 1, 1),  % Raj na Ziemi
   [](0, 0, 1, 0, 0, 0, 0, 1, 1, 0),  % Agentura 1
   [](1, 0, 0, 0, 0, 1, 1, 0, 0, 0),  % Agentura 2
   [](0, 0, 1, 1, 0, 0, 0, 0, 0, 0),  % Mafia 1
   [](0, 1, 0, 0, 0, 1, 0, 0, 0, 0),  % Mafia 2
   [](0, 0, 0, 0, 0, 0, 1, 0, 0, 1),  % Biznesmeni od Hazardu
   [](0, 0, 1, 0, 0, 1, 0, 0, 0, 0),  % Handlarze Złomu
   [](0, 0, 0, 0, 0, 0, 1, 0, 0, 0),  % Producenci Hulajnóg
   [](0, 1, 0, 0, 0, 0, 0, 0, 0, 0),  % Hodowcy Krokodyli
   [](0, 0, 0, 0, 1, 0, 0, 0, 0, 1))).% Pożyteczni Idioci
```

Program generuje następujący komunikat:

```
Wyznaczanie rozwiazania optymalnego:
Found a solution with cost 6
Found a solution with cost 4
Found no solution with cost 2.0 .. 3.0
Minimalna liczba parlamentarzystow : 4
Wytypowani parlamentarzysci : [](0, 1, 1, 0, 0, 0, 1, 0, 0, 1)

Wyznaczanie wszystkich rozwiazan optymalnych:
Minimalna liczba parlamentarzystow : 4
Wytypowani parlamentarzysci : [](0, 1, 1, 0, 0, 0, 1, 0, 0, 1)
Minimalna liczba parlamentarzystow : 4
```

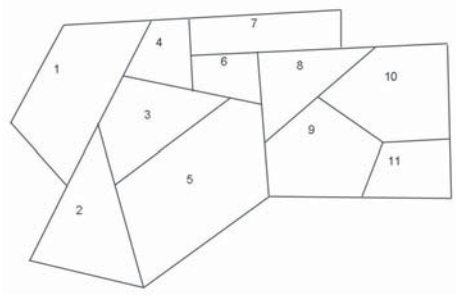
```
Wytypowani parlamentarzysci : [](0, 1, 1, 0, 1, 0, 1, 0, 0, 0)
Wyznaczono 2 rozwiazania optymalne.
```

Otrzymany rezultat jest zarazem dobrą jak i złą wiadomością: dobrą, gdyż czterech parlamentarzystów może reprezentować sobą wszystkie istotne nurty myśli politycznej i społecznej; złą, bo są dwa takie rozwiązania, co będzie z pewnością przyczyną niesnasek w koalicji.

5.6.9 Stacje pogotowia medycznego

Kolejny przykład jest również przykładem często spotykanego problemu wyznaczenia minimalnej liczby elementów zbioru, pozostających w określonych relacjach z innymi elementami tego zbioru.

Urząd Miasta uzyskał znaczne środki na tworzenie sieci nowych Stacji Pogotowia Medycznego (SPM). Miasto składa sie z 11 dzielnic pokazanych na rysunku 5.8. Stacje Pogotowia Medycznego można ulokować w dowolnej dzielnicy, przy czym istnieje zawsze możliwość udzielenia pomocy mieszkańcom tej dzielnicy i dzielnicom sąsiadującym. Konserwatywna większość w Radzie Miejska przegłosowała wniosek żądający minimalizacji liczby zakładanych Stacji Pogotowia Medycznego, a w celu uniknięcia konfliktów kompetencyjnych żądała dodatkowo, by żadna z dzielnic z ulokowaną w niej stacją nie mogła sąsiadować z inną dzielnicą, również posiadająca stację, oraz by dzielnice nie posiadające stacji sąsiadowały tylko z jedną dzielnicą posiadającą stację.



Rysunek 5.8: Plan dzielnic dla lokalizacji Stacji Pogotowia Medycznego

Program `5_12_SPM.ecl` rozwiązuje problem minimalizacji liczby stacji przy

spełnieniu warunku dostarczenia pomocy mieszkańcom wszystkich dzielnic:

```
/*1*/  :-lib(ic).
/*2*/  :-lib(ic_global).
 /*3*/  :-lib(branch_and_bound).
 /*4*/  top:-
/*5*/  Stacje = [S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11],
/*6*/  Stacje :: 0..1,
% Si = 1 - w dzielnicy i-tej zostanie zlokalizowania SPM
% Si = 0 - w dzielnicy i-tej nie zostanie zlokalizowana SPM

% Jeżeli dzielnica 1 ma SPM, to dzielnice 2, 3 i 4 nie mogą mieć SPM-u;
% jeżeli dzielnica 1 nie ma SPM, to jedna z dzielnic 2, 3 lub 4 musi
% mieć SPM:
/*7*/   S1+S2+S3+S4                          #= 1,

% Jeżeli dzielnica 2 ma SPM, to dzielnice 1, 3 i 5 nie mogą mieć SPM-u;
% Jeżeli dzielnica 2 nie ma SPM, to jedna z dzielnic 1, 3 lub 5 musi
% mieć SPM:
/*8*/   S1+S2+S3+   S5                       #= 1,

% Jeżeli dzielnica 3 ma SPM, to dzielnice 1, 2, 4, 5 i 6 nie mogą mieć SPM-u;
% Jeżeli dzielnica 3 nie ma SPM, to jedna z dzielnic 1, 2, 4, 5 lub 6  musi
% mieć SPM:
/*9*/   S1+S2+S3+S4+S5+S6                    #= 1,

% Jeżeli dzielnica 4 ma SPM, to dzielnice 1, 3, 6 i 7 nie mogą mieć SPM-u;
% Jeżeli dzielnica 4 nie ma SPM, to jedna z dzielnic 1, 3, 6 lub 7  musi
% mieć SPM:
/*10*/     S1+   S3+S4+   S6+S7                 #= 1,

% Jeżeli dzielnica 5 ma SPM, to dzielnice 2, 3, 6, 8 i 9 nie mogą mieć SPM-u;
% Jeżeli dzielnica 5 nie ma SPM, to jedna z dzielnic 2, 3, 6, 8 lub 9 musi
% mieć SPM:
/*11*/      S2+S3+   S5+S6+   S8+S9           #= 1,

% Jeżeli dzielnica 6 ma SPM, to dzielnice 2, 4, 5, 7 i 8 nie mogą mieć SPM-u;
% Jeżeli dzielnica 6 nie ma SPM, to jedna z dzielnic 2, 4, 5, 7 lub 8 musi
% mieć SPM:
/*12*/        S3+S4+S5+S6+S7+S8              #= 1,

% Jeżeli dzielnica 7 ma SPM, to dzielnice 4, 6 i 8 nie mogą mieć SPM-u;
% Jeżeli dzielnica 7 nie ma SPM, to jedna z dzielnic 4, 6 lub 8 musi
% mieć SPM:
/*13*/           S4+   S6+S7+S8              #= 1,

% Jeżeli dzielnica 8 ma SPM, to dzielnice 5, 6, 7, 9 i 10 nie mogą mieć SPM-u;
% Jeżeli dzielnica 8 nie ma SPM, to jedna z dzielnic 5, 6, 7, 9 lub 10 musi
```

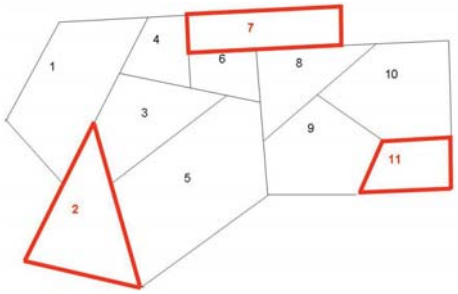
```
% mieć SPM:
/*14*/              S5+S6+S7+S8+S9+S10       #= 1,

% Jeżeli dzielnica 9 ma SPM, to dzielnice 5, 8, 10 i 11 nie mogą mieć SPM-u;
% Jeżeli dzielnica 9 nie ma SPM, to jedna z dzielnic 5, 8, 10 lub 11 musi
% mieć SPM:
/*15*/              S5+      S8+S9+S10+S11   #= 1,

% Jeżeli dzielnica 10 ma SPM, to 8, 9 i 11 nie mogą mieć SPM-u;
% Jeżeli dzielnica 10 nie ma SPM, to jedna z dzielnic 8, 9 lub 11 musi
% mieć SPM:
/*16*/                        S8+S9+S10+S11   #= 1,

% Jeżeli dzielnica 11 ma SPM, to dzielnice 9 i 11 nie mogą mieć SPM-u;
% Jeżeli dzielnica 11 nie ma SPM, to jedna z dzielnic 9 lub 11 musi
% mieć SPM:
/*17*/                           S9+S10+S11   #= 1,

/*18*/ Liczba_stacji #= S1+S2+S3+S4+S5+S6+S7+S8+S9+S10+S11,

/*19*/ sumlist(Stacje,Liczba_stacji),
/*20*/ minimize(labeling(Stacje),Liczba_stacji),

/*21*/write("Minimalna liczba stacji":Liczba_stacji),nl,

/*22*/ write("Stacje: "),write([S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11]),nl,
/*23*/ pisz([S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11]),nl,nl.

/*24*/pisz([S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11]):-
/*25*/     print("SPM-y zostaną zlokalizowane w dzielnicach:"),nl,
/*26*/     ((S1 is 1) -> print(" 1, ") ; print("")),
/*27*/     ((S2 is 1) -> print(" 2, ") ; print("")),
/*28*/     ((S3 is 1) -> print(" 3, ") ; print("")),
/*29*/     ((S4 is 1) -> print(" 4, ") ; print("")),
/*30*/     ((S5 is 1) -> print(" 5, ") ; print("")),
/*31*/     ((S6 is 1) -> print(" 6, ") ; print("")),
/*32*/     ((S7 is 1) -> print(" 7, ") ; print("")),
/*33*/     ((S8 is 1) -> print(" 8, ") ; print("")),
/*34*/     ((S9 is 1) -> print(" 9, ") ; print("")),
/*35*/     ((S10 is 1) -> print(" 10, ") ; print("")),
/*36*/     ((S11 is 1) -> print(" 11, ") ; print("")).
```

Rozwiązaniem jest:

```
Found a solution with cost 3
Found no solution with cost 0.0 .. 2.0
Stacje: [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1]
```

```
SPM-y zostaną zlokalizowane w dzielnicach:  2,  7,  11
```

Przedstawia je rysunek 5.9.



Rysunek 5.9: Dzielnice ze Stacjami Pogotowia Medycznego

Zbadajmy jeszcze, czy otrzymane rozwiazanie jest jedynym rozwiązaniem optymalnym. Do tego służy program `5_13_SPM-y_wszystkie.ecl`, dla którego (podobnie jak w rozdziale) 5.10.1, ustalono liczbę stacji na wartości optymalnej równej 3:

```
/*1*/  :-lib(ic).
/*2*/  :-lib(ic_global).

/*3*/  top:-
/*4*/  Stacje = [S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11],
/*5*/  Stacje :: 0..1,
% Si = 1 - w dzielnicy i-tej zostanie zlokalizowania SPM
% Si = 0 - w dzielnicy i-tej nie zostanie zlokalizowana SPM

% Jeżeli dzielnica 1 ma SPM, to 2, 3 i 4 nie mogą mieć SPM-u;
% jeżeli dzielnica 1 nie ma SPM, to jedna z dzielnic 2, 3 lub 4 musi
%      mieć SPM:
/*6*/   S1+S2+S3+S4                          #= 1,

% Jeżeli dzielnica 2 ma SPM, to dzielnice 1, 3 i 5 nie mogą mieć SPM-u;
% Jeżeli dzielnica 2 nie ma SPM, to jedna z dzielnic 1, 3 lub 5 musi
%      mieć SPM:
/*7*/   S1+S2+S3+   S5                       #= 1,

% Jeżeli dzielnica 3 ma SPM, to dzielnice 1, 2, 4, 5 i 6 nie mogą mieć SPM-u;
% Jeżeli dzielnica 3 nie ma SPM, to jedna z dzielnic 1, 2, 4, 5 lub 6  musi
```

```
%      mieć SPM:
/*8*/   S1+S2+S3+S4+S5+S6                    #= 1,

% Jeżeli dzielnica 4 ma SPM, to dzielnice 1, 3, 6 i 7 nie mogą mieć SPM-u;
% Jeżeli dzielnica 4 nie ma SPM, to jedna z dzielnic 1, 3, 6 lub 7  musi
%      mieć SPM:
/*9*/     S1+   S3+S4+   S6+S7                  #= 1,

% Jeżeli dzielnica 5 ma SPM, to dzielnice 2, 3, 6, 8 i 9 nie mogą mieć SPM-u;
% Jeżeli dzielnica 5 nie ma SPM, to jedna z dzielnic 2, 3, 6, 8 lub 9 musi
%      mieć SPM:
/*10*/      S2+S3+   S5+S6+   S8+S9            #= 1,

% Jeżeli dzielnica 6 ma SPM, to dzielnice 2, 4, 5, 7 i 8 nie mogą mieć SPM-u;
% Jeżeli dzielnica 6 nie ma SPM, to jedna z dzielnic 2, 4, 5, 7 lub 8 musi
%      mieć SPM:
/*11*/        S3+S4+S5+S6+S7+S8               #= 1,

% Jeżeli dzielnica 7 ma SPM, to dzielnice 4, 6 i 8 nie mogą mieć SPM-u;
% Jeżeli dzielnica 7 nie ma SPM, to jedna z dzielnic 4, 6 lub 8 musi
%      mieć SPM:
/*12*/           S4+   S6+S7+S8               #= 1,

% Jeżeli dzielnica 8 ma SPM, to dzielnice 5, 6, 7, 9 i 10 nie mogą mieć SPM-u;
% Jeżeli dzielnica 8 nie ma SPM, to jedna z dzielnic 5, 6, 7, 9 lub 10 musi
%      mieć SPM:
/*13*/              S5+S6+S7+S8+S9+S10        #= 1,

% Jeżeli dzielnica 9 ma SPM, to dzielnice 5, 8, 10 i 11 nie mogą mieć SPM-u;
% Jeżeli dzielnica 9 nie ma SPM, to jedna z dzielnic 5, 8, 10 lub 11 musi
%      mieć SPM:
/*14*/              S5+      S8+S9+S10+S11    #= 1,

% Jeżeli dzielnica 10 ma SPM, to dzielnice 8, 9 i 11 nie mogą mieć SPM-u;
% Jeżeli dzielnica 10 nie ma SPM, to 8, 9 lub 11 musi mieć SPM:
/*15*/                       S8+S9+S10+S11    #= 1,

% Jeżeli dzielnica 11 ma SPM, to dzielnice 9 i 11 nie mogą mieć SPM-u;
% Jeżeli dzielnica 11 nie ma SPM, to jedna z dzielnic 9 lub 11 musi
%      mieć SPM:
/*16*/                          S9+S10+S11    #= 1,

/*17*/  sumlist(Stacje,3),  % Liczba stacji jest optymalna
/*18*/  labeling(Stacje),

/*19*/  write("Stacje mozna zbudowac w dzielnicach:"),
/*20*/   (foreach(Stacja,Stacje),
/*21*/   count(I,1,11)
/*22*/   do
```

```
/*23*/   (Stacja #= 1 -> (write(" "),write(I)); true)
/*24*/    ),nl, fail.

/*25*/ top:-
/*26*/   writeln("To wszystkie rozwiazania").
```

Rozwiązanie tym razem ma postać:

```
Stacje mozna zbudowac w dzielnicach: 2 7 11
To wszystkie rozwiazania
```

Wyznaczone rozwiązanie optymalne jest więc rozwiązaniem pojedynczym.

5.7 Optymalne przyporządkowania

5.7.1 Siedem maszyn - siedem operacji - podejście BO

Dzielenie resursów pomiędzy operacje (patrz rozdział 4.4.7) można również optymalizować. Ilustruje to następujący przykład:

Każda z 7 maszyn może wykonać dowolną z 7 operacji, jednak z różnym kosztem, co przedstawiono w tablicy 5.2.

Maszyna	Operacje						
	1	2	3	4	5	6	7
1	15	23	43	27	76	43	91
2	45	76	32	39	72	37	48
3	56	45	87	75	34	76	29
4	13	45	34	51	52	21	76
5	45	49	18	48	58	98	23
6	23	25	29	39	52	41	12
7	76	98	86	41	34	76	77

Tablica 5.2: Koszty 7 operacji dla 7 maszyn

Należy wyznaczyć taki przydział operacji maszynom, by zminimalizować koszt wykonania operacji. Do tego służy program `5_14_opty77_BO.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).
```

```
      % Uij - Zastosowanie maszyny i dla operacji j:
     % Uij = 1 - maszyna i wykonuje operację j.
     % Uij = 0 - maszyna i nie wykonuje operacji j.

/*3*/   top :-
/*4*/     Zastosowanie_maszyn =
            [U11,U12,U13,U14,U15,U16,U17,
            U21,U22,U23,U24,U25,U26,U27,
            U31,U32,U33,U34,U35,U36,U37,
            U41,U42,U43,U44,U45,U46,U47,
            U51,U52,U53,U54,U55,U56,U57,
            U61,U62,U63,U64,U65,U66,U67,
            U71,U72,U73,U74,U75,U76,U77],
/*5*/     Zastosowanie_maszyn :: 0..1,
/*6*/     Koszt :: 1..700,

/*7*/     U11+U21+U31+U41+U51+U61+U71 #= 1,
/*8*/     U12+U22+U32+U42+U52+U62+U72 #= 1,
/*9*/     U13+U23+U33+U43+U53+U63+U73 #= 1,
/*10*/    U14+U24+U34+U44+U54+U64+U74 #= 1,
/*11*/    U15+U25+U35+U45+U55+U65+U75 #= 1,
/*12*/    U16+U26+U36+U46+U56+U66+U76 #= 1,
/*13*/    U17+U27+U37+U47+U57+U67+U77 #= 1,

/*14*/    U11+U12+U13+U14+U15+U16+U17 #= 1,
/*15*/    U21+U22+U23+U24+U25+U26+U27 #= 1,
/*16*/    U31+U32+U33+U24+U35+U36+U37 #= 1,
/*17*/    U41+U42+U43+U44+U45+U46+U47 #= 1,
/*18*/    U51+U52+U53+U54+U55+U56+U57 #= 1,
/*19*/    U61+U62+U63+U64+U65+U66+U67 #= 1,
/*20*/    U71+U72+U73+U74+U75+U76+U77 #= 1,

/*21*/    Koszt #=
            U11*15 + U12*23 + U13*43 + U14*27 + U15*76 + U16*43 + U17*91 +
            U21*45 + U22*76 + U23*32 + U24*39 + U25*72 + U26*37 + U27*48 +
            U31*56 + U32*45 + U33*87 + U34*75 + U35*34 + U36*76 + U37*29 +
            U41*13 + U42*45 + U43*34 + U44*51 + U45*52 + U46*21 + U47*76 +
            U51*45 + U52*49 + U53*18 + U54*48 + U55*58 + U56*98 + U57*23 +
            U61*23 + U62*25 + U63*29 + U64*39 + U65*52 + U66*41 + U67*12 +
            U71*76 + U72*98 + U73*86 + U74*41 + U75*34 + U76*76 + U77*77,

/*22*/    bb_min(labeling(
            [U11,U12,U13,U14,U15,U16,U17,
            U21,U22,U23,U24,U25,U26,U27,
            U31,U32,U33,U34,U35,U36,U37,
            U41,U42,U43,U44,U45,U46,U47,
            U51,U52,U53,U54,U55,U56,U57,
            U61,U62,U63,U64,U65,U66,U67,
            U71,U72,U73,U74,U75,U76,U77]),
```

```
            Koszt,bb_options with [strategy:step]),

/*23*/    write("Koszt calkowity: "),writeln(Koszt),
/*24*/    przedstaw_wyniki(1,[U11,U12,U13,U14,U15,U16,U17],[15,23,43,27,76,43,91]),
/*25*/    przedstaw_wyniki(2,[U21,U22,U23,U24,U25,U26,U27],[45,76,32,39,72,37,48]),
/*26*/    przedstaw_wyniki(3,[U31,U32,U33,U34,U35,U36,U37],[56,45,87,75,34,76,29]),
/*27*/    przedstaw_wyniki(4,[U41,U42,U43,U44,U45,U46,U47],[13,45,34,51,52,21,76]),
/*28*/    przedstaw_wyniki(5,[U51,U52,U53,U54,U55,U56,U57],[45,49,18,48,58,98,23]),
/*29*/    przedstaw_wyniki(6,[U61,U62,U63,U64,U65,U66,U67],[23,25,29,39,52,41,12]),
/*30*/    przedstaw_wyniki(7,[U71,U72,U73,U74,U75,U76,U77],[76,98,86,41,34,76,77]),
/*31*/     fail.

/*32*/  top:-
/*33*/    writeln("To wszystko!").

/*34*/  przedstaw_wyniki(M,U,C):-
/*35*/    element(N, U, 1),
/*36*/    element(N, C, Op_Koszt),
/*37*/    write("Maszyna "),write(M),write(" wykonuje operacje "),write(N),
                    write(" z kosztem "),write(Op_Koszt),writeln(".").
```

Program generuje komunikat:

```
Found a solution with cost 332
Found a solution with cost 309
Found a solution with cost 307
Found a solution with cost 289
Found a solution with cost 259
Found a solution with cost 252
Found a solution with cost 222
Found a solution with cost 211
Found a solution with cost 183
Found a solution with cost 178
Found no solution with cost 1.0 .. 177.0
Koszt calkowity: 178
Maszyna 1 wykonuje operacje 2 z kosztem 23.
Maszyna 2 wykonuje operacje 6 z kosztem 37.
Maszyna 3 wykonuje operacje 5 z kosztem 34.
Maszyna 4 wykonuje operacje 1 z kosztem 13.
Maszyna 5 wykonuje operacje 3 z kosztem 18.
Maszyna 6 wykonuje operacje 7 z kosztem 12.
```

```
Maszyna 7 wykonuje operacje 4 z kosztem 41.
To wszystko!
```

Dla sprawdzenia unikatowości optymalnego rozwiązania można zastosować program `5_15_opty77_wszystkie_BO.ecl`. W programie tym koszt jest ustalony na poprzednio wyliczonej wartości optymalnej `178`, a program szuka wszystkich rozwiązań dopuszczalnych. Program `5_15_opty77_wszystkie_BO.ecl` jest następujący:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).
       % Uij - Zastosowanie maszyny i dla operacji j:
      % Uij = 1 - maszyna i wykonuje operację j.
      % Uij = 0 - maszyna i nie wykonuje operacji j.

/*3*/   top :-
/*4*/     Zastosowanie_maszyn =
             [U11,U12,U13,U14,U15,U16,U17,
             U21,U22,U23,U24,U25,U26,U27,
             U31,U32,U33,U34,U35,U36,U37,
             U41,U42,U43,U44,U45,U46,U47,
             U51,U52,U53,U54,U55,U56,U57,
             U61,U62,U63,U64,U65,U66,U67,
             U71,U72,U73,U74,U75,U76,U77],
/*5*/     Zastosowanie_maszyn :: 0..1,
/*6*/     Koszt is 178,

/*7*/     U11+U21+U31+U41+U51+U61+U71 #= 1,
/*8*/     U12+U22+U32+U42+U52+U62+U72 #= 1,
/*9*/     U13+U23+U33+U43+U53+U63+U73 #= 1,
/*10*/    U14+U24+U34+U44+U54+U64+U74 #= 1,
/*11*/    U15+U25+U35+U45+U55+U65+U75 #= 1,
/*12*/    U16+U26+U36+U46+U56+U66+U76 #= 1,
/*13*/    U17+U27+U37+U47+U57+U67+U77 #= 1,

/*14*/    U11+U12+U13+U14+U15+U16+U17 #= 1,
/*15*/    U21+U22+U23+U24+U25+U26+U27 #= 1,
/*16*/    U31+U32+U33+U24+U35+U36+U37 #= 1,
/*17*/    U41+U42+U43+U44+U45+U46+U47 #= 1,
/*18*/    U51+U52+U53+U54+U55+U56+U57 #= 1,
/*19*/    U61+U62+U63+U64+U65+U66+U67 #= 1,
/*20*/    U71+U72+U73+U74+U75+U76+U77 #= 1,

/*21*/    Koszt #=
            U11*15 + U12*23 + U13*43 + U14*27 + U15*76 + U16*43 + U17*91 +
            U21*45 + U22*76 + U23*32 + U24*39 + U25*72 + U26*37 + U27*48 +
```

```
            U31*56 + U32*45 + U33*87 + U34*75 + U35*34 + U36*76 + U37*29 +
            U41*13 + U42*45 + U43*34 + U44*51 + U45*52 + U46*21 + U47*76 +
            U51*45 + U52*49 + U53*18 + U54*48 + U55*58 + U56*98 + U57*23 +
            U61*23 + U62*25 + U63*29 + U64*39 + U65*52 + U66*41 + U67*12 +
            U71*76 + U72*98 + U73*86 + U74*41 + U75*34 + U76*76 + U77*77,

/*22*/    labeling(
            [U11,U12,U13,U14,U15,U16,U17,
            U21,U22,U23,U24,U25,U26,U27,
            U31,U32,U33,U34,U35,U36,U37,
            U41,U42,U43,U44,U45,U46,U47,
            U51,U52,U53,U54,U55,U56,U57,
            U61,U62,U63,U64,U65,U66,U67,
            U71,U72,U73,U74,U75,U76,U77]),

/*22*/    write("Koszt calkowity: "),writeln(Koszt),
/*23*/    przedstaw_wyniki(1,[U11,U12,U13,U14,U15,U16,U17],[15,23,43,27,76,43,91]),
/*24*/    przedstaw_wyniki(2,[U21,U22,U23,U24,U25,U26,U27],[45,76,32,39,72,37,48]),
/*25*/    przedstaw_wyniki(3,[U31,U32,U33,U34,U35,U36,U37],[56,45,87,75,34,76,29]),
/*26*/    przedstaw_wyniki(4,[U41,U42,U43,U44,U45,U46,U47],[13,45,34,51,52,21,76]),
/*27*/    przedstaw_wyniki(5,[U51,U52,U53,U54,U55,U56,U57],[45,49,18,48,58,98,23]),
/*28*/    przedstaw_wyniki(6,[U61,U62,U63,U64,U65,U66,U67],[23,25,29,39,52,41,12]),
/*29*/    przedstaw_wyniki(7,[U71,U72,U73,U74,U75,U76,U77],[76,98,86,41,34,76,77]),
/*30*/    fail.

/*31*/  top:-
/*32*/    writeln("To wszystko!").

/*33*/  przedstaw_wyniki(M,U,C):-
/*34*/    element(N, U, 1),
/*35*/    element(N, C, Op_Koszt),
/*36*/    write("Maszyna "),write(M),write(" wykonuje operacje "),write(N),
                    write(" z kosztem "),write(Op_Koszt),writeln(".").
```

Okazuje się, że wyznaczone poprzednio rozwiązanie optymalne jest unikatowe. Program generuje komunikat:

```
Koszt calkowity: 178
Maszyna 1 wykonuje operacje 2 z kosztem 23.
Maszyna 2 wykonuje operacje 6 z kosztem 37.
Maszyna 3 wykonuje operacje 5 z kosztem 34.
Maszyna 4 wykonuje operacje 1 z kosztem 13.
Maszyna 5 wykonuje operacje 3 z kosztem 18.
Maszyna 6 wykonuje operacje 7 z kosztem 12.
```

```
Maszyna 7 wykonuje operacje 4 z kosztem 41.
To wszystko!
```

5.7.2 Siedem maszyn - siedem operacji - podejście CLP

Tak jak poprzednio, podejście *CLP* jest dla zagadnienia z rozdziału 5.7.1 bardziej oszczędne w odniesieniu do liczby potrzebnych zmiennych. Ilustruje to program `5_16_opty77_CLP.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).
 /*3*/   top :-
/*4*/       [O1,O2,O3,O4,O5,O6,O7] :: 1..7,
/*5*/       [K1,K2,K3,K4,K5,K6,K7] :: 1..100,

/*6*/       Koszt :: 1..700,
/*7*/       alldifferent([O1,O2,O3,O4,O5,O6,O7]),

/*8*/       element(O1,[15,23,43,27,76,43,91],K1),
/*9*/       element(O2,[45,76,32,39,72,37,48],K2),
  % Jeżeli O2 = 6, to maszyna 2 wykonuje operację 6 przy kosztach K2 = 37.
/*10*/      element(O3,[56,45,87,75,34,76,29],K3),
/*11*/      element(O4,[13,45,34,51,52,21,76],K4),
/*12*/      element(O5,[45,49,18,48,58,98,23],K5),
/*13*/      element(O6,[23,25,29,39,52,41,12],K6),
/*14*/      element(O7,[76,98,86,41,34,76,77],K7),

/*
element(Oi,[K1i,K2i,K3i,K4i,K5i,K6i,K7i],KOi)
element(Numer_operacji_wykonywanej_przez_maszynę_i,
         Lista_kosztow_operacji_wykonywanych_przez_maszynę_i,
              Koszt_operacji_z_listy_wskazany_przez_numer_operacji)

 Maszyna i-ta wykonuje operację o numerze Oi z listy
 numerów operacji z kosztem KOi znajdującym się na pozycji Oi
 w liście kosztów dla maszyny i-tej.
*/

/*15*/      Koszt #=  K1+K2+K3+K4+K5+K6+K7,
/*16*/      bb_min(labeling([O1,O2,O3,O4,O5,O6,O7]),Koszt,
                bb_options with [strategy:step]),
/*17*/      przedstaw_wyniki([O1,K1,O2,K2,O3,K3,O4,K4,O5,K5,
                O6,K6,O7,K7],1),
/*18*/      write("Koszt calkowity: "),write(Koszt).
```

```
/*19*/  przedstaw_wyniki([],_).
/*20*/  przedstaw_wyniki([A,B|R],N):-
/*21*/      write("Maszyna "),write(N),write(" wykonuje operacje "),
/*22*/      write(A),write(" przy kosztach "),write(B),write("."),nl,
/*23*/      M is N+1,
/*24*/      przedstaw_wyniki(R,M).
```

Program generuje komunikat pokazujący kolejne poprawy minimalnego wskaźnika jakości:

```
Found a solution with cost 405
Found a solution with cost 375
Found a solution with cost 373
Found a solution with cost 327
Found a solution with cost 293
Found a solution with cost 251
Found a solution with cost 217
Found a solution with cost 207
Found a solution with cost 204
Found a solution with cost 191
Found a solution with cost 184
Found a solution with cost 181
Found a solution with cost 178
Found no solution with cost 153.0..177.0
Maszyna 1 wykonuje operacje 2 przy kosztach 23.
Maszyna 2 wykonuje operacje 6 przy kosztach 37.
Maszyna 3 wykonuje operacje 5 przy kosztach 34.
Maszyna 4 wykonuje operacje 1 przy kosztach 13.
Maszyna 5 wykonuje operacje 3 przy kosztach 18.
Maszyna 6 wykonuje operacje 7 przy kosztach 12.
Maszyna 7 wykonuje operacje 4 przy kosztach 41.
Koszt calkowity: 178
```

W przypadku tym gałęzie drzewa poszukiwań są inne aniżeli w przypadku program `5_14_opty77_BO.ecl`: ilustruje to różnica komunikatów *Found a solution with cost*. Różnice programów `5_14_opty77_BO.ecl` i `5_16_opty77_CLP.ecl` skutkują bowiem odmiennymi strategiami poszukiwań przy dochodzeniu do tego samego rozwiązania.
Dla sprawdzenia unikatowości optymalnego rozwiązania można zastosować pro-

gram `5_17_opty77_wszystkie_CLP.ecl`. W programie tym ponownie koszt jest ustalony na poprzednio wyliczonej wartości optymalnej `178`, a program szuka wszystkich rozwiązań dopuszczalnych.

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).
 /*3*/   top :-
/*4*/     [O1,O2,O3,O4,O5,O6,O7] :: 1..7,
/*5*/     [K1,K2,K3,K4,K5,K6,K7] :: 1..100,
/*6*/     Koszt is 178,
/*7*/     alldifferent([O1,O2,O3,O4,O5,O6,O7]),
/*8*/     element(O1,[15,23,43,27,76,43,91],K1),
/*9*/     element(O2,[45,76,32,39,72,37,48],K2),
/*10*/    element(O3,[56,45,87,75,34,76,29],K3),
/*11*/    element(O4,[13,45,34,51,52,21,76],K4),
/*12*/    element(O5,[45,49,18,48,58,98,23],K5),
/*13*/    element(O6,[23,25,29,39,52,41,12],K6),
/*14*/    element(O7,[76,98,86,41,34,76,77],K7),

/*15*/    Koszt #= K1+K2+K3+K4+K5+K6+K7,
/*16*/    labeling([O1,O2,O3,O4,O5,O6,O7]),
/*17*/    przedstaw_wyniki([O1,K1,O2,K2,O3,K3,O4,K4,O5,K5,O6,K6,O7,K7],1),
/*18*/    write("Koszt calkowity: "),writeln(Koszt),
/*19*/    fail.

/*20*/  top:-
/*21*/    writeln("To wszystko!").

/*22*/  przedstaw_wyniki([],_).
/*23*/  przedstaw_wyniki([A,B|R],N):-
/*24*/    write("Maszyna "),write(N),write(" wykonuje operacje "),
/*25*/    write(A),write(" przy kosztach "),write(B),write("."),nl,
/*26*/    M is N+1,
/*27*/    przedstaw_wyniki(R,M).
```

Program generuje komunikat:

```
Maszyna 1 wykonuje operacje 2 przy kosztach 23.
Maszyna 2 wykonuje operacje 6 przy kosztach 37.
Maszyna 3 wykonuje operacje 5 przy kosztach 34.
Maszyna 4 wykonuje operacje 1 przy kosztach 13.
Maszyna 5 wykonuje operacje 3 przy kosztach 18.
Maszyna 6 wykonuje operacje 7 przy kosztach 12.
Maszyna 7 wykonuje operacje 4 przy kosztach 41.
```

```
Koszt calkowity:  178
To wszystko!
```

Jest rzeczą oczywistą, że komunikat ten jest identyczny z generowanym w przypadku programu `5_15_opty77_wszystkiel_BO.ecl`.

5.7.3 Plan przewozu kopalin 1

Zagadnienia marszrutyzacji, będące sztandarowymi aplikacjami badań operacyjnych, są w istocie swojej zagadnieniami przyporządkowania. Są one również wdzięcznym tematem przykładów z *CLP*. Rozpatrzmy następujący przykład:

Trzy kopalnie k1, k2 i k3 dostarczają kopalin do pięciu składów s1, s2, s3, s4 i s5 położonych w różnych miejscowościach. Każdy ze składów może przyjąć do 400 t kopalin miesięcznie, natomiast możliwości wydobywcze kopalń wynoszą K1=600 t, K2=K3=700 t kopalin miesięcznie. Koszty wydobycia 1 tony kopalin wynoszą odpowiednio 108, 96 i 102 JP. Koszty transportu 1 tony kopalin pomiędzy kopalniami a składami przedstawia tablica 5.3.

Kopalnia	Skład				
	s1	s2	s3	s4	s5
k1	14	5	9	24	15
k2	30	24	11	8	19
k3	9	22	15	7	18

Tablica 5.3: Koszty transportu tony kopalin

Ile kopalin powinny wyprodukować kopalnie i ile kopalin dostarczyć do poszczególnych składów, by zminimalizować całkowity koszt wydobycia i transportu? Odpowiedni program (`5_18_kopalnie_1.ecl`) ma postać:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).
 /*3*/   top :-

% kopalnia k1 wyprodukuje A1 ton na skład s1,
% A2 ton na skład s2,...,itd.:
/*4*/       [A1,A2,A3,A4,A5] :: 0..600,
% kopalnia k2 wyprodukuje B1 ton na skład s1,
```

```
% B2 ton na skład s2,...,itd:
/*5*/       [B1,B2,B3,B4,B5] :: 0..700,
% kopalnia k3 wyprodukuje C1 ton na skład s1,
% C2 ton na skład s2,...,itd.:
/*6*/       [C1,C2,C3,C4,C5] :: 0..700,

/*7*/       Koszt :: 0..300000,

/*8*/       A1+A2+A3+A4+A5 #=600,    % wydobycie kopalin w kopalni k1
/*9*/       B1+B2+B3+B4+B5 #=700,    % wydobycie kopalin w kopalni k2
/*10*/      C1+C2+C3+C4+C5 #=700,    % wydobycie kopalin w kopalni k3

/*11*/      A1+B1+C1 #=400,         % Pojemność składu s1
/*12*/      A2+B2+C2 #=400,         % Pojemność składu s2
/*13*/      A3+B3+C3 #=400,         % Pojemność składu s3
/*14*/      A4+B4+C4 #=400,         % Pojemność składu s4
/*15*/      A5+B5+C5 #=400,         % Pojemność składu s5

% Koszt będący sumą kosztów transportu i wydobycia:
/*16*/      Koszt #= 14*A1+5*A2+9*A3+24*A4+15*A5+
                     30*B1+24*B2+11*B3+8*B4+19*B5+
                     9*C1+22*C2+15*C3+7*C4+18*C5+
                     108*A1+108*A2+108*A3+108*A4+108*A5+
                     96*B1+96*B2+96*B3+96*B4+96*B5+
                     102*C1+102*C2+102*C3+102*C4+102*C5,

/*17*/      bb_min(search([A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,
            C1,C2,C3,C4,C5],0,first_fail,indomain,complete,[]),
                     Koszt,bb_options with [strategy:continue]),

/*18*/write("Kopalnia k1 wywiozla do skladow:"),nl,
/*19*/      write("A1 = "),write(A1),write(" ton."),nl,
/*20*/      write("A2 = "),write(A2),write(" ton."),nl,
/*21*/      write("A3 = "),write(A3),write(" ton."),nl,
/*22*/      write("A4 = "),write(A4),write(" ton."),nl,
/*23*/      write("A5 = "),write(A5),write(" ton."),nl,nl,

/*24*/write("Kopalnia k2 wywiozla do skladow:"),nl,
/*25*/      write("B1 = "),write(B1),write(" ton."),nl,
/*26*/      write("B2 = "),write(B2),write(" ton."),nl,
/*27*/      write("B3 = "),write(B3),write(" ton."),nl,
/*28*/      write("B4 = "),write(B4),write(" ton."),nl,
/*29*/      write("B5 = "),write(B5),write(" ton."),nl,nl,

/*30*/write("Kopalnia k3 wywiozla do skladow:"),nl,
/*31*/      write("C1 = "),write(C1),write(" ton."),nl,
/*32*/      write("C2 = "),write(C2),write(" ton."),nl,
/*33*/      write("C3 = "),write(C3),write(" ton."),nl,
/*34*/      write("C4 = "),write(C4),write(" ton."),nl,
```

```
/*35*/      write("C5 = "),write(C5),write(" ton."),nl,nl,

/*36*/write("Laczne minimalne koszty wydobycia i transportu: "),
          write(Koszt),nl,nl.
```

Program ten szuka rozwiązania bardzo powoli, generując ciąg komunikatów:

```
Found a solution with cost 233000
Found a solution with cost 232999
Found a solution with cost 232998
Found a solution with cost 232997
...
Found a solution with cost 232964
Found a solution with cost 232963
Found a solution with cost 232962
Found a solution with cost 232961
Found a solution with cost 232960
...
```

Musimy bardzo długo czekać na rozwiązanie:

```
...
Found a solution with cost 223100
Found no solution with cost 0.0 .. 223000.0

Kopalnia k1 wywiozla do skladow:
A1 = 000  ton kopalin.
A2 = 400  ton kopalin.
A3 = 000  ton kopalin.
A4 = 000  ton kopalin.
A5 = 200  ton kopalin.

Kopalnia k2 wywiozla do skladow:
B1 = 000  ton kopalin.
B2 = 000  ton kopalin.
B3 = 400  ton kopalin.
B4 = 100  ton kopalin.
B5 = 200  ton kopalin.

Kopalnia k3 wywiozla do skladow:
C1 = 400  ton kopalin.
C2 = 000  ton kopalin.
C3 = 000  ton kopalin.
C4 = 300  ton kopalin.
C5 = 000  ton kopalin.

Laczne minimalne koszty wydobycia i transportu: 223100
```

5.7.4 Plan przewozu kopalin 2

Powolność działania programu `5_10_kopalnie_1` jest w pewnym stopniu spowodowana rozległością dziedzin deklarowanych w liniach `/*4*/`, `/*5*/` i `/*6*/`. Aby przyśpieszyć znalezienie rozwiązania, można wyrazić dziedziny w setkach ton, zastępując linie `/*4*/`,..., `/*15*/` następującymi:

```
% kopalnia k1 wyprodukuje A1 setek ton na skład s1,
% A2 setek ton na skład s2,...,itd.:
/*4*/       [A1,A2,A3,A4,A5] :: 0..6,
% kopalnia k2 wyprodukuje B1 setek ton na skład s1,
% B2 setek ton na skład s2,...,itd.:
/*5*/       [B1,B2,B3,B4,B5] :: 0..7,
% kopalnia k3 wyprodukuje B1 setek ton na skład s1,
% C2 setek ton na skład s2,...,itd.:
/*6*/       [C1,C2,C3,C4,C5] :: 0..7,
/*7*/       Koszt :: 0..2500,    %koszt dla setek ton

/*8*/       A1+A2+A3+A4+A5 #=6,  % wydobycie kopalin (setki ton) w k1
/*9*/       B1+B2+B3+B4+B5 #=7,  % wydobycie kopalin (setki ton) w k2
/*10*/      C1+C2+C3+C4+C5 #=7,  % wydobycie kopalin (setki ton) w k3

/*11*/      A1+B1+C1 #=4,      % Pojemność (setki ton) składu s1
/*12*/      A2+B2+C2 #=4,      % Pojemność (setki ton) składu s2
/*13*/      A3+B3+C3 #=4,      % Pojemność (setki ton) składu s3
/*14*/      A4+B4+C4 #=4,      % Pojemność (setki ton) składu s4
/*15*/      A5+B5+C5 #=4,      % Pojemność (setki ton) składu s5
```

Dziedzina kosztu została wyrażona dla setek ton i zmniejszona w świetle wyników uzyskanych przez program `5_18_kopalnie_1.ecl`. Wprowadzając ponadto oczywiste zmiany do linii `/*18*/,...,/*36*/`, otrzymuje się program `5_19_kopalnie_2.ecl`:

```
/*1*/   :- lib(ic).
/*2*/   :- lib(branch_and_bound).

/*3*/   top :-
% kopalnia k1 wyprodukuje A1 ton na skład s1,
% A2 ton na skład s2,...:
/*4*/  [A1,A2,A3,A4,A5] :: 0..6,
% kopalnia k2 wyprodukuje B1 ton na skład s1,
% B2 ton na skład s2,...:
/*5*/  [B1,B2,B3,B4,B5] :: 0..7,
% kopalnia k3 wyprodukuje C1 ton na skład s1,
% C2 ton na skład s2,...:
```

```
/*6*/  [C1,C2,C3,C4,C5] :: 0..7,
/*7*/  Koszt :: 0..2500,
/*8*/  A1+A2+A3+A4+A5 #=6, % wydobycie w kopalni k1
/*9*/  B1+B2+B3+B4+B5 #=7, % wydobycie w kopalni k2
/*10*/  C1+C2+C3+C4+C5 #=7, % wydobycie w kopalni k3
/*11*/  A1+B1+C1 #=4, % Pojemność składu s1
/*12*/  A2+B2+C2 #=4, % Pojemność składu s2
/*13*/  A3+B3+C3 #=4, % Pojemność składu s3
/*14*/  A4+B4+C4 #=4, % Pojemność składu s4
/*15*/  A5+B5+C5 #=4, % Pojemność składu s5

 %Koszt będący suma kosztów transportu i wydobycia:
/*16*/ Koszt #= 14*A1+5*A2+9*A3+24*A4+15*A5+30*B1+24*B2+11*B3+8*B4+19*B5+
       9*C1+22*C2+15*C3+7*C4+18*C5+ 108*A1+108*A2+108*A3+108*A4+108*A5+
       96*B1+96*B2+96*B3+96*B4+96*B5+102*C1+102*C2+102*C3+102*C4+102*C5,

/*17*/   bb_min(labeling([A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,C1,C2,C3,C4,C5]),
                          Koszt,bb_options with [strategy:step]),nl,nl,

/*18*/  write('Kopalnia K1 wywiozla do skladow:'),nl,
/*19*/  write('A1 = '),write(A1),write('00  ton  kopalin.'),nl,
/*20*/  write('A2 = '),write(A2),write('00  ton  kopalin.'),nl,
/*21*/  write('A3 = '),write(A3),write('00  ton  kopalin.'),nl,
/*22*/  write('A4 = '),write(A4),write('00  ton  kopalin.'),nl,
/*23*/  write('A5 = '),write(A5),write('00  ton  kopalin.'),nl,nl,

/*24*/  write('Kopalnia K2 wywiozla do skladow:'),nl,
/*25*/  write('B1 = '),write(B1),write('00  ton  kopalin.'),nl,
/*26*/  write('B2 = '),write(B2),write('00  ton  kopalin.'),nl,
/*27*/  write('B3 = '),write(B3),write('00  ton  kopalin.'),nl,
/*28*/  write('B4 = '),write(B4),write('00  ton  kopalin.'),nl,
/*29*/  write('B5 = '),write(B5),write('00  ton  kopalin.'),nl,nl,

/*30*/  write('Kopalnia K3 wywiozla do skladow:'),nl,
/*31*/  write('C1 = '),write(C1),write('00  ton  kopalin.'),nl,
/*32*/  write('C2 = '),write(C2),write('00  ton  kopalin.'),nl,
/*33*/  write('C3 = '),write(C3),write('00  ton  kopalin.'),nl,
/*34*/  write('C4 = '),write(C4),write('00  ton  kopalin.'),nl,
/*35*/  write('C5 = '),write(C5),write('00  ton  kopalin.'),nl,nl,
/*36*/  write('Laczne minimalne koszty wydobycia i transportu = '),
             write(Koszt),write('00'),nl,nl.
```

Program ten generuje "błyskawicznie" komunikat:

```
Found a solution with cost 2328
Found a solution with cost 2310
```

```
Found a solution with cost 2292
...
Found a solution with cost 2241
Found a solution with cost 2236
Found a solution with cost 2231
Found no solution with cost 0.0 .. 2230.0

Kopalnia k1 wywiozla do skladow:
A1 = 000  ton kopalin.
A2 = 400  ton kopalin.
A3 = 000  ton kopalin.
A4 = 000  ton kopalin.
A5 = 200  ton kopalin.

Kopalnia k2 wywiozla do skladow:
B1 = 000  ton kopalin.
B2 = 000  ton kopalin.
B3 = 400  ton kopalin.
B4 = 100  ton kopalin.
B5 = 200  ton kopalin.

Kopalnia k3 wywiozla do skladow:
C1 = 400  ton kopalin.
C2 = 000  ton kopalin.
C3 = 000  ton kopalin.
C4 = 300  ton kopalin.
C5 = 000  ton kopalin.

Laczne minimalne koszty wydobycia i transportu: 223100
```

Jest rzeczą oczywistą, że zastosowane przeskalowanie zmiennych nie dla każdego problemu jest możliwe.

5.7.5 Plan przewozu kopalin 3

Przykłady omawiane w rozdziałach 5.7.3 i 5.7.4 są przykładami problemów *programowania matematycznego całkowitoliczbowego*: wskaźnik jakości jest liniową funkcją zmiennych decyzyjnych całkowitoliczbowych, a ograniczenia są układem równań lub nierówności liniowych względem zmiennych decyzyjnych. Dla rozwiązywania tego typu problemów ECL^iPS^e CPS udostępnia bardzo efektywny *solwer* o nazwie *eplex*. Wymaga on, by symbole operacji arytmetycznych i relacyjnych dla zmiennych całkowitych były prefiksowane symbolem `$`. Jego zastosowanie zostanie zilustrowane znanym już problemem przewozu kopalin (zob.`5_20_kopalnie_3.ecl`):

```
/*1*/   :- lib(eplex).

/*2*/   top :-
/*3*/       Zmienne = [A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,C1,C2,C3,C4,C5],
/*4*/       Zmienne $:: 0.0..1.0Inf,
 % Domyślną dziedziną dla wszystkich zmiennych przykładu rozwiązywanego
 % za pomocą solwera eplex jest (-1.0Inf..1.0Inf).

/*5*/       integers(Zmienne),
  % Tak deklaruje się całkowitoliczbowość poszukiwanego rozwiązania

  % wydobycie kopalin w kopalni K1:
/*6*/       A1+A2+A3+A4+A5 $=600,
  % wydobycie kopalin w kopalni K2:
/*7*/       B1+B2+B3+B4+B5 $=700,
  % wydobycie kopalin w kopalni K3:
/*8*/       C1+C2+C3+C4+C5 $=700,

  % Pojemności składów:
/*9*/       A1+B1+C1 $=400,
/*10*/      A2+B2+C2 $=400,
/*11*/      A3+B3+C3 $=400,
/*12*/      A4+B4+C4 $=400,
/*13*/      A5+B5+C5 $=400,

/*14*/      Koszt $= 14*A1+5*A2+9*A3+24*A4+15*A5+
      30*B1+24*B2+11*B3+8*B4+19*B5+9*C1+22*C2+15*C3+7*C4+18*C5+
      108*A1+108*A2+108*A3+108*A4+108*A5+96*B1+96*B2+96*B3+96*B4+
      96*B5+102*C1+102*C2+102*C3+102*C4+102*C5,

/*15*/      eplex_solver_setup(min(Koszt)),
/*16*/      eplex_solve(Koszt),

/*17*/     write("Kopalnia k1 wywiozla do skladow:"),nl,
/*18*/     write("A1 = "),write(A1),write(" ton."),nl,
/*19*/     write("A2 = "),write(A2),write(" ton."),nl,
/*20*/     write("A3 = "),write(A3),write(" ton."),nl,
/*21*/     write("A4 = "),write(A4),write(" ton."),nl,
/*22*/     write("A5 = "),write(A5),write(" ton."),nl,nl,

/*23*/     write("Kopalnia k2 wywiozla do skladow:"),nl,
/*24*/     write("B1 = "),write(B1),write(" ton."),nl,
/*25*/     write("B2 = "),write(B2),write(" ton."),nl,
/*26*/     write("B3 = "),write(B3),write(" ton."),nl,
/*27*/     write("B4 = "),write(B4),write(" ton."),nl,
/*28*/     write("B5 = "),write(B5),write(" ton."),nl,nl,
```

```
/*29*/      write("Kopalnia k3 wywiozla do skladow:"),nl,
/*30*/      write("C1 = "),write(C1),write(" ton."),nl,
/*31*/      write("C2 = "),write(C2),write(" ton."),nl,
/*32*/      write("C3 = "),write(C3),write(" ton."),nl,
/*33*/      write("C4 = "),write(C4),write(" ton."),nl,
/*34*/      write("C5 = "),write(C5),write(" ton."),nl,nl.
```

Rozwiązanie generowane po czasie 0.12 s przez *eplex* zawiera trójki o strukturze
`Kres_dolny..Kres_górny @ Wartość_zmiennej`,
w której interesuje nas jedynie `Wartość_zmiennej`:

```
Kopalnia k1 wywiozla do skladow:
A1 = _6066{0.0 .. 1.79769313486232e+308 @ 0.0}
A2 = _6050{0.0 .. 1.79769313486232e+308 @ 400.0}
A3 = _6034{0.0 .. 1.79769313486232e+308 @ 0.0}
A4 = _6018{0.0 .. 1.79769313486232e+308 @ 0.0}
A5 = _6002{0.0 .. 1.79769313486232e+308 @ 200.0}

Kopalnia k2 wywiozla do skladow:
B1 = _5986{0.0 .. 1.79769313486232e+308 @ 0.0}
B2 = _5970{0.0 .. 1.79769313486232e+308 @ 0.0}
B3 = _5954{0.0 .. 1.79769313486232e+308 @ 400.0}
B4 = _5938{0.0 .. 1.79769313486232e+308 @ 300.0}
B5 = _5922{0.0 .. 1.79769313486232e+308 @ 0.0}

Kopalnia k3 wywiozla do skladow:
C1 = _5906{0.0 .. 1.79769313486232e+308 @ 400.0}
C2 = _5890{0.0 .. 1.79769313486232e+308 @ 0.0}
C3 = _5874{0.0 .. 1.79769313486232e+308 @ 0.0}
C4 = _5858{0.0 .. 1.79769313486232e+308 @ 100.0}
C5 = _5842{0.0 .. 1.79769313486232e+308 @ 200.0}00  ton kopalin.

Laczne minimalne koszty wydobycia i transportu:
223100.000
```

5.7.6 Plan przewozu kopalin 4

Komunikat generowany przez program `5_20_kopalnie_3.ecl` jest słabo czytelny. Czytelność można znacznie poprawić wprowadzając modyfikacje pokazane w programie `5_21_kopalnie_4.ecl`:

```
/*1*/ :- lib(eplex).
 /*2*/ top :-
/*3*/       Zmienne = [A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,C1,C2,C3,C4,C5],
/*4*/       Zmienne $:: 0.0..1.0Inf,
/*5*/        integers(Zmienne),
   % Wydobycie kopalin w kopalni K1:
/*6*/        A1+A2+A3+A4+A5 $=600,
   % Wydobycie kopalin w kopalni K2:
/*7*/        B1+B2+B3+B4+B5 $=700,
   % Wydobycie kopalin w kopalni K3:
/*8*/        C1+C2+C3+C4+C5 $=700,
   % Pojemność składów:
/*9*/         A1+B1+C1 $=400,
/*10*/        A2+B2+C2 $=400,
/*11*/        A3+B3+C3 $=400,
/*12*/        A4+B4+C4 $=400,
/*13*/        A5+B5+C5 $=400,

/*14*/       Koszt $= 14*A1+5*A2+9*A3+24*A4+15*A5+
      30*B1+24*B2+11*B3+8*B4+19*B5+9*C1+22*C2+15*C3+7*C4+18*C5+
      108*A1+108*A2+108*A3+108*A4+108*A5+96*B1+96*B2+96*B3+96*B4+
      96*B5+102*C1+102*C2+102*C3+102*C4+102*C5,

/*15*/       eplex_solver_setup(min(Koszt)),
/*16*/       eplex_solve(Koszt),
/*17*/       eplex_get(vars,Vars),
/*18*/       eplex_get(typed_solution,Vals),
/*19*/       Vars = Vals,nl,

/*20*/ write("Kopalnia K1 wywiozła do:"),nl,
/*21*/       write("skladu s1 = "),write(A1),write(" ton kopalin."),nl,
/*22*/       write("skladu s2 = "),write(A2),write(" ton kopalin."),nl,
/*23*/       write("skladu s3 = "),write(A3),write(" ton kopalin."),nl,
/*24*/       write("skladu s4 = "),write(A4),write(" ton kopalin."),nl,
/*25*/       write("skladu s5 = "),write(A5),write(" ton kopalin."),nl,nl,
/*26*/       write("Kopalnia K2 wywiozła do:"),nl,
/*27*/       write("skladu s1 = "),write(B1),write(" ton kopalin."),nl,
/*28*/       write("skladu s2 = "),write(B2),write(" ton kopalin."),nl,
/*29*/       write("skladu s3 = "),write(B3),write(" ton kopalin."),nl,
/*30*/       write("skladu s4 = "),write(B4),write(" ton kopalin."),nl,
/*31*/       write("skladu s5 = "),write(B5),write(" ton kopalin."),nl,nl,
/*32*/       write("Kopalnia K3 wywiozła do:"),nl,
/*33*/       write("skladu s1 = "),write(C1),write(" ton kopalin."),nl,
/*34*/       write("skladu s2 = "),write(C2),write(" ton kopalin."),nl,
```

```
/*35*/       write("skladu s3 = "),write(C3),write(" ton kopalin."),nl,
/*36*/       write("skladu s4 = "),write(C4),write(" ton kopalin."),nl,
/*37*/       write("skladu s5 = "),write(C5),write(" ton kopalin."),nl,nl,
/*38*/       write("Laczne minimalne koszty wydobycia i transportu: "),
                          write(Koszt).
```

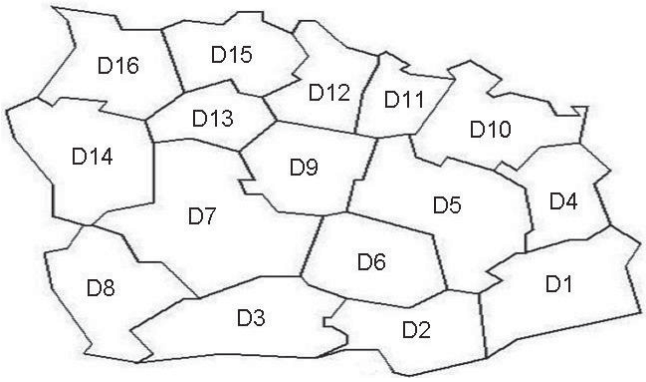
Program generuje komunikat:

```
Kopalnia K1 wywiozla do:
  skladu s1 = 0 ton kopalin.
  skladu s2 = 400 ton kopalin.
  skladu s3 = 0 ton kopalin.
  skladu s4 = 0 ton kopalin.
  skladu s5 = 200 ton kopalin.

Kopalnia K2 wywiozla do:
  skladu s1 = 0 ton kopalin.
  skladu s2 = 0 ton kopalin.
  skladu s3 = 400 ton kopalin.
  skladu s4 = 300 ton kopalin.
  skladu s5 = 0 ton kopalin.

Kopalnia K3 wywiozla do:
  skladu s1 = 400 ton kopalin.
  skladu s2 = 0 ton kopalin.
  skladu s3 = 0 ton kopalin.
  skladu s4 = 100 ton kopalin.
  skladu s5 = 200 ton kopalin.

Laczne minimalne koszty wydobycia i transportu: 223100.0
```

5.7.7 Kolorowanie map

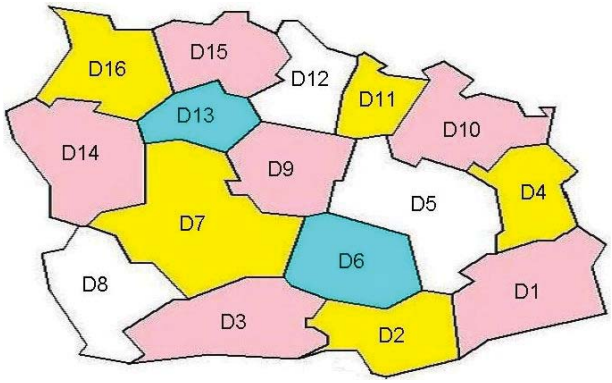
Spróbujmy zweryfikować znane twierdzenie o kolorowaniu krawędzi grafu płaskiego (patrz rozdział 2.4.7 i rozdział 3.7.4). Zrobimy to na przykładzie kolorowania mapy administracyjnej Absurdolandii, pokazanej na rysunku 5.10. Zadanie to jest również przykładem zadania przyporządkowania.

Należy wykonać kolorowanie tej mapy tak, by sąsiadujące ze sobą dzielnice (oznaczone symbolami D1, D2,...,D16) nie miały jednakowych kolorów, przy czym całkowita liczba użytych kolorów powinna być minimalna. Odpowiedni program (`5_22_mapa.ecl`) jest następujący:



Rysunek 5.10: Mapa administracyjna Absurdolandii

```
/*1*/   :- lib(ic).
/*2*/   :- lib(ic_edge_finder3).
 /*3*/   :- lib(branch_and_bound).

/*4*/   top:-
/*5*/ [D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16] :: 1..16,
%  Di - numer koloru przyporządkowanego dzielnicy i-tej
/*6*/ kolory([D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16]),
/*7*/ maxlist([D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16],M),
/*8*/ minimize(
        labeling([D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16]),M),nl,
/*9*/ write("Minimalna liczba kolorow = "),write(M),nl,
/*10*/ write("Dzielnice = "),nl,
        write("D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16"),
/*11*/ write("Kolory = ")
,         write([D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16]).

/*12*/ kolory([D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16]):-
/*13*/ D1 #\= D4, /*14*/ D1 #\= D5, /*15*/ D1 #\= D2, /*16*/ D2 #\= D5,
/*17*/ D2 #\= D6, /*18*/ D2 #\= D3, /*19*/ D3 #\= D6, /*20*/ D3 #\= D7,
/*21*/ D3 #\= D8, /*22*/ D4 #\= D10, /*23*/ D4 #\= D5, /*24*/ D5 #\= D10,
/*25*/ D5 #\= D11, /*26*/ D5 #\= D9, /*27*/ D5 #\= D6, /*28*/ D6 #\= D9,
/*29*/ D6 #\= D7, /*30*/ D7 #\= D9, /*31*/ D7 #\= D13, /*32*/ D7 #\= D14,
/*33*/ D7 #\= D8, /*34*/ D8 #\= D14, /*35*/ D9 #\= D11, /*36*/ D9 #\= D12,
/*37*/ D9 #\= D13, /*38*/ D10 #\= D11, /*39*/ D11 #\= D12, /*40*/ D12 #\= D15,
```



Rysunek 5.11: Wynik kolorowania mapy administracyjnej Absurdolandii

```
/*41*/ D12 #\= D13, /*42*/ D13 #\= D15, /*43*/ D13 #\= D16, /*44*/ D13 #\= D14,
/*45*/ D14 #\= D16, /*46*/ D15 #\= D16, /*47*/ D14 #\= D8.
```

Rozwiązaniem jest:

```
Minimalna liczba kolorow = 4
Dzielnice = D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16
Kolory =    [1, 2, 1, 2, 3, 4, 2, 3, 1, 1,  2,  3,  4,  1,  1,   2]
```

Rozwiązanie to jest słabo czytelne. Oznacza ono np. że dzielnica `D1` (której na liście kolorów odpowiada pierwsza jedynka), ma mieć inny kolor niż dzielnica `D2` (której na liście kolorów odpowiada pierwsza dwójka). Jeżeli natomiast przyjmiemy (zupełnie arbitralnie), że:

```
 1 = kolor różowy      2 = kolor żółty
 3 = kolor biały       4 = kolor niebiesko-zielony,
```

to możemy mapę administracyjną Absurdolandii pokolorować jak pokazano na rysunku 5.11.

5.7.8 Walczymy o równomierne udeszczowienie

Problem reprezentacji zbiorów może również polegać na wyznaczeniu takiego zbioru zawierającego elementy innych zbiorów, który spełnia poza tym kryterium minimalnego kosztu zawartych w nim elementów. Bardzo często zagadnie-

nie to sprowadza się do znanej z badań operacyjnych postaci kanonicznej, bez konieczności dokonywania skomplikowanych transformacji. Ilustruje to następujący przykład:

Światowa Organizacja Sprawiedliwości Totalnej za jedną ze swych najważniejszych misji uznała walkę z nierównomiernymi opadami deszczowymi na kuli ziemskiej. Nierównomierność ta - zdaniem Organizacji - jest niesprawiedliwością wołającą o pomstę do *Matki Ziemi*, gdyż właśnie w niej można upatrywać praprzyczynę wszelkich innych niesprawiedliwości. Sukcesy 15-letniej działalności Organizacji na tym polu były jednak bardzo umiarkowane. Przyczynę tego słusznie upatrywano w niedostatecznym finansowaniu, które pozwalało na stosowanie tylko bardzo prymitywnych form wyrównywania tej niesprawiedliwości, np. rurociągów transportujących deszczówkę z rejonów jej obfitości w rejony jej niedostatku. Dopiero wprowadzenie przez Rząd Światowy obowiązkowego i powszechnego Podatku Deszczowego (płaconego zarówno przez kraje obfitujące w opady jak i kraje pozbawione opadów) pozwoliło wreszcie na gruntowną reorganizację form walki o równomierne udeszczowienie. W szczególności uznano za niezbędne powołanie następujących *Agend Deszczowych*: 1)*Światowego Banku Deszczowego*, zbierającego podatki i finansującego wydatki, 2)*Floty Deszczowej*, składającej się z samolotów nadeszczających i oddeszczających, 3)*Satelitarnego Monitoringu Deszczowego*, 4)*Lokalnych Pól Elektrycznych* jonizujących powietrze dla wzmożenia opadów, 5)*Agendy Edukacji Deszczowej*, koordynującej deszczową edukację, od zajęć w szkołach poczynając (przedmiot "Sprawiedliwość deszczowa" w klasach młodszych, "Zarządzanie deszczem" w klasach starszych), a na sieci *Instytucji Korekcyjno-Edukacyjnych* (dla przekonywania i pozyskiwania szczególnie zatwardziałych oponentów sprawiedliwości deszczowej) kończąc, 6)*Agendy Lobbingu Deszczowego*, której celem jest zarówno dotarcie do przywódców państwowych z odpowiednimi argumentami, zachęcającymi ich do dobrowolnego opodatkowania się powyżej wyznaczonych limitów jak i wspomaganie *Grup Pokojowych Aktywistów Deszczowych*, 7)*Światowego Instytutu Deszczowego*, finansującego i koordynującego badania naukowe w zakresie deszczologii (przydzielającego m. in. *Granty Deszczowe dla Młodych Badaczy*) i organizującego coroczne *Naukowe Kongresy Deszczologiczne* na wybranych stadionach piłkarskich.

Tak szeroko zakrojona działalność wymaga oczywiście nowych, wysoko kwalifikowanych specjalistów. Szczęśliwym trafem, trzej wybitni międzynarodowi działacze deszczowi - profesorowie Hoaxman i Luftmensch oraz pułkownik baron Fraud of Cundelbury - mają progeniturę, która od lat dziecięcych, przysłuchując się dyskusjom przy rodzinnym stole, nabyła tak głębokiej wiedzy z obszaru

deszczologii teoretycznej i stosowanej, jakiej nie dostarczyłby im najlepszy uniwersytet. Równie szczęśliwym trafem, progenitura ta właśnie rozgląda się za nowymi posadami.

I tak dwie córeczki pułkownika barona musiały zrezygnować z posad wiceprezesek Banku Bzdurnych Inicjatyw. Jak się okazało, starsza nie do końca rozumiała, co to są procenty, czym naraziła bank na ogromne straty[1], a młodszej notorycznie myliły się tryliony europejskie z trylionami amerykańskimi i angielskimi, co stało się powodem ogromnych kłopotów banku[2].

Syn profesora Hoaxmana, po przedwcześnie przerwanych studiach wychowania fizycznego, pracował jako *caddy* w ekskluzywnym klubie golfowym. Przysłuchując się przez czas dłuższy rozmowom grających bankowców doszedł do słusznego wniosku, że świetnie sobie w tym fachu poradzi. Odtąd jego marzeniem było zrobić karierę w biznesie bankowym.

Ambicją młodego Luftmenscha był mundur, najlepiej taki elegancki, granatowy lub biały, ze złotymi szamerunkami i różnokolorowymi baretkami, no i oczywiście władza i uwielbienie dziewcząt, nieodłączne od takich mundurów. W mundurach tych mógłby skutecznie łączyć nadzór nad flotą deszczową (samolotami nadeszczającymi dowodziłby w mundurze granatowym, a oddeszczającymi - w mundurze białym) z nadzorem nad instytucjami korekcyjno-edukacyjnymi. Niestety, z powodu zbyt wysokiego stopnia niestabilności emocjonalnej, młody Luftmensch nie został przyjęty do Renomowanej Szkoły Wojskowej.

Teraz cała czwórka dostrzegła swe szanse. Za namową tatusiów wszyscy złożyli wnioski o zatrudnienie w charakterze organizatorów nowych agend deszczowych. Wnioski były bardzo lakoniczne (w końcu nazwiska mówią same za siebie) i zawierały wyłącznie nazwę agendy i oczekiwane wynagrodzenie (w miliardach JP na rok). Wszyscy wnioskodawcy, na fali alkoholowo-narkotykowego entuzjazmu, zadeklarowali godną uznania chęć organizowania kilku agend. Niestety, tatusiowie nie uzgodnili między sobą, kto do czego będzie startował; w wyniku agendy preferowane przez młodych częściowo pokrywały się, co przedstawiono w tablicy 5.4.

Iluminatowe szefostwo Organizacji nie uznało tego jednak za wadę, gdyż - jak każde szefostwo - uwielbiało rozstrzygać konflikty kompetencyjne swych podwładnych, a co tu rozstrzygać gdy konfliktów brak. Ze względu na opinię

[1] Było to całkowicie wybaczalne. Biedaczka nie była - z powodu ostrego zatrucia alkoholowo-narkotykowego - na tych lekcjach *Matematyki Domowej* w college'u, na których omawiano procenty.

[2] W tym przypadku okolicznością usprawiedliwiającą może być to, że - z powodu pilnej potrzeby pozbycia się owocu upojnej nocy spędzonej nie bardzo wiadomo z kim - nie była obecna na lekcjach *Matematyki Domowej* poświęconych liczbom dużym.

Kto / Agenda	Hoaxman Jr	Luftmensch Jr	Starsza Ms Cundelbury	Młodsza Ms Cundelbury
Bank Deszczowy	●			●
Flota samolotów		●	●	
Monitoring satelitarny			●	●
Pola jonizujące		●		●
Akcja edukacyjna		●	●	
Akcja lobbingowa	●			●
Instytut naukowy	●	●		
Zarobki (mld JP)	6	5	5	12

Tablica 5.4: Propozycje organizacji agend deszczowych

publiczną postanowiono jednak wybrać - pod hasłem konkursu - najtańsze propozycje obejmujące wszystkie agendy, posługując się w tym celu programem `5_23_deszcz.ecl`:

```
/*1*/  :-lib(ic).
/*2*/  :-lib(branch_and_bound).

/*3*/  top :-
 % Xj = 1 - wniosek j-tego wnioskodawcy jest uwzględniony.
 % Xj = 0 - wniosek j-tego wnioskodawcy jest odrzucony.
/*4*/  Zmienne=[X1,X2,X3,X4],
/*5*/  Zmienne :: 0..1,

/*6*/  X1 + X4 #>= 1,
/*7*/  X2 + X3 #>= 1,
/*8*/  X3 + X4 #>= 1,
/*9*/  X2 + X4 #>= 1,
/*10*/ X3 + X4 #>= 1,
/*11*/ X1 + X4 #>= 1,
/*12*/ X1 + X2 #>= 1,
```

```
/*13*/ Koszt #= 6*X1 + 5*X2 + 5*X3 + 12*X4,

/*14*/ minimize(search(Zmienne,0,first_fail,indomain,
         complete,[]),Koszt),
/*15*/ writeln("Zmienne":Zmienne ),
/*16*/ writeln("Koszt":Koszt).
```

Program generuje komunikat:

```
Found a solution with cost 17
Found a solution with cost 16
Found no solution with cost 0.0 .. 15.0
Zmienne : [1, 1, 1, 0]
Koszt : 16
```

Jak widać, wniosek zgłoszony przez Młodszą Ms Cundelbury nie został uwzględniony.

5.7.9 Send Most Money

Popularna łamigłówka *Send More Money* (patrz rozdział 4.4.1) ma również swą wersję optymalizacyjną o nazwie *Send Most Money*, której celem jest maksymalizacja wartości zmiennej *Money*. Odpowiedni program `5_24_smm.ecl`, zaczerpnięty z witryny [Kjellerstrand-13], jest postaci:

```
/*1*/  :-lib(ic).
/*2*/  :-lib(branch_and_bound).

/*3*/  top :-
   % 1) Wyznaczanie maksymalnej wartości zmiennej MONEY:
   % a)Tworzymy listę LD ośmiu zmiennych odpowiadających
   %   ośmiu literom słów "Send Most Money":
/*4*/    length(LD, 8),
   % b)Deklarujemy dziedzinę zmiennych z listy LD:
/*5*/    LD :: 0..9,
   % c)Formułujemy podstawowe ograniczenie:
/*6*/    send_most_money(LD, MONEY),
   % Chcemy maksymalizować MONEY, ale minimize/2 służy tylko
   % do minimalizacji. Dlatego będziemy minimalizować "ujemne"
   % MONEY:
/*7*/    MONEY_UJEMNE #= -MONEY,

/*8*/    writeln("Wyznaczanie maksymalnej wartości zmiennej
                 MONEY:"),
```

```
/*9*/    minimize(search(LD,0,first_fail,indomain,complete, []),
                 MONEY_UJEMNE),
/*10*/   writeln([MONEY, LD]),

   % 2) Wyznaczanie wszystkich rozwiązań dla maksymalnej
   %    wartości MONEY:
/*11*/   length(LD2, 8),
/*12*/   LD2 :: 0..9,
/*13*/   findall(LD2, (send_most_money(LD2, MONEY),
               labeling(LD2)
                        ),
/*14*/         Wszystko),
/*15*/   length(Wszystko, Dlugosc),
/*16*/   printf("%d rozwiazania dla maksymalnej wartosci
               MONEY = %d:\n", [Dlugosc, MONEY]),
/*17*/   writeln(Wszystko).

/*18*/ send_most_money([S,E,N,D,M,O,T,Y], MONEY) :-
/*19*/   MONEY #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y,
/*20*/   alldifferent([S,E,N,D,M,O,T,Y]),
/*21*/   M #\= 0,
/*22*/   S #\= 0,
/*23*/   1000 * S + 100 * E + 10 * N + D +
               1000 * M + 100 * O + 10 * S + T #=  MONEY.
```

Program generuje komunikat:

```
Wyznaczanie maksymalnej wartosci zmiennej MONEY:
Found a solution with cost -10437
Found a solution with cost -10438
Found a solution with cost -10548
Found a solution with cost -10657
Found a solution with cost -10765
Found a solution with cost -10768
Found a solution with cost -10875
Found a solution with cost -10876
Found no solution with cost -10878.0 .. -10877.0
[10876, [9, 7, 8, 2, 1, 0, 4, 6]]
2 rozwiazania dla maksymalnej wartosci MONEY = 10876:
[[9, 7, 8, 2, 1, 0, 4, 6],
 [9, 7, 8, 4, 1, 0, 2, 6]]
```

Przykład zasługuje na dodatkową uwagę, gdyż zastosowano w nim odmienny od dotychczas przedstawianych sposób wyznaczania wielokrotnych rozwiązań optymalnych. Użyto w tym celu predykatu `findall/3` (wiersz `/*13*/`). Sposób ten może być również stosowany dla innych zadań z wielokrotnymi rozwiązaniami optymalnymi.

5.8 Trudne optymalne przyporządkowania

5.8.1 Lokalizacja hurtowni - podejście BO

Podstawowy klasyczny problem lokalizacji hurtowni można sformułować następująco: *dla pewnej liczby klientów i pewnego zbioru możliwych lokalizacji hurtowni, wyznaczyć lokalizacje hurtowni tak by uzyskać minimum kosztów budowy hurtowni i zaopatrzenia klientów.* Problemy te są trudnymi problemami badań operacyjnych ze względu na konieczność stosowania zmiennych całkowitych i zmiennych binarnych. Różnorodne problemy lokalizacji hurtowni ciągle są wyzwaniami dla badaczy. Jednym z pierwszych problemów tego typu, rozwiązanych metodami *CLP*, był problem hurtowni przedstawiony w pionierskiej książce [van Hentenryck-89]. Ze względu na znaczenie tego problemu dla inwestorów, szereg jego odmian, o różnym stopniu trudności, zostało analizowanych i rozwiązanych w ramach badań operacyjnych.

Rozpocznijmy rozważania na temat lokalizacji hurtowni od prostego przykładu: istnieją trzy potencjalne lokalizacje hurtowni, które będą obsługiwać pięciu klientów. Koszty budowy tych hurtowni i koszty zaopatrywania klientów przedstawia tablica 5.5.

Klient	Hurtownia		
	1	2	3
1	5	7	20
2	4	20	1
3	20	2	5
4	20	20	4
5	3	20	8
Koszt budowy	18	20	28

Tablica 5.5: Koszt zaopatrzenia i koszt budowy 3 hurtowni dla 5 klientów

Rozwiązanie przedstawia program `5_25_hurtownie_BO.ecl`:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(branch_and_bound).

/*3*/ top:-
/*4*/     hurtownie(_,_).

/*5*/  hurtownie(Z,Koszt):-

  %  Wj = 1 - hurtownia j będzie zbudowana
  %  Wj = 0 - hurtownia j nie będzie zbudowana
  %  Tij = 1 - klient i jest obsługiwany przez hurtownię j
  %  Tij = 0 - klient i nie jest obsługiwany przez hurtownię j

/*6*/  Z=[W1,W2,W3,T11,T12,T13,T21,T22,T23,T31,T32,T33,T41,T42,T43,T51,T52,T53],
/*7*/    Z :: 0..1,

/*8*/  T11 + T12 + T13 #= 1,  % klient 1 jest obsługiwany przez jedną hurtownię
/*9*/  T21 + T22 + T23 #= 1,  % klient 2 jest obsługiwany przez jedną hurtownię
/*10*/ T31 + T32 + T33 #= 1,  % klient 3 jest obsługiwany przez jedną hurtownię
/*11*/ T41 + T42 + T43 #= 1,  % klient 4 jest obsługiwany przez jedną hurtownię
/*12*/ T51 + T52 + T53 #= 1,  % klient 5 jest obsługiwany przez jedną hurtownię

/*13*/ T11 #=< W1, % hurtownia 1, jeżeli  zbudowana, może obsługiwać klienta 1
/*14*/ T21 #=< W1, % hurtownia 1, jeżeli  zbudowana, może obsługiwać klienta 2
/*15*/ T31 #=< W1, % hurtownia 1, jeżeli  zbudowana, może obsługiwać klienta 3
/*16*/ T41 #=< W1, % hurtownia 1, jeżeli  zbudowana, może obsługiwać klienta 4
/*17*/ T51 #=< W1, % hurtownia 1, jeżeli  zbudowana, może obsługiwać klienta 5

/*18*/ T12 #=< W2, % hurtownia 2, jeżeli  zbudowana, może obsługiwać klienta 1
/*19*/ T22 #=< W2, % hurtownia 2, jeżeli  zbudowana, może obsługiwać klienta 2
/*20*/ T32 #=< W2, % hurtownia 2, jeżeli  zbudowana, może obsługiwać klienta 3
/*21*/ T42 #=< W2, % hurtownia 2, jeżeli  zbudowana, może obsługiwać klienta 4
/*22*/ T52 #=< W2, % hurtownia 2, jeżeli  zbudowana, może obsługiwać klienta 5

/*23*/ T13 #=< W3, % hurtownia 3, jeżeli  zbudowana, może obsługiwać klienta 1
/*24*/ T23 #=< W3, % hurtownia 3, jeżeli  zbudowana, może obsługiwać klienta 2
/*25*/ T33 #=< W3, % hurtownia 3, jeżeli  zbudowana, może obsługiwać klienta 3
/*26*/ T43 #=< W3, % hurtownia 3, jeżeli  zbudowana, może obsługiwać klienta 4
/*27*/ T53 #=< W3, % hurtownia 3, jeżeli  zbudowana, może obsługiwać klienta 5

/*28*/     Koszt #= 18*W1+10*W2+28*W3+5*T11+7*T12+100*T13+4*T21+100*T22+1*T23+
               100*T31+2*T32+5*T33+100*T41+100*T42 +4*T43+3*T51+100*T52+8*T53,

/*29*/     bb_min(labeling([W1,W2,W3,T11,T12,T13,T21,T22,T23,T31,T32,T33,T41,
               T42,T43,T51,T52,T53]),Koszt, bb_options{strategy:restart}),nl,

/*30*/     write("Lista hurtowni: "),writeln([W1,W2,W3]),nl,
```

```
/*31*/     write("Lista klientow i hurtowni:"),nl,
/*32*/     writeln("[T11,T12,T13,T21,T22,T23,T31,T32,T33,T41,T42,T43,
                T51,T52,T53]"),
/*32*/     writeln([T11,T12,T13,T21,T22,T23,T31,T32,T33,T41,T42,T43,
                T51,T52,T53]),
/*33*/     write("Koszt: "),writeln(Koszt),nl.
```

Komunikat ma postać:

```
Found a solution with cost 66
Found a solution with cost 64
Found no solution with cost 0.0 .. 63.0
Lista hurtowni: [1, 0, 1]
Lista klientow i hurtowni:
[T11,T12,T13,T21,T22,T23,T31,T32,T33,T41,T42,T43,T51,T52,T53]
[1,  0,  0,  0,  0,  1,  0,  0,  1,  0,  0,  1,  1,  0,  0]
Koszt: 64
```

Znaczenie listy klientów i hurtowni jest następujące:

```
T11 = 1, tzn. klient 1 jest obsługiwany przez hurtownię 1;
T23 = 1, tzn. klient 2 jest obsługiwany przez hurtownię 3;
T33 = 1, tzn. klient 3 jest obsługiwany przez hurtownię 3;
T43 = 1, tzn. klient 4 jest obsługiwany przez hurtownię 3;
T51 = 1, tzn. klient 5 jest obsługiwany przez hurtownię 1;
```

5.8.2 Lokalizacja hurtowni - podejście CLP - 1

Zagadnienie lokalizacji przedstawione tablicą 5.5 można rozwiązać stosując podejście *CLP* na szereg różnych sposobów. Jeden z nich przedstawia program `5_26_hurtownie_CLP_1.ecl`, stosujący technikę z rozdziału 2.1.4:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(branch_and_bound).

   %op(+Precedens, +Asocjatywność, ++Nazwa)
/*3*/  :- op(960, fx, if).
/*4*/  :- op(950,xfx, then).

/*5*/  top:-
/*6*/          hurtownie(_,_,_).
```

```
/*7*/  hurtownie(Hs,Ks,Koszt):-
/*8*/          Hs=[H1,H2,H3],
      % jeżeli Hi=0, hurtownia "i" nie jest zakładana
      % jeżeli Hi=1, hurtownia "i" jest zakładana
/*9*/          Hs::0..1,

/*10*/         Ks=[K1,K2,K3,K4,K5],
/*11*/         Ks::1..3,
      %   Ki - numer hurtowni obsługującej i-tego klienta

/*12*/         element(K1,[5,7,20],Koszt_1),
/*13*/         element(K2,[4,20,1],Koszt_2),
/*14*/         element(K3,[20,2,5],Koszt_3),
/*15*/         element(K4,[20,20,4],Koszt_4),
/*16*/         element(K5,[3, 20,8],Koszt_5),

         % jeżeli hurtownia "i" nie jest zakładana, nie pojawi się w liście Ks:
/*17*/         if (H1 #= 0) then z_posrod(Ks,1),
/*18*/         if (H2 #= 0) then z_posrod(Ks,2),
/*19*/         if (H3 #= 0) then z_posrod(Ks,3),

/*20*/         Koszt #= 18*H1+20*H2+28*H3+Koszt_1+Koszt_2+Koszt_3+Koszt_4+Koszt_5,

/*21*/         bb_min((labeling(Hs),labeling(Ks)),Koszt,
                       bb_options{strategy:restart}),

/*22*/         write("   Lista hurtowni: "),
                  writeln([H1,H2,H3]),
/*23*/         write("   Lista klientow i hurtowni: "),
                  writeln([K1,K2,K3,K4,K5]),
/*24*/         write("   Koszt: "),
                  writeln(Koszt).

/*25*/ z_posrod([],_).
/*26*/ z_posrod([],_):-
/*27*/         K #\= N,
/*28*/         z_posrod(Ks,N).

/*29*/ if Warunek then Wniosek :-
/*30*/         Warunek =.. KLista,
/*31*/         append(KLista, [Bool], RLista),
/*32*/         Term =.. RLista,
/*33*/         call(Term),
/*34*/         wywolaj_jesli(Wniosek, Bool).

/*35*/ delay wywolaj_jesli(_Wniosek, Bool) if var(Bool).
/*36*/ wywolaj_jesli(_Wniosek, 0).
```

```
/*37*/ wywolaj_jesli(Wniosek, 1) :-
/*38*/         call(Wniosek).
```

Program generuje komunikat:

```
Found a solution with cost 66
Found a solution with cost 64
Found no solution with cost 15.0 .. 63.0
Lista hurtowni: [1, 0, 1]
Lista klientow i hurtowni: [1, 3, 3, 3, 1]
Koszt: 64
```

Znaczenie listy hurtowni (`[1, 0, 1]`) jest następujące: tylko hurtownie `1` i `3` zostaną zbudowane.
Znaczenie listy hurtowni i klientów (`[1, 3, 3, 3, 1]`) jest następujące: klient `1` będzie obsługiwany przez hurtownię `1`, klienci `2`, `3` i `4` przez hurtownię `3`, klient `5` - przez hurtownię `1`.

W programie zastosowano szereg nowych predykatów standardowych, które wymagają komentarza:

- predykat `?Term =.. ?Lista` jest spełniony, jeżeli `Lista` jest listą, której pierwszym elementem jest nazwa predykatu `Term`, a pozostałymi elementami są argumenty tego predykatu, jeżeli w ogóle je ma. Np.:

  ```
  Term =.. [likes,"John",play].
  ```

  daje

  ```
  Term = likes("John",play),
  ```

  natomiast:

  ```
  s([1,4,5,6]) =.. List.
  ```

  daje

  ```
  List = [s,[1,4,5,6]].
  ```

- predykat standardowy `call(+Cel)` jest spełniony, gdy `Cel` jest spełniony. Predykat ten wywołuje `Cel`, który musi być spełniony tylko w chwili jego wywoływania. W rozpatrywanym przypadku oznacza to dla linii `35,..,38` konieczność czekania (`delay`) aż `Bool` jest spełniony, po czym następuje wywołanie `Cel`.

- `op(960, fx, if)` i `op(950,xfx, then)` oznaczają, że część "`then`" z linii `17`, `18` i `19` jest wywołana dopiero po wywołaniu części "`if`" , zob. rozdział 2.1.4.

Tak jak poprzednio, liczba zmiennych potrzebnych dla modelowania problemu przy podejściu *BO* jest większa aniżeli liczba zmiennych potrzebna w przypadku podejścia *CLP*.

5.8.3 Lokalizacja hurtowni - podejście CLP - 2

Program `5_26_hurtownie_CLP_1.ecl` rozpatrywany poprzednio miał słabe właściwości propagacyjne[3], za co odpowiada wielokrotne stosowanie predykatu standardowego `element/3`. Następny program `5_27_hurtownie_CLP_2.ecl`, będący nieznaczna modyfikacja programu *Warehouse location* autorstwa J. Schimpfa (zob. [Schimpf-10]), jest znacznie lepszy. Korzysta on z heurystyki porządkującej - dla każdego klienta - hurtownie w kolejności wzrastającego kosztu zaopatrzenia. Działanie tego programu zostanie zilustrowane bardziej złożonym problem lokalizacji hurtowni, przedstawionym w tablicy 5.6.

Klienci	Hurtownie			
	1	2	3	4
1	5	7	1	20
2	14	8	100	300
3	2	20	50	12
4	110	2	200	5
5	300	300	8	200
6	3	100	8	5
7	30	40	20	80
8	230	50	70	8
9	20	350	70	98
10	30	450	370	250
Koszt budowy	18	10	28	20

Tablica 5.6: Koszt zaopatrzenia i koszt budowy 4 hurtowni dla 10 klientów

Program `5_27_hurtownie_CLP_2.ecl` ma postać:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_sets).
 /*3*/  :- lib(branch_and_bound).
```

[3]Praktycznie oznacza to powolne działanie w przypadku większej liczby zmiennych.

```
/*4*/  top:-
      % przedstawienie danych:
/*5*/     tablica_kosztu_budowy(TablicaKosztuBudowy),
/*6*/     tablica_kosztu_dostawy(TablicaKosztuDostawy),
/*7*/     dim(TablicaKosztuDostawy,[LiczbaKlientow,LiczbaHurtowni]),
/*8*/     dim(TablicaKosztuBudowy,[LiczbaHurtowni]),

      % modelowanie ograniczeń:
/*9*/     intset(ListaBudowanychHurtowni,1,LiczbaHurtowni),
/*10*/    (
/*11*/    for(NumerKlienta, 1, LiczbaKlientow),
/*12*/    foreach(NumerHurtowniDlaKLienta,
                    HurtowniePrzydzieloneKlientom),
/*13*/    foreach(KosztDostawyDlaKlienta,
                    ListaKosztowDostawyDlaKlientow),
/*14*/    param(ListaBudowanychHurtowni,
                    TablicaKosztuDostawy,LiczbaHurtowni)
/*15*/    do
/*16*/    WektorKosztowDostawy is
            TablicaKosztuDostawy[NumerKlienta,1..LiczbaHurtowni],
/*17*/    element(NumerHurtowniDlaKLienta,WektorKosztowDostawy,
            KosztDostawyDlaKlienta),
/*18*/    NumerHurtowniDlaKLienta in ListaBudowanychHurtowni
/*19*/    ),
/*20*/    weight(ListaBudowanychHurtowni,TablicaKosztuBudowy,
            KosztBudowy),

          % wskaźnik jakości
/*21*/    KosztCalkowity #= KosztBudowy +
                  sum(ListaKosztowDostawyDlaKlientow),

      % poszukiwania i propagacja:
/*22*/    sortuj_hurtownie(TablicaKosztuDostawy,
                  PosortowaneListyHurtowniDlaKlientow),
/*23*/    minimize(
/*24*/    (
/*25*/    insetdomain(ListaBudowanychHurtowni, increasing, _, _),
/*26*/    labeling(HurtowniePrzydzieloneKlientom,
                  PosortowaneListyHurtowniDlaKlientow),

      % wyświetlanie wyników pośrednich
/*27*/    write("Lista budowanych hurtowni = "),
                  writeln(ListaBudowanychHurtowni),
/*28*/    write("Hurtownie przydzielone klientom = "),
                  writeln(HurtowniePrzydzieloneKlientom),
/*29*/    write("Lista kosztow dostawy dla klientow = "),
                  writeln(ListaKosztowDostawyDlaKlientow)

/*30*/    ),
```

```
/*31*/    KosztCalkowity),nl,

      % wyświetlanie wyników końcowych
/*32*/    write("Lista budowanych hurtowni = "),
                writeln(ListaBudowanychHurtowni),
/*33*/    write("Hurtownie przydzielone klientom = "),
                writeln(HurtowniePrzydzieloneKlientom),
/*34*/    write("Posortowane listy hurtowni dla klientow = "),
                writeln(PosortowaneListyHurtowniDlaKlientow),
/*35*/    write("Lista kosztow dostawy dla klientow = "),
                writeln(ListaKosztowDostawyDlaKlientow),
/*36*/    write("Koszt calkowity: "),writeln(KosztCalkowity).

      % heurystyka: dla każdego klienta sortuj hurtownie
      % zgodnie z rosnącym kosztem dostawy
/*37*/  sortuj_hurtownie(TablicaKosztuDostawy,
                    PosortowaneListyHurtowniDlaKlientow) :-
/*38*/    dim(TablicaKosztuDostawy,[LiczbaKlientow,LiczbaHurtowni]),
/*39*/    ( for(I,1,LiczbaHurtowni),
/*40*/    foreach(I,ListaIdentyfikatorowHurtowni)
/*41*/    do
/*42*/    true
/*43*/    ),
/*44*/    (
/*45*/    for(NumerKlienta, 1, LiczbaKlientow),
/*46*/    foreach(PosortowanaListaHurtowniDlaKlienta,
                    PosortowaneListyHurtowniDlaKlientow),
/*47*/    param(TablicaKosztuDostawy,LiczbaHurtowni,
                    ListaIdentyfikatorowHurtowni)
/*48*/    do
/*49*/    KosztyDostaw is TablicaKosztuDostawy[NumerKlienta,
                    1..LiczbaHurtowni],
/*50*/    sortowanie(KosztyDostaw, ListaIdentyfikatorowHurtowni,
                    PosortowanaListaHurtowniDlaKlienta)
/*51*/    ).

      % uziemiaj zmienne "HurtowniePrzydzieloneKlientom"
      % zgodnie z heurystyką
/*52*/  labeling(HurtowniePrzydzieloneKlientom,
                    PosortowaneListyHurtowniDlaKlientow) :-
/*53*/    (
/*54*/    foreach(NumerHurtowniDlaKLienta,
                    HurtowniePrzydzieloneKlientom),
/*55*/    foreach(PosortowanaListaHurtowniDlaKlienta,
                    PosortowaneListyHurtowniDlaKlientow)
/*56*/    do
/*57*/    member(NumerHurtowniDlaKLienta,
                    PosortowanaListaHurtowniDlaKlienta)
/*58*/    ).
```

```
     % ograniczenie pomocnicze: heurystyka sortowania
/*59*/  sortowanie(Klucze, Wartosci, PosortowaneWartosci) :-
/*60*/    (foreach(K,Klucze),
/*61*/    foreach(W,Wartosci),
/*62*/    foreach(K-W,WartosciKluczy)
/*63*/    do
/*64*/    true),
/*65*/    keysort(WartosciKluczy, PosortowaneWartosciKluczy),
/*66*/    (foreach(W,PosortowaneWartosci),
/*67*/    foreach(_K-W,PosortowaneWartosciKluczy)
/*68*/    do
/*69*/    true).

/*70*/    tablica_kosztu_dostawy([](
/*71*/    [](5,7,1,20),
/*72*/    [](14,8,100,300),
/*73*/    [](2,20,50,12),
/*74*/    [](110,2,200,5),
/*75*/    [](300,300,8,200),
/*76*/    [](3,100,8,5),
/*77*/    [](30,40,20,80),
/*78*/    [](230,50,70,8),
/*79*/    [](20,350,70,98),
/*80*/    [](30,450,370,250)
/*81*/    )).

/*81*/    tablica_kosztu_budowy([](18,10,28,20)).
```

Przebieg wyznaczania rozwiązania jest następujący:

```
Lista budowanych hurtowni = [1]
Hurtownie przydzielone klientom = [1,1,1,1,1,1,1,1,1,1]
Lista kosztow dostawy dla klientow = [5,14,2,110,300,3,30,230,20,30]
Found a solution with cost 762
Lista budowanych hurtowni = [1, 2]
Hurtownie przydzielone klientom = [1,2,1,2,1,1,1,2,1,1]
Lista kosztow dostawy dla klientow = [5,8,2,2,300,3,30,50,20,30]
Found a solution with cost 478
Lista budowanych hurtowni = [1, 3]
Hurtownie przydzielone klientom = [3,1,1,1,3,1,3,3,1,1]
Lista kosztow dostawy dla klientow = [1,14,2,110,8,3,20,70,20,30]
Found a solution with cost 324
Lista budowanych hurtowni = [1,2,3]
Hurtownie przydzielone klientom = [3,2,1,2,3,1,3,2,1,1]
Lista kosztow dostawy dla klientow = [1,8,2,2,8,3,20,50,20,30]
Found a solution with cost 200
```

```
Lista budowanych hurtowni = [1,3,4]
Hurtownie przydzielone klientom = [3,1,1,4,3,1,3,4,1,1]
Lista kosztow dostawy dla klientow = [1,14,2,5,8,3,20,8,20,30]
Found a solution with cost 177
Found no solution with cost 102.0 .. 176.0

Lista budowanych hurtowni = [1, 3, 4]
Hurtownie przydzielone klientom = [3,1,1,4,3,1,3,4,1,1]
Posortowane listy hurtowni dla klientow =
    [[3,1,2,4],[2,1,3,4],[1,4,2,3],[2,4,1,3],[3,4,1,2],
    [1,4,3,2],[3,1,2,4],[4,2,3,1],[1,3,4,2],[1,4,3,2]]
Lista kosztow dostawy dla klientow = [1,14,2,5,8,3,20,8,20,30]
Koszt calkowity: 177
```

5.8.4 Lokalizacja hurtowni - podejście CLP - 3

Problem z rozdziału 5.8.3 można rozwiązać również nie korzystając ze zbiorów, lecz korzystając z predykatu `fromto/4`. Przedstawiono to w przykładzie `5_28_hurtownie_CLP_3.ecl`[4], w którym skorzystano z następujących zmiennych:

- `ListaHurtowniKlientow` - lista zmiennych określających, które hurtownie zostały przydzielona którym klientom, np. `ListaHurtowniKlientow = [3, 1, 1, 4, 3, 1, 3, 4, 1, 1]` oznacza, że klient `5` będzie obsługiwany przez hurtownię `3`.
- `ListaHurtowniZbudowanych` - lista zmiennych określających, które hurtownie zostaną zbudowane, np. `ListaHurtowniZbudowanych= [1,0,1, 1]` oznacza, że hurtownia `2` nie zostanie zbudowana.
- `KosztyDojazdu` - tablica jednowymiarowa, której elementami są listy kosztów dojazdu dla kolejnych klientów.
- `KosztyBudowy` - lista kosztów budowy hurtowni.
- `KosztCalkowity` - suma kosztów budowy hurtowni i kosztów dojazdu do hurtowni.

Program `5_28_hurtownie_CLP_3.ecl` ma postać:

[4]Program ten zaproponował p. dr inż. Łukasz Domagała.

```
/*1*/  :-lib(ic).
/*2*/  :-lib(ic_global).
 /*3*/  :-lib(branch_and_bound).

/*4*/  top:-
/*5*/     zdefiniuj_problem(KosztyDojazdu,KosztyBudowy),
/*6*/     ogranicz(ListaHurtowniKlientow,
                 ListaHurtowniZbudowanych,KosztyDojazdu,
                      KosztyBudowy,KosztCalkowity),
/*7*/     optymalizuj(ListaHurtowniKlientow,
                 KosztCalkowity),
/*8*/     wypisz_wynik(ListaHurtowniKlientow,
                 ListaHurtowniZbudowanych,KosztCalkowity).

/*9*/  zdefiniuj_problem(KosztyDojazdu,KosztyBudowy):-
/*10*/    KosztyDojazdu=[](
          /*        H1  H2  H3  H4  */
/*11*/    /* K1 */ [5  ,7  ,1  ,20 ],
/*12*/    /* K2 */ [14 ,8  ,100,300],
/*13*/    /* K3 */ [2  ,20 ,50 ,12 ],
/*14*/    /* K4 */ [110,2  ,200,5  ],
/*15*/    /* K5 */ [300,300,8  ,200],
/*16*/    /* K6 */ [3  ,100,8  ,5  ],
/*17*/    /* K7 */ [30 ,40 ,20 ,80],
/*18*/    /* K8 */ [230,50 ,70 ,8  ],
/*19*/    /* K9 */ [20 ,350,70 ,98 ],
/*20*/    /* K10*/ [30 ,450,370,250]
/*21*/    ),

/*22*/    KosztyBudowy=[18,10,28,20].

/*23*/ ogranicz(ListaHurtowniKlientow,
                     ListaHurtowniZbudowanych,KosztyDojazdu,
                         KosztyBudowy,KosztCalkowity):-
/*24*/    dim(KosztyDojazdu,[LiczbaKlientow]),
/*25*/    length(KosztyBudowy,MaksymalnaLiczbaHurtowni),
    % Na podstawie "LiczbyKlientow" tworzy się nieuziemioną
    % "ListęHurtowniKlientow":
/*26*/    length(ListaHurtowniKlientow,LiczbaKlientow),
    % której dziedzina obejmuje wszystkie możliwe hurtownie:
/*27*/    ListaHurtowniKlientow#::[1..MaksymalnaLiczbaHurtowni],

    % Koszty budowy hurtowni:
/*28*/    (foreach(KosztBudowyHurtowni,KosztyBudowy),
/*29*/    foreach(HurtowniaZbudowana,ListaHurtowniZbudowanych),
/*30*/    fromto(ListaKosztow,[
                 KosztBudowyHurtowniLub0|ListaKosztowOut],
                       ListaKosztowOut,ListaKosztowKlienta),
/*31*/    count(NrHurtowni,1,MaksymalnaLiczbaHurtowni),
```

```
/*32*/    param(ListaHurtowniKlientow)
/*33*/    do
     % Liczba klientów hurtowni o numerze NrHurtowni:
/*34*/    occurrences(NrHurtowni, ListaHurtowniKlientow,
                      LiczbaKlientowHurtowniNr),
/*35*/    #>(LiczbaKlientowHurtowniNr,0,HurtowniaZbudowana),
/*36*/    KosztBudowyHurtowniLub0#=
                HurtowniaZbudowana * KosztBudowyHurtowni
/*37*/    ),

    % Koszty dojazdu do hurtowni:
/*38*/    (foreach(HurtowniaKlienta,ListaHurtowniKlientow),
/*39*/    foreacharg(ListaKosztowDojazdu,KosztyDojazdu),
/*40*/    foreach(KosztKlienta,ListaKosztowKlienta)
/*41*/    do
/*42*/    element(HurtowniaKlienta,ListaKosztowDojazdu,KosztKlienta)
/*43*/    ),

    % Wyznaczenie kosztu całkowitego
/*44*/    sumlist(ListaKosztow, KosztCalkowity).

/*45*/ optymalizuj(ListaHurtowniKlientow,KosztCalkowity):-
/*46*/    BBOptions=bb_options{strategy:continue, from:0},
/*47*/    bb_min(labeling(ListaHurtowniKlientow),KosztCalkowity,
                BBOptions).

/*48*/ wypisz_wynik(ListaHurtowniKlientow,ListaHurtowniZbudowanych,
                KosztCalkowity):-
/*49*/    write("KosztCalkowity = "),write(KosztCalkowity),nl,
/*50*/    write("Hurtownie zbudowane: "),
                write(ListaHurtowniZbudowanych),nl,
/*51*/    write("Hurtownie przyporzadkowane klientom = "),
                write(ListaHurtowniKlientow),nl.
```

Rozwiązanie ma postać:

```
Found a solution with cost 762
Found a solution with cost 592
Found a solution with cost 560
Found a solution with cost 498
Found a solution with cost 328
Found a solution with cost 296
Found a solution with cost 286
Found a solution with cost 220
```

```
Found a solution with cost 198
Found a solution with cost 188
Found a solution with cost 181
Found a solution with cost 177
Found no solution with cost 102.0 .. 176.0
KosztCalkowity=177
Hurtownie zbudowane:[1,0,1,1]
Hurtownie przyporzadkowane klientom:[3,1,1,4,3,1,3,4,1,1]
```

5.9 Przykłady optymalnego rozkładowania

Rozkładowaniem (ang. *timetabling*) nazywamy przydział resursów różnym działaniom, osobom, przedziałom czasu w sposób zapewniający osiągnięcie określonego celu. Ściślej - rozkładowanie to tworzenie takich podzbiorów iloczynów kartezjańskich następujących zbiorów:

- zbiór resursów (np. sale wykładowe, stoliki w barze szybkiej obsługi, stanowiska kolektorów opłat autostradowych),
- zbiór osób (np. studentów, wykładowców, pracowników obsługi);
- zbiór czynności (np. gotowanie, serwowanie, wykładanie, słuchanie wykładu, zbieranie opłat, kontrolowanie);
- zbiór przedziałów czasowych (dni tygodnia, godziny lekcyjne od-do),

które spełniają ograniczenia związane z pewnym działaniem (np. prowadzenie zajęć ze studentami, leczenie pacjentów, serwowanie posiłków, zbieranie opłat za korzystanie z autostrady), przy równoczesnej minimalizacji pewnego wskaźnika jakości. W wyniku otrzymuje się *rozkład czynności* (np. rozkład zajęć, rozkład dyżurów) informujący kto, gdzie i co ma robić i za pomocą czego.

CLP znalazło bardzo wdzięczne pole zastosowań dla wszelkiego rodzaju zadań rozkładowania pracy załóg (ang. *crew timetabling problems*). Rozpatrzmy kilka z nich.

5.9.1 Bar szybkiej obsługi

Bar szybkiej obsługi ma następujące zapotrzebowanie na pracowników: poniedziałek – 20, wtorek – 16, środa - 13, czwartek – 16, piątek - 19, sobota –

14, niedziela - 12. Każdy pracownik pracuje przez 5 kolejnych dni i ma wolne 2 kolejne dni. Należy wyznaczyć minimalną liczbę pracowników, zaspakajających codzienne zapotrzebowanie na pracowników. Odpowiednim programem jest `5_29_zaloga.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(branch_and_bound).

/*3*/   top:-
       % Zapotrzebowanie na liczbę pracowników dla kolejnych dni
/*4*/       Zapotrzebowanie = [20,16,13,16,19,14,12],

       % Dziedzina zmiennych problemu:
       % Pn - liczba pracowników rozpoczynających pracę w
       % poniedziałek, itd.:
/*5*/       [Pn,Wt,Sr,Cz,Pi,So,Ni] :: 0..50,

       % W poniedziałek pracują pracownicy (oznaczeni zmienną
       % "Poniedzialek") rozpoczynający pracę w poniedziałek,
       % czwartek, piątek, sobotę i niedzielę, natomiast wolne
       % mają pracownicy rozpoczynający pracę we wtorek i środę.
/*6*/       Poniedzialek #= Pn + Cz + Pi + So + Ni,

       % Liczba pracowników pracujących w poniedziałek
       % nie może być mniejsza od zapotrzebowania:
/*7*/       element(1,Zapotrzebowanie,D1),
/*8*/       Poniedzialek #>= D1,

       % Analogicznie do powyższych ograniczeń definiuje
       % się ograniczenia dla następnych dni tygodnia

/*9*/       Wtorek #= Wt + Pi + So + Ni + Pn,
/*10*/      element(2,Zapotrzebowanie,D2),
/*11*/      Wtorek #>= D2,

/*12*/      Sroda #= Sr + So + Ni + Pn + Wt,
/*13*/      element(3,Zapotrzebowanie,D3),
/*14*/      Sroda #>= D3,

/*15*/      Czwartek #= Cz + Ni + Pn + Wt + Sr,
/*16*/      element(4,Zapotrzebowanie,D4),
/*17*/      Czwartek #>= D4,

/*18*/      Piatek #= Pi + Pn + Wt + Sr + Cz,
/*19*/      element(5,Zapotrzebowanie,D5),
/*20*/      Piatek #>= D5,
```

```
/*21*/      Sobota #= So + Wt + Sr + Cz + Pi,
/*22*/      element(6,Zapotrzebowanie,D6),
/*23*/      Sobota #>= D6,

/*24*/      Niedziela #= Ni + Sr + Cz + Pi + So,
/*25*/      element(7,Zapotrzebowanie,D7),
/*26*/      Niedziela #>= D7,

      % Za pomocą predykatu "bb_min(_)" minimalizuje się liczbę
      % pracowników potrzebnych do zrealizowania tygodniowego
      % zapotrzebowania. Predykat ten wywołuje
      % predykat 'labeling/1' uzgadniający wartości zmiennych
      % Pn,Wt,Sr,Cz,Pi,So,Ni z ich dziedzinami:

/*27*/      Liczba_pracownikow #= Pn+Wt+Sr+Cz+Pi+So+Ni,
/*28*/      bb_min(labeling([Pn,Wt,Sr,Cz,Pi,So,Ni]),
              Liczba_pracownikow,bb_options with [strategy:step]),

      write("W poniedzialek podejmuje prace "),write(Pn),
            write(" pracownikow."),nl,
      write("We wtorek podejmuje prace "),write(Wt),
            write(" pracownikow."),nl,
      write("W srode podejmuje prace "),write(Sr),
            write(" pracownikow."),nl,
      write("W czwartek podejmuje prace "),write(Cz),
            write(" pracownikow."),nl,
      write("W piatek podejmuje prace "),write(Pi),
            write(" pracownikow."),nl,
      write("W sobote podejmuje prace "),write(So),
            write(" pracownikow."),nl,
      write("W niedziele podejmuje prace "),write(Ni),
            write(" pracownikow."),nl,
      write("Trzeba zatrudnic "),write(Liczba_pracownikow),
            write(" pracownikow.").
```

Program generuje następujące rozwiązanie:

```
Found a solution with cost 35
Found a solution with cost 34
...
Found a solution with cost 22
Found no solution with cost 0.0 ..21.0
```

```
W poniedzialek podejmuje prace 8 pracownikow.
We wtorek podejmuje prace 2 pracownikow.
W srode podejmuje prace 0 pracownikow.
W czwartek podejmuje prace 6 pracownikow.
W piatek podejmuje prace 3 pracownikow.
W sobote podejmuje prace 3 pracownikow.
W niedziele podejmuje prace 0 pracownikow.
Trzeba zatrudnic 22 pracownikow.
```

Rozwiązanie to przedstawia rozkład pracy z rysunku 5.12.

Rysunek 5.12: Rozkład pracy załogi baru

Iloczynem kartezjańskim z definicji rozkładowania jest w tym przypadku iloczyn kartezjański *zbioru dni tygodnia* (poniedziałek, wtorek,...,niedziela) i *zbioru zapotrzebowanych pracowników* (pracownik_1, pracownik_2, ..., pracownik_20). Iloczyn ten to zbiór par (dzień, pracownik) dla wszystkich "kratek" z rysunku 5.12. Rozwiązaniem problemu są podzbiory iloczynu kartezjańskiego, zaznaczo-

ne na rysunku kolorami:

- czerwonym (pracownicy podejmujący pracę w poniedziałek i pracujący do piątku);
- zielonym (pracownicy podejmujący pracę we wtorek i pracujący do soboty);
- granatowym (pracownicy podejmujący pracę w czwartek i pracujący do poniedziałku);
- żółtym (pracownicy podejmujący pracę w piątek i pracujący do wtorku);
- szarym (pracownicy podejmujący pracę w sobotę i pracujący do środy).

5.9.2 Blaski i nędza optymalizacji

Optymalność rozwiązania oznacza tylko tyle, że maksymalizuje ono wybrany wskaźnik jakości. Praktycznie jakość rozwiązania optymalnego może (w pewnych sytuacjach) bardzo odbiegać od naszych intuicyjnych oczekiwań. Sprostanie tym oczekiwaniom może wymagać zmiany założeń: modelu matematycznego problemu i jego wskaźnika jakości. Ilustruje to następujący przykład harmonogramowania pracy załogi.

5.9.3 Kolekta opłat autostradowych

Przed wjazdem na 4-pasmową płatną autostradę jest 8 stanowisk kolektorów opłat. Z przeprowadzonych badań wynika, że zapotrzebowanie na kolektorów opłat o różnych porach dnia i nocy w okresie 24 godzinnym jest następujące:

```
od godz. 24 do  6  - 2 kolektorów
od godz.  6 do 10  - 8 kolektorów
od godz. 10 do 12  - 4 kolektorów
od godz. 12 do 16  - 3 kolektorów
od godz. 16 do 18  - 6 kolektorów
od godz. 18 do 22  - 5 kolektorów
od godz. 22 do 24  - 3 kolektorów
```

W ciągu 24 godzin każdy kolektor pracuje 4 godziny, ma godzinę przerwy i pracuje następne 4 godziny. Kolektor opłat może rozpocząć pracę o dowolnej godzinie. Należy wyznaczyć minimalną liczbę potrzebnych kolektorów opłat i określić godziny, o których rozpoczynają pracę. W tym celu definiuje się następujące zmienne:

```
X1 - liczba kolektorów rozpoczynających pracę o godz. 1
X2 - liczba kolektorów rozpoczynających pracę o godz. 2
```

```
X3 - liczba kolektorów rozpoczynających pracę o godz. 3
X4 - liczba kolektorów rozpoczynających pracę o godz. 4
...
X24 -liczba kolektorów rozpoczynających pracę o godz. 24.
```

Odpowiedni program (`5_30_kolektorzy.ecl`) ma postać:

```
/*1*/  :- lib(eplex).

/*2*/   top:-
/*3*/       Zmienne = [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,
                  X11,X12,X13,X14,X15,X16,X17,X18,
                      X19,X20,X21,X22,X23,X24],
/*4*/        Zmienne $:: 0.0..1.0Inf,
/*5*/        integers(Zmienne),

          %Liczba kolektorów pracujących od 24 do 1:
/*6*/         X24+X23+X22+X21+X19+X18+X17+X16 $>= 2,
          %Liczba kolektorów pracujących od 1 do 2:
/*7*/         X1+X24+X23+X22+X20+X19+X18+X17 $>= 2,
          %Liczba kolektorów pracujących od 2 do 3:
/*8*/         X2+X1+X24+X23+X21+X20+X19+X18 $>= 2,
          %Liczba kolektorów pracujących od 3 do 4:
/*9*/        X3+X2+X1+X24+X22+X21+X20+X19 $>= 2,
          %Liczba kolektorów pracujących od 4 do 5:
/*10*/        X4+X3+X2+X1+X23+X22+X21+X20 $>= 2,
          %Liczba kolektorów pracujących od 5 do 6:
/*11*/        X5+X4+X3+X2+X24+X23+X22+X21 $>= 2,
          %Liczba kolektorów pracujących od 6 do 7:
/*12*/        X6+X5+X4+X3+X1+X24+X23+X22 $>= 8,
          %Liczba kolektorów pracujących od 7 do 8:
/*13*/        X7+X6+X5+X4+X2+X1+X24+X23 $>= 8,
          %Liczba kolektorów pracujących od 8 do 9:
/*14*/        X8+X7+X6+X5+X3+X2+X1+X24 $>= 8,
          %Liczba kolektorów pracujących od 9 do 10:
/*15*/        X9+X8+X7+X6+X4+X3+X2+X1 $>= 8,
          %Liczba kolektorów pracujących od 10 do 11:
/*16*/        X10+X9+X8+X7+X5+X4+X3+X2 $>= 4,
          %Liczba kolektorów pracujących od 11 do 12:
/*17*/        X11+X10+X9+X8+X6+X5+X4+X3 $>= 4,
          %Liczba kolektorów pracujących od 12 do 13:
/*18*/        X12+X11+X10+X9+X7+X6+X5+X4 $>= 3,
          %Liczba kolektorów pracujących od 13 do 14:
/*19*/        X13+X12+X11+X10+X8+X7+X6+X5 $>= 3,
          %Liczba kolektorów pracujących od 14 do 15:
/*20*/        X14+X13+X12+X11+X9+X8+X7+X6 $>= 3,
          %Liczba kolektorów pracujących od 15 do 16:
```

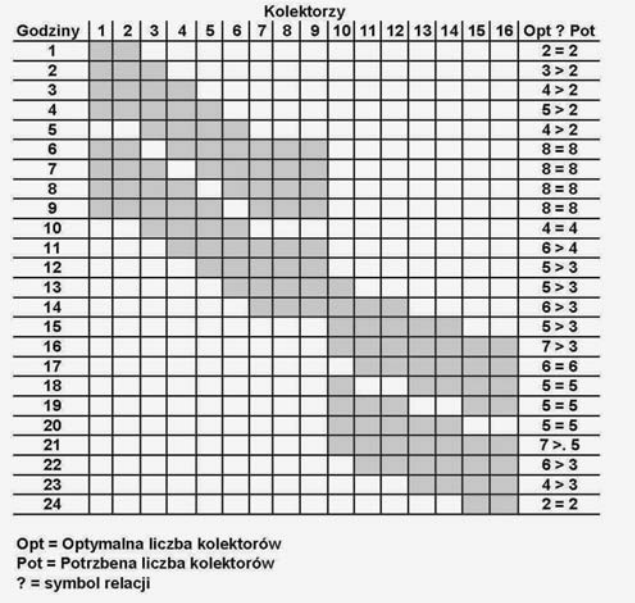
```
/*21*/        X15+X14+X13+X12+X10+X9+X8+X7 $>= 3,
          %Liczba kolektorów pracujących od 16 do 17:
/*22*/        X16+X15+X14+X13+X11+X10+X9+X8 $>= 6,
          %Liczba kolektorów pracujących od 17 do 18:
/*23*/        X17+X16+X15+X14+X12+X11+X10+X9 $>= 6,
          %Liczba kolektorów pracujących od 18 do 19:
/*24*/        X18+X17+X16+X15+X13+X12+X11+X10 $>= 5,
          %Liczba kolektorów pracujących od 19 do 20:
/*25*/        X19+X18+X17+X16+X14+X13+X12+X11 $>= 5,
          %Liczba kolektorów pracujących od 20 do 21:
/*26*/        X20+X19+X18+X17+X15+X14+X13+X12 $>= 5,
          %Liczba kolektorów pracujących od 21 do 22:
/*27*/        X21+X20+X19+X18+X16+X15+X14+X13+X12 $>= 5,
          %Liczba kolektorów pracujących od 22 do 23:
/*28*/        X22+X21+X20+X19+X17+X16+X15+X14 $>= 3,
          %Liczba kolektorów pracujących od 23 do 24:
/*29*/        X23+X22+X21+X20+X18+X17+X16+X15 $>= 3,

/*30*/       Liczba_kolektorow $= X1+X2+X3+X4+X5+X6+
               X7+X8+X9+X10+X11+X12+X13+X14+X15+X16+X17+
                     X18+X19+X20+X21+X22+X23+X24),

/*31*/       eplex_solver_setup(min(Liczba_kolektorow)),
/*32*/       eplex_solve(Liczba_kolektorow),
/*33*/       eplex_get(vars,Vars),
/*34*/       eplex_get(typed_solution,Vals),
/*35*/       Vars = Vals,nl,

/*36*/       Liczba is X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12
                     +X13+X14+X15+X16+X17+X18+X19+X20+X21+X22+X23+X24,

/*37*/      write("Liczba_kolektorow = "),write(Liczba),nl,

/*37*/      (foreach(A,
             ["1","2","3","4","5","6","7","8","9","10","11","12",
             "13","14","15","16","17","18","19","20","21","22","23","24"]),
/*38*/      foreach(X,
             [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,
             X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24])
/*39*/      do
/*40*/      write("Liczba kolektorow zaczynajaca od godz. "),
                          write(A),write(" = "),write(X),nl).
```

Rozwiązanie uzyskane po czasie 0.02s ma postać:

```
Liczba_kolektorow = 16
    Liczba kolektorow zaczynajaca od godz. 1   = 2
    Liczba kolektorow zaczynajaca od godz. 2   = 1
    Liczba kolektorow zaczynajaca od godz. 3   = 1
    Liczba kolektorow zaczynajaca od godz. 4   = 1
    Liczba kolektorow zaczynajaca od godz. 5   = 1
    Liczba kolektorow zaczynajaca od godz. 6   = 3
    Liczba kolektorow zaczynajaca od godz. 7   = 0
    Liczba kolektorow zaczynajaca od godz. 8   = 0
    Liczba kolektorow zaczynajaca od godz. 9   = 0
    Liczba kolektorow zaczynajaca od godz. 10  = 0
    Liczba kolektorow zaczynajaca od godz. 11  = 0
    Liczba kolektorow zaczynajaca od godz. 12  = 0
    Liczba kolektorow zaczynajaca od godz. 13  = 1
    Liczba kolektorow zaczynajaca od godz. 14  = 2
    Liczba kolektorow zaczynajaca od godz. 15  = 2
    Liczba kolektorow zaczynajaca od godz. 16  = 2
    Liczba kolektorow zaczynajaca od godz. 17  = 0
    Liczba kolektorow zaczynajaca od godz. 18  = 0
    Liczba kolektorow zaczynajaca od godz. 19  = 0
    Liczba kolektorow zaczynajaca od godz. 20  = 0
    Liczba kolektorow zaczynajaca od godz. 21  = 0
    Liczba kolektorow zaczynajaca od godz. 22  = 0
    Liczba kolektorow zaczynajaca od godz. 23  = 0
    Liczba kolektorow zaczynajaca od godz. 24  = 0
```

Rozwiązanie przedstawiono graficznie na rysnku 5.13. Iloczynem kartezjańskim z definicji rozkładowania jest w tym przypadku iloczyn kartezjański *zbioru godzin* (1,2,...,24) i *zbioru kolektorów* (1,2,...). Iloczyn ten to zbiór par (godzina pracy, kolektor pracujący) dla wszystkich "kratek" z rysunku 5.13. Rozwiązaniem problemu są podzbiory iloczynu kartezjańskiego, zaznaczone na rysunku kolorem szarym, idąc od góry rysunku w dół:

- dwóch kolektorów podejmujących pracę o godz. 24 i pracujących - z godzinną przerwą - do godz 9;
- itd.,...,itd.
- dwóch kolektorów podejmujących pracę o godz. 15 i pracujących - z godzinną przerwą - do godz 24.



Rysunek 5.13: Rozkład pracy kolektorów

Rozwiązanie to - aczkolwiek formalnie optymalne - jest rozrzutne: dla dużej liczby godzin liczba pracujących kontrolerów jest większa od zapotrzebowania. Można to spróbować naprawić na drodze zmiany regulaminu pracy lub na drodze rewizji zapotrzebowania.

5.9.4 Psy

Firma „Dog Service" jest potentatem w dziedzinie szkolenia i wypożyczania psów dla wykrywania narkotyków, alkoholu, papierosów oraz materiałów wybuchowych. Każdy pies wypożyczony z firmy „Dog Service" jest ekspertem w każdym z wymienionych wykrywań. Z usług firmy „Dog Service" korzysta również „Duże Wschodnie Przejście Graniczne". Wypożyczony pies wraz ze swym przewodnikiem mogą pracować na dobę nie dłużej niż 6 godzin, po każdej godzinie pracy jest co najmniej godzina wypoczynku. Zapotrzebowanie „Dużego Wschodniego Przejścia Granicznego" na psy, wyznaczone na drodze analizy ba-

zy danych, jest zależne od pory dnia i kształtuje się (w dobach roboczych) następująco:

od godz. 24 do 4 - 2 psy pracujące
od godz. 4 do 8 - 4 psy pracujące
od godz. 8 do 10 - 6 psów pracujących
od godz. 10 do 12 - 8 psów pracujących
od godz. 12 do 16 - 6 psów pracujących
od godz. 16 do 18 - 4 psy pracujące
od godz. 18 do 22 - 2 psy pracujące
od godz. 22 do 24 - 3 psy pracujące

Ile psów powinno o każdej godzinie roboczej doby przystąpić do pracy na „Dużym Wschodnim Przejściu Granicznym", aby całkowita liczba psów pracujących tam w ciągu roboczej doby była minimalna? Zadanie to rozwiązuje program `5_31_psy.ecl`:

```
/*1*/  :- lib(eplex).
 /*2*/ top:-
/*3*/ Psy = [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,
             X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24],
/*4*/  Psy $: :   0.0..1.0Inf,
/*5*/  integers(Psy),

   %  Liczba psów pracujących od 24 do 1:
/*6*/  X24+X22+X20+X18+X16+X14 $>= 2,

  %  Liczba psów pracujących od 1 do 2:
/*7*/  X1+X23+X21+X19+X17+X15 $>= 2,

  %  Liczba psów pracujących od 2 do 3:
/*8*/  X2+X24+X22+X20+X18+X16 $>= 2,

  %  Liczba psów pracujących od 3 do 4:
/*9*/  X3+X1+X23+X21+X19+X17 $>= 2,

  %  Liczba psów pracujących od 4 do 5:
/*10*/  X4+X2+X24+X22+X20+X18 $>= 4,

  %  Liczba psów pracujących od 5 do 6:
/*11*/  X5+X3+X1+X23+X21+X19 $>= 4,

  %  Liczba psów pracujących od 6 do 7:
/*12*/  X6+X4+X2+X24+X22+X20 $>= 4,
```

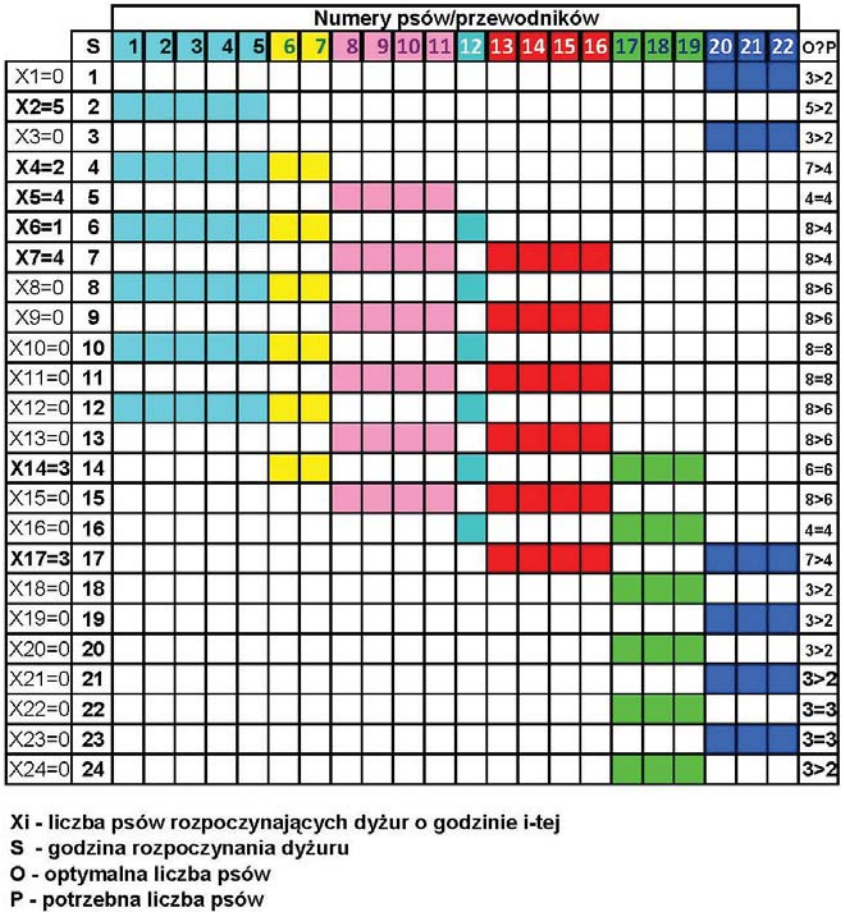
```
  %  Liczba psów pracujących od 7 do 8:
/*13*/  X7+X5+X3+X1+X23+X21 $>= 4,

  %  Liczba psów pracujących od 8 do 9:
/*14*/  X8+X6+X4+X2+X24+X22 $>= 6,

  %  Liczba psów pracujących od 9 do 10:
/*15*/   X9+X7+X5+X3+X1+X23 $>= 6,

  %  Liczba psów pracujących od 10 do 11:
/*16*/  X10+X8+X6+X4+X2+X24 $>= 8,

  %  Liczba psów pracujących od 11 do 12:
/*17*/  X11+X9+X7+X5+X3+X1 $>= 8,

  %  Liczba psów pracujących od 12 do 13:
/*18*/  X12+X10+X8+X6+X4+X2 $>= 6,

  %  Liczba psów pracujących od 13 do 14:
/*19*/  X13+X11+X9+X7+X5+X3 $>= 6,

  %  Liczba psów pracujących od 14 do 15:
/*20*/  X14+X12+X10+X8+X6+X4 $>= 6,

  %  Liczba psów pracujących od 15 do 16:
/*21*/  X15+X13+X11+X9+X7+X5 $>= 6,

  %  Liczba psów pracujących od 16 do 17:
/*22*/  X16+X14+X12+X10+X8+X6 $>= 4,  ,

  %  Liczba psów pracujących od 17 do 18:
/*23*/  X17+X15+X13+X11+X9+X7 $>= 4, ,

  %  Liczba psów pracujących od 18 do 19:
/*24*/  X18+X16+X14+X12+X10+X8 $>= 2,

  %  Liczba psów pracujących od 19 do 20:
/*25*/  X19+X17+X15+X13+X11+X9 $>= 2,

  %  Liczba psów pracujących od 20 do 21:
/*26*/  X20+X18+X16+X14+X12+X10 $>= 2,

  %  Liczba psów pracujących od 21 do 22:
/*27*/  X21+X19+X17+X15+X13+X11 $>= 2,

  %  Liczba psów pracujących od 22 do 23:
/*28*/  X22+X20+X18+X16+X14+X12 $>= 3,
```

```
  %  Liczba psów pracujących od 23 do 24:
/*29*/  X23+X21+X19+X17+X15+X13 $>= 3,

/*30*/  (Liczba_psow $= X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12+
              X13+X14+X15+X16+X17+X18+X19+X20+X21+X22+X23+X24),

/*31*/  eplex_solver_setup(min(Liczba_psow)),
/*32*/  eplex_solve(Liczba_psow),
/*33*/  eplex_get(vars,Vars),
/*34*/  eplex_get(typed_solution,Vals),
/*35*/  Vars = Vals,nl,

/*36*/  Liczba is X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12+
              X13+X14+X15+X16+X17+X18+X19+X20+X21+X22+X23+X24,
/*37*/  write("Liczba potrzebnych psow = "),write(Liczba),nl,nl,

/*38*/ (foreach(A,
          ["1","2","3","4","5","6","7","8","9","10","11"," 2",
           "13","14","15","16","17"," 8","19","20","21","22","23","24"]),
/*39*/  foreach(X,
          [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,
           X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24])
/*40*/  do
/*41*/  write("O godzinie "),write(A),write(" rozpoczyna prace "),write(X),
                write(" psow. "),nl).
```

Program generuje komunikat:

```
Liczba potrzebnych psow = 22
O godzinie 1  rozpoczyna prace 0 psow.  O godzinie 2  rozpoczyna prace 5 psow.
O godzinie 3  rozpoczyna prace 0 psow.  O godzinie 4  rozpoczyna prace 2 psow.
O godzinie 5  rozpoczyna prace 4 psow.  O godzinie 6  rozpoczyna prace 1 psow.
O godzinie 7  rozpoczyna prace 4 psow.  O godzinie 8  rozpoczyna prace 0 psow.
O godzinie 9  rozpoczyna prace 0 psow.  O godzinie 10 rozpoczyna prace 0 psow.
O godzinie 11 rozpoczyna prace 0 psow. O godzinie 12 rozpoczyna prace 0 psow.
O godzinie 13 rozpoczyna prace 0 psow. O godzinie 14 rozpoczyna prace 3 psow.
O godzinie 15 rozpoczyna prace 0 psow.  O godzinie 16 rozpoczyna prace 0 psow.
O godzinie 17 rozpoczyna prace 3 psow.  O godzinie 18 rozpoczyna prace 0 psow.
O godzinie 19 rozpoczyna prace 0 psow.  O godzinie 20 rozpoczyna prace 0 psow.
O godzinie 21 rozpoczyna prace 0 psow.  O godzinie 22 rozpoczyna prace 0 psow.
O godzinie 23 rozpoczyna prace 0 psow.  O godzinie 24 rozpoczyna prace 0 psow.
```

Rozkład pracy psów został przedstawiony na rysunku 5.14. Przedstawiono tam również odchyłki optymalnej liczby psów od liczby psów potrzebnej.



Rysunek 5.14: Rozkład pracy psów

5.9.5 Policjanci

Miejski Komisariat Policji[5] powinien dysponować w kolejnych godzinach doby co najmniej taką liczbą policjantów, która została przedstawiona w tablicy 5.7:

Godziny	Okres	Niezbędna liczba policjantów
2 - 6	1	22
6 - 10	2	55
10 - 14	3	88
14 - 18	4	110
18 - 22	5	44
22 - 2	6	33

Tablica 5.7: Zapotrzebowanie na policjantów

Zapotrzebowanie to zostało określone dla kolejnych 6 okresów 4-godzinnych. Policjant rozpoczyna pracę z początkiem takiego okresu i pracuje 8 kolejnych, do końca następnego okresu godzin, tylko raz w ciągu doby. Należy wyznaczyć liczbę policjantów rozpoczynających pracę w kolejnych okresach tak, by całkowita liczba potrzebnych w ciągu doby policjantów była minimalna. Czyni to program (`5_32_policjanci.ecl`):

```
/*1*/    :-lib(ic).
/*2*/    :-lib(branch_and_bound).

/*3*/    top:-
  % Poli_i - liczba policjantów rozpoczynająca służbę z początkiem okresu i-tego
/*4*/             [Poli1,Poli2,Poli3,Poli4,Poli5,Poli6] :: 0..200,

   %Policjanci w pracy w okresie 1:
/*5*/             Poli6 + Poli1 #>= 22,

   %Policjanci w pracy w okresie 2:
/*6*/             Poli1 + Poli2 #>= 55,

   %Policjanci w pracy w okresie 3:
/*7*/             Poli2 + Poli3 #>= 88,
```

[5]Temat przykładu pochodzi z książki [Wagner-75], rozwiązanie zostało opracowane przez Autora.

```
   %Policjanci w pracy w okresie 4:
/*8*/             Poli3 + Poli4 #>= 110,

   %Policjanci w pracy w okresie 5:
/*9*/             Poli4 + Poli5 #>= 44,

   %Policjanci w pracy w okresie 6:
/*10*/            Poli5 + Poli6 #>= 33,

/*11*/            Suma  #= Poli1 + Poli2 + Poli3 + Poli4 + Poli5 + Poli6,

/*12*/            bb_min(labeling([Poli1,Poli2,Poli3,
                           Poli4,Poli5,Poli6]),
                           Suma,bb_options with [strategy:step]),nl,

/*13*/           write("Liczba policjantow zaczynajaca prace z poczatkiem:"),nl,
/*14*/           write("okresu 1: "),write(Poli1),nl,
/*15*/           write("okresu 2: "),write(Poli2),nl,
/*16*/           write("okresu 3: "),write(Poli3),nl,
/*17*/           write("okresu 4: "),write(Poli4),nl,
/*18*/           write("okresu 5: "),write(Poli5),nl,
/*19*/           write("okresu 6: "),write(Poli6),nl,nl,
/*20*/           write("Minimalna liczba potrzebnych policjantow: "),write(Suma).
```

Rozwiązanie ma postać:

```
Found a solution with cost 198
Found no solution with cost 20.0 .. 197.0

Liczba policjantow zaczynajaca prace z poczatkiem:
okresu 1: 0
okresu 2: 55
okresu 3: 33
okresu 4: 77
okresu 5: 0
okresu 6: 33

Minimalna liczba potrzebnych policjantow: 198
```

Zbadajmy jeszcze sprawę wielokrotności optymalnego rozwiazania. Jak to już stwierdzono w rozdziale 5.6.1, nie da się tego uzyskać za pomoca predykatu `fail`, lecz jedynie na drodze wyznaczania rozwiązań dopuszczalnych dla zadanego wskaźnika jakości, równego wskaźnikowi optymalnemu. Zrobiono to w programie `5_33_policjanci_wszyscy.ecl`, gdzie liczbę policjantów ustalono

na wartości optymalnej 198:

```
/*1*/    :-lib(ic).

/*2*/    top:-
/*3*/             assert(licznik(0)),
  % Poli_i - liczba policjantów rozpoczynająca służbę z początkiem okresu i-tego
/*4*/             [Poli1,Poli2,Poli3,Poli4,Poli5,Poli6] :: 0..200,

   %Policjanci  w pracy w okresie 1:
/*5*/             Poli6 + Poli1 #>= 22,

   %Policjanci  w pracy w okresie 2:
/*6*/             Poli1 + Poli2 #>= 55,

   %Policjanci  w pracy w okresie 3:
/*7*/             Poli2 + Poli3 #>= 88,

   %Policjanci  w pracy w okresie 4:
/*8*/             Poli3 + Poli4 #>= 110,

   %Policjanci  w pracy w okresie 5:
/*9*/             Poli4 + Poli5 #>= 44,

   %Policjanci  w pracy w okresie 6:
/*10*/            Poli5 + Poli6 #>= 33,

/*11*/            198  #= Poli1 + Poli2 + Poli3 + Poli4 + Poli5 + Poli6,
/*12*/            labeling([Poli1,Poli2,Poli3,Poli4,Poli5,Poli6]),

/*13*/            licz,
/*14*/            licznik(Liczba),

/*15*/           write("Liczba policjantow zaczynajaca prace z poczatkiem:"),nl,
/*16*/           write("okresu 1: "),write(Poli1),nl,
/*17*/           write("okresu 2: "),write(Poli2),nl,
/*18*/           write("okresu 3: "),write(Poli3),nl,
/*19*/           write("okresu 4: "),write(Poli4),nl,
/*20*/           write("okresu 5: "),write(Poli5),nl,
/*21*/           write("okresu 6: "),write(Poli6),nl,nl,
/*22*/           write("Minimalna liczba potrzebnych policjantow: "),write(Suma),
/*23*/           nl,nl,fail.

/*24*/   top:-
/*25*/           write("To wszystkie rozwiazania!").

/*26*/   licz:-
/*27*/           retract(licznik(Stary)),
```

```
/*28*/          Nowy is Stary  1, +
/*29*/          assert(licznik(Nowy)).
```

Rozwiązań optymalnych jest 32154 (słownie: trzydzieści dwa tysiące sto pięćdziesiąt cztery), dlatego pokażemy poniżej tylko pierwsze i ostatnie:

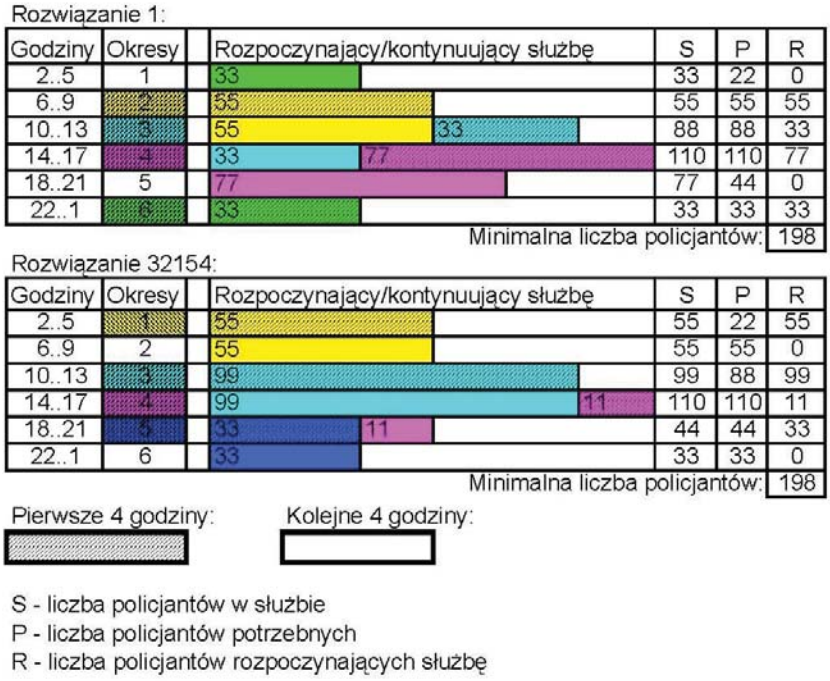
```
  Rozwiazanie 1:
  Liczba policjantow zaczynajaca sluzbe z poczatkiem:
  okresu 1: 0
  okresu 2: 55
  okresu 3: 33
  okresu 4: 77
  okresu 5: 0
  okresu 6: 33
 Minimalna liczba potrzebnych  policjantow: 198

    ..............................

 Rozwiazanie 32154:
  Liczba policjantow zaczynajaca sluzbe z poczatkiem:
  okresu 1: 55
  okresu 2: 0
  okresu 3: 99
  okresu 4: 11
  okresu 5: 33
  okresu 6: 0
 Minimalna liczba potrzebnych  policjantow: 198
```

Obydwa rozwiązania przedstawiono na rysunku 5.15:

5.10 Przykłady szeregowania

W przypadku rozkładowania:

- nie mieliśmy do czynienia z *ograniczeniami kolejnościowymi*, żądającymi zakończenia pewnych zadań przed rozpoczęciem innych zadań, oraz wykluczającymi równoległą w czasie realizację pewnych zadań;
- nie mieliśmy do czynienia z *ruchomymi* czasami rozpoczęcia i zakończenia zadań, lecz z sztywną siatką przedziałów czasowych (dni tygodnia, godziny od-do), w których zadania powinny się zmieścić.



Rysunek 5.15: Pierwsze i ostatnie rozwiązanie rozkładu pracy policjantów

Szeregowanie określonego zbioru zadań polega na wyznaczeniu takiej kolejności wykonywania zadań z tego zbioru, która spełnia wszystkie ograniczenia kolejnościowe, ograniczenia nierównoczesnościowe i ograniczenia czasowe zadań przy minimalizacji całkowitego czasu wykonania wszystkich zadań

Elementami problemu szeregowania są:

- *ograniczenia kolejnościowe*, zgodnie z którymi jakaś czynność `A` może rozpocząć się dopiero po zakończeniu innej czynności `B`, dla której znany jest `Czas_trwania_B`:

```
Start_A #>= Start_B + Czas_trwania_B.
```

- *ograniczenia nierównoczesności*, zwane również *ograniczeniami dyzjunktywnymi*, zgodnie z którymi w przypadku dwóch czynności `A` i `B` o czasach trwania odpowiednio równych `Czas_trwania_A` i `Czas_trwania_B`, korzystających z tego samego resursu mogącego obsługiwać tylko jedną z tych czynności, czynność `B` może się odbywać dopiero po zakończeniu czynności `A` lub czynność `A` może się odbywać dopiero po zakończeniu czynności `B`. Można to zapisać ograniczeniami:

```
dyzjunktywne(Start_A,Czas_trwania_A,Start_B,Czas_trwania_B):-
    Start_A #>= Start_B + Czas_trwania_B.
dyzjunktywne(Start_A,Czas_trwania_A,Start_B,Czas_trwania_B):-
    Start_B #>= Start_A + Czas_trwania_A.
```

- *ograniczenia resursów*, zgodnie z którymi wykonanie pewnej czynności wymaga resursu określonej wielkości `Resurs_A`, nie większego od pewnego resursu granicznego `Resurs_graniczny`:

```
Resurs_A #=< Resurs_graniczny.
```

Zarówno ograniczenia kolejnościowe jak i ograniczenia nierównoczesności oraz ruchome czasy rozpoczęcia i zakończenia czynności są nieodłącznymi elementami ogromnej liczby procesów w przemyśle i biznesie.

5.10.1 Ograniczenia kolejnościowe - budujemy dom

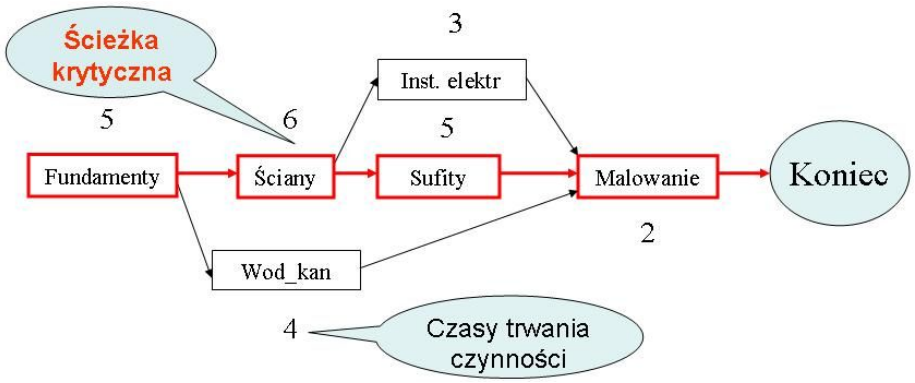
Ograniczenia kolejnościowe występują z reguły przy pracach budowlanych, gdzie coś musi być zrobione zanim można przystąpić do wykonywania czegoś innego. Przykładem zagadnienia z ograniczeniami kolejnościowymi może być budowa domu jednorodzinnego. Tablica 5.8 przedstawia czynności (nazwane dla uproszczenia nazwami elementów budowy), czasy trwania tych czynności oraz ograniczenia kolejnościowe.

Na rysunku 5.16 przedstawiono sieć ograniczeń kolejnościowych pomiędzy czynnościami i czasów ich trwania. Sieć ta jest siecią typu *aktywności na łuku* (ang. *AoA - Activities on Arc*). Ograniczenia resursów występują w tym przykładzie w postaci ukrytej, a mianowicie jako czasy trwania poszczególnych

Czynność	Czas trwania	Poprzednik
Fundamenty	5	Nie ma
Ściany	6	Fundamenty
`Wod_kan`	3	Fundamenty
Sufity	5	Ściany
Inst. elektr.	3	Ściany
Malowanie	2	Inst. elektr. `Wod_kan` Sufity
Koniec	0	Malowanie

Tablica 5.8: Dane dla budowy domu

czynności: gdybyśmy np. nie mieli ograniczonej liczby koparek, pomp betonowych, cieśli i murarzy, to być może zdołalibyśmy wykonać fundamenty w ciągu jednego dnia.



Rysunek 5.16: Sieć aktywności na łuku dla budowy domu jednorodzinnego

Zakładając, że budowa rozpoczyna się w dniu 0, jesteśmy zainteresowani wyznaczeniem minimalnego czasu trwania budowy domu (tzn. najwcześniejszego terminu zakończenia budowy). Chcielibyśmy również znać *ścieżkę krytyczną*[6]

[6]Omawiane zagadnienie jest rozwiązywane w badaniach operacyjnych pod hasłem *PERT*

tzn. najkrótszą sekwencję czynności poczynając od czynności początkowej a kończąc na czynności końcowej. Znajomość tej ścieżki jest istotna dlatego, gdyż jedynym sposobem skrócenia czasu trwania budowy jest skrócenie czasu trwania czynności ze ścieżki krytycznej. Program `5_34_dom.ecl` umożliwia wyznaczenie trzech wariantów minimalnego rozwiązania:

1. Wyznaczenie pojedynczego rozwiązania minimalnego. W tym celu należy odkomentować wszystkie linie o numerach `/*?a*/` a zakomentować wszystkie linie o numerach `/*?b*/`, co odpowiada wymienionemu przykładowi.

2. Wyznaczenie - dla znanego pojedynczego rozwiązania minimalnego - wszystkich rozwiązań minimalnych. W tym celu należy odkomentować wszystkie linie o numerach `/*?b*/` a zakomentować wszystkie linie o numerach `/*?a*/`

3. Wyznaczenie Ścieżki krytycznej. Dla programu wyznaczającego wszystkie rozwiazania minimalne należy dodatkowo odkomentować wszystkie linie o numerach `/*x*/` i zakomentować wszystkie linie o numerach `/*17b*/`, `/*18*/`, `/*19*/`,...,`/*24*/`.

Program ten jest postaci:

```
/*1*/  :-lib(ic).
      % dla wyznaczenia pojedynczego rozwiązania minimalnego:
/*2a*/  :-lib(branch_and_bound).

/*3*/  top:-

      % dla wyznaczenia pojedynczego rozwiązania minimalnego:
/*4a*/  dom(_).

      % dla wyznaczenia wszystkich rozwiązań minimalnych:
/*4b*/%       findall(Operacje, dom(Operacje),_).

/*5*/  dom(Operacje):-
/*6*/    Operacje = [Fundamenty,Sciany,Wod_kan,Sufity,Inst_elektr,
                  Malowanie,Koniec],

      % dla wyznaczenia pojedynczego rozwiązania minimalnego:
/*7a*/    Operacje  :: 0..25,
```

(*Program Evaluation and Review Technique*) albo *CPM* (*Critical Path Method*). Metody te są standardowymi metodami stosowanymi przy zarządzaniu dużymi projektami. Ich pierwszym zastosowaniem był projekt *Polaris* budowy rakiet balistycznych dla okrętów podwodnych. Dzięki wymienionym metodom został on podobno zakończony 18 miesięcy przed terminem.

```
      % dla wyznaczenia wszystkich rozwiązań minimalnych:
/*7b*/%       Operacje  :: 0..18,

/*8*/    Sciany #>= Fundamenty + 5,
/*9*/    Wod_kan #>= Fundamenty + 5,
/*10*/   Sufity #>= Sciany + 6,
/*11*/   Inst_elektr #>= Sciany + 6,
/*12*/   Malowanie #>=  Inst_elektr + 3,
/*13*/   Malowanie #>= Wod_kan + 4,
/*14*/   Malowanie #>=  Sufity + 5,
/*15*/   Koniec #>= Malowanie + 2,

      % dla wyznaczenia pojedynczego rozwiązania minimalnego:
/*16a*/  Koniec #=< 25,

      % dla wyznaczenia wszystkich rozwiązań minimalnych:
/*16b*/%     Koniec #= 18,

      % dla wyznaczenia pojedynczego rozwiązania minimalnego:
/*17a*/  minimize(labeling(Operacje),Koniec),nl,

      % dla wyznaczenia wszystkich rozwiązań minimalnych:
/*17b*/%     labeling(Operacje),

%/*x*/   get_domain(Fundamenty,  DFundamenty),
%/*x*/   get_domain(Sciany,  DSciany),
%/*x*/   get_domain(Wod_kan,  DWod_kan),
%/*x*/   get_domain(Sufity,  DSufity),
%/*x*/   get_domain(Inst_elektr,  DInst_elektr),
%/*x*/   get_domain(Malowanie,  DMalowanie),
%/*x*/   get_domain(Koniec,  DKoniec),

%/*x*/   write("Dziedzina fundamentow: "),  write(DFundamenty),  nl,
%/*x*/   write("Dziedzina scian: "),  write(DSciany),  nl,
%/*x*/   write("Dziedzina wod_kan: "),  write(DWod_kan),  nl,
%/*x*/   write("Dziedzina sufitow: "),  write(DSufity),  nl,
%/*x*/   write("Dziedzina inst_elektr: "),  write(DInst_elektr),  nl,
%/*x*/   write("Dziedzina malowania: "),  write(DMalowanie),  nl,
%/*x*/   write("Dziedzina konca "),  write(DKoniec),  nl,

/*18*/  write("Czas rozpoczecia fundamentow: "),write(Fundamenty),
/*19*/  nl,write("Czas rozpoczecia scian: "),write(Sciany),nl,
/*20*/  write("Czas rozpoczecia wod_kan: "),write(Wod_kan),nl,
/*21*/  write("Czas rozpoczecia sufitow: "),write(Sufity),nl,
/*22*/  write("Czas rozpoczecia inst_elektr: "),write(Inst_elektr),
/*23*/  nl,write("Czas rozpoczecia malowania: "),write(Malowanie),nl,
/*24*/  write("Koniec: "),write(Koniec),nl.
```

Dla przypadku wyznaczania pojedynczego rozwiązania optymalnego program generuje komunikat:

```
Found a solution with cost 18

Czas rozpoczecia fundamentow: 0
Czas rozpoczecia scian: 5 Czas
rozpoczecia wod_kan: 5
Czas rozpoczecia sufitow: 11
Czas rozpoczecia
inst_elektr: 11
Czas rozpoczecia malowania: 16
Koniec: 18
```

Aby określić wszystkie rozwiązania optymalne, wyznaczony minimalny czas budowy (Koniec: 18) został zastosowany jako górny kraniec dziedziny zmiennej `Operacje` w linii `/*7b*/` dla przypadku wyznaczania wszystkich rozwiązań optymalnych. W tym celu należy odkomentować wszystkie linie o numerach `/*?b*/` a zakomentować wszystkie linie o numerach `/*?a*/`, otrzymując program `5_35_dom_wszystkie.ecl`. Generuje on komunikat o 10 rozwiązaniach, z których pokazano tylko pierwsze trzy:

```
Czas rozpoczecia fundamentow:0
Czas rozpoczecia scian: 5
Czas rozpoczecia wod_kan: 5
Czas rozpoczecia sufitow: 11
Czas rozpoczecia inst_elektr: 11
Czas rozpoczecia malowania: 16
Koniec: 18

Czas rozpoczecia fundamentow: 0
Czas rozpoczecia scian: 5
Czas rozpoczecia wod_kan: 5
Czas rozpoczecia sufitow: 11
Czas rozpoczecia inst_elektr: 12
Czas rozpoczecia malowania: 16
Koniec: 18

Czas rozpoczecia fundamentow: 0
Czas rozpoczecia scian: 5
Czas rozpoczecia wod_kan: 5
```

```
Czas rozpoczecia sufitow: 11
Czas rozpoczecia inst_elektr: 13
Czas rozpoczecia malowania: 16
Koniec: 18
```

Aby wyznaczyć ścieżkę krytyczną, należy odkomentować wszystkie linie programu `5_35_dom_wszystkie.ecl` rozpoczynające się symbolem `/*x*/` i zakomentować linie `17b`, `18`, `19`,..., `24`, co daje program `5_36_dom_sc_kryt.ecl` generujący komunikat:

```
Dziedzina fundamentow: 0
Dziedzina scian: 5
Dziedzina wod_kan: 5 .. 12
Dziedzina sufitow: 11
Dziedzina inst_elektr: 11 .. 13
Dziedzina malowania: 16
Dziedzina konca 18
```

Sens tego komunikatu jest następujący: ponieważ dziedzinami o pojedynczej wartości są dziedziny fundamentów, ścian, sufitów, malowania i końca, czynności te są jedynymi znajdującymi się na ścieżce krytycznej, zob. rysunek 5.16.

5.10.2 Ograniczenia dyzjunktywne - ograniczone resursy

Szeregowanie z ograniczeniem dyzjunktywnym ilustruje program `5_37_szer_dyz.ecl` dla czterech zadań Z1,..Z4. Zadania `Z2` i `Z3` są dyzjunktywne, gdyż korzystają ze wspólnego resursu, którego wielkość umożliwia wyłącznie realizację jednego z tych zadań. Odpowiedni program `5_37_szer.dyz.ecl` jest następujący:

```
/*1*/  :-lib(ic).
/*2*/  :-lib(branch_and_bound).

/*3*/  top :-
/*4*/  szeregowanie(_).

/*5*/  szeregowanie([Z1,Z2,Z3,Z4,Koniec]):-
/*6*/       [Z1,Z2,Z3,Z4,Koniec] :: 0..15,
/*7*/       Z1 + 3 #=< Z2,
/*8*/       Z1 + 3 #=< Z3,
/*9*/       Z2 + 4 #=< Z4,
```

```
/*10*/     Z3 + 2 #=< Z4,
/*11*/     Z4 + 1 #= Koniec,
/*12*/     dyzjunktywne([Z2,4,Z3,2]),
/*13*/     minimize(labeling([Z1,Z2,Z3,Z4,Koniec]),Koniec),
/*14*/     writeln("Z1 ":Z1),
/*15*/     writeln("Z2 ":Z2),
/*16*/     writeln("Z3 ":Z3),
/*17*/     writeln("Z4 ":Z4),
/*18*/     writeln("Koniec ":Koniec).

/*19*/ dyzjunktywne([Z1,D1,Z2,_]):-
/*20*/     Z1 + D1 #=< Z2.

/*21*/ dyzjunktywne([Z1,_,Z2,D2]):-
/*22*/     Z2 + D2 #=< Z1.
```

Program generuje komunikat:

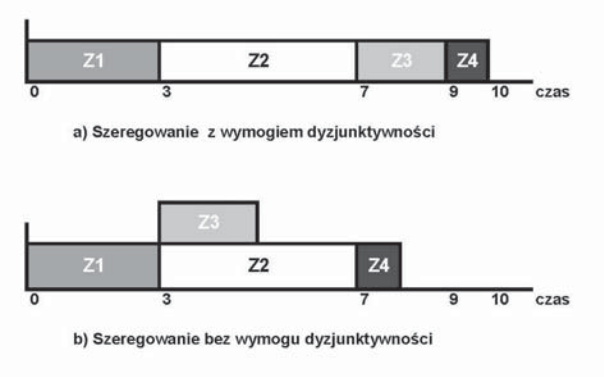
```
Found a solution with cost 10
Z1 : 0
Z2 : 3
Z3 : 7
Z4 : 9
Koniec  : 10,
```

któremu odpowiada wykres Gantta[7] z rysunku 5.17 a).

Jeżeli usunąć wymóg dyzjunktywności (tzn. zakomentować linię `/*12*/`, co odpowiada założeniu o dostępności resursu umożliwiającego równoczesne wykonywanie zadań `Z2` i `Z3`), otrzymuje się komunikat:

[7]Henry Gantt (1861-1919) był jednym z prekursorów nauki o zarządzaniu projektami. W latach 20 minionego stulecia opracował (zgodnie z danymi z witryny http://www.ganttchart.com/History.html) system wykresów czasowych ilustrujących współbieżnie realizowane operacje. System ten został nazwany wykresem Gantta. Wykresy Gantta zastosowano po raz pierwszy dla dużych projektów jak np. zapora Hoover Dam, w 1931 r.

Należy wspomnieć, że działający w zaborze rosyjskim i w Rosji Karol Adamiecki (1866-1933), organizator i dyrektor różnych przedsiębiorstw, jeden z współtwórców nauki o organizacji i zarządzaniu, już w 1903 r. przedstawił identyczną technikę (zwaną przez siebie "harmonogramem") w Towarzystwie Technicznym w Jekatierinosławiu (obecnie Dniepropietrowsk) w formie odczytu w języku rosyjskim pt. "Wykreślna metoda organizowania pracy zbiorowej w walcowniach". Adamiecki był wówczas dyrektorem technicznym w Towarzystwie Akcyjnym Walcowni Rur i Żelaza w Jekatierinosławiu.



Rysunek 5.17: Dwa przypadki szeregowania dyzjunktywnego

```
Found a solution with cost 8
Z1 : 0
Z2 : 3
Z3  : 3
Z4 : 7
Koniec  : 8,
```

któremu odpowiada wykres Gantta z rysunku 5.17 b), gdzie widać równoległość w czasie zadań Z2 i Z3.

Zauważmy, że liczba dyzjunkcji szybko wzrasta ze wzrostem liczby zadań korzystających ze wspólnego resursu. Jeżeli dla pojedynczego resursu i 2 zadań (jak w przykładzie) mamy 2 dyzjunkcje, to dla 3 zadań będzie $3 * 2 = 6$ dyzjunkcji, dla 4 zadań będzie $4 * 6 = 24$ dyzjunkcji, dla 5 zadań będzie $5 * 24 = 120$ dyzjunkcji, a dla n zadań będzie x_n dyzjunkcji, gdzie $x_n = n * x_{n-1}$. Dlatego rozwiązywanie dyzjunkcji zdefiniowanych jak w powyższym przykładzie jest numerycznie mało efektywne. Dlatego też, aczkolwiek zastosowany sposób deklaracji dyzjunkcji jest bardzo poglądowy (silnie deklaratywny), w dalszym ciągu nie będziemy z niego korzystać; w następnym rozdziale poznamy znacznie bardziej efektywny sposoby modelowania dyzjunkcji za pomocą ograniczeń globalnych `cumulative/4` i `disjunctive/2`.

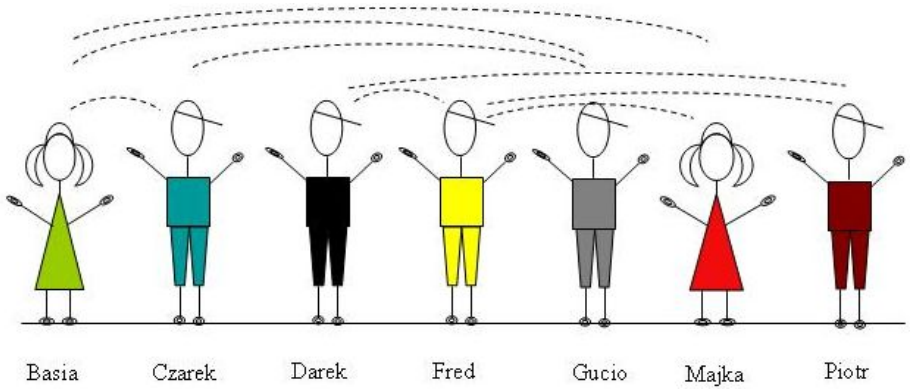
5.10.3 Optymalizacja a sprzeczność ograniczeń - fotografia

Często spotykamy problemy zawierające ograniczenia sprzeczne, np. dwóch wykładowców pragnie o tej samej porze w tej samej sali prowadzić różne zajęcia. *Ograniczeniami sprzecznymi* są takie, których równoczesne spełnienie nie jest możliwe. Rozpatrzmy prosty przykład sprzecznych ograniczeń, zainspirowany przykładem zamieszczonym w witrynie systemu `Mozart/Oz` [Mozart/Oz-10]:

Basia, Czarek, Darek, Fred, Gucio, Majka i Piotr ustawiają się w szereg do fotografii grupowej. Niektórzy z nich mają określone preferencje odnośnie do swych sąsiadów z lewa i prawa:

1. `Basia chce stać obok Gucia i Majki.`
2. `Czarek chce stać obok Basi i Gucia.`
3. `Fred chce stać obok Majki i Darka.`
4. `Piotr chce stać obok Freda i Darka,`

zob. rysunek 5.18. Łatwo wykazać, że spełnienie wszystkich preferencji nie jest możliwe. Można się w tym celu posłużyć programem `5_38_foto_1.ecl`:



Rysunek 5.18: Kandydaci do pamiątkowej fotografii i ich preferencje

```
/*1*/  :-lib(ic).
/*2*/  top :-
/*3*/  Osoby = [Basia, Czarek, Darek, Fred, Gucio, Majka, Piotr],
  % Znaczenie zmiennych jest następujące: Basia jest numerem
  % pozycji, na której (od lewej licząc) będzie "Basia",
  % Czarek jest numerem pozycji, na której będzie  "Czarek", itd.
/*4*/  Osoby :: 1..7,
/*5*/  alldifferent(Osoby),

/*6*/  obok(Basia,Gucio),
/*7*/  obok(Basia,Majka),
/*8*/  obok(Czarek,Basia),
/*9*/  obok(Czarek,Gucio),
/*10*/ obok(Fred,Majka),
/*11*/ obok(Fred,Darek),
/*12*/ obok(Piotr,Fred),
/*13*/ obok(Piotr,Darek),
/*14*/ search(Osoby,0,first_fail,indomain,complete,[]),
/*15*/ writeln("Osoby ":Osoby).

/*16*/ obok(X,Y):-
/*17*/     X #= Y + 1.
/*18*/ obok(X,Y):-
/*19*/     Y #= X + 1.
```

Uruchomienie tego programu generuje komunikat `No`. Jeżeli zaś zakomentować linie `6` i `11` (rezygnujemy z jednej preferencji dla Basi i z jednej dla Freda), to otrzymuje się dwa rozwiązania:

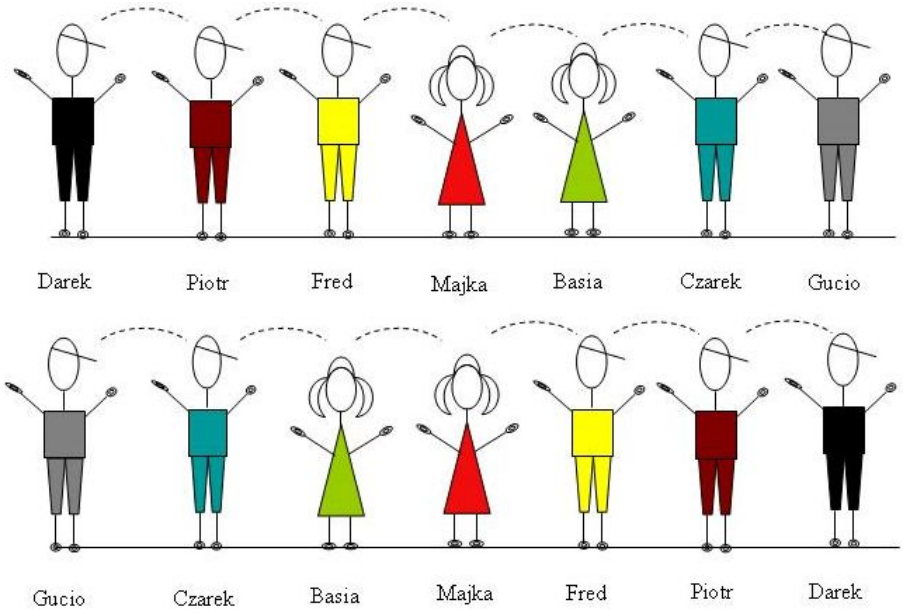
```
Osoby  : [5, 6, 1, 3, 7, 4, 2]
Osoby  : [3, 2, 7, 5, 1, 4, 6]
```

co odpowiada ustawieniom z rysunku 5.19:

```
Darek - Piotr - Fred - Majka - Basia - Czarek - Gucio
Gucio - Czarek - Basia - Majka - Fred - Piotr - Darek
```

Nasuwa się oczywiście pytanie: jeżeli nie można uczynić zadość wszystkim preferencjom, jaka jest maksymalna liczba preferencji, dla których to jest możliwe? Odpowiedzi na to udzieli nam program `5_39_foto_2.ecl`:

```
/*1*/  :-lib(ic).
/*2*/  :-lib(branch_and_bound).
```



Rysunek 5.19: Ustawienie bez ograniczeń 6 i 11

```
/*3*/  top :-
/*4*/  preferencje(Preferencje),
/*5*/  dim(Preferencje,[LiczbaPreferencji,2]),
/*5*/  dim(Pozycje, [7]),
/*6*/  Pozycje :: 1..7,
/*7*/  alldifferent(Pozycje),

/*8*/  length(Roznice, LiczbaPreferencji),
/*9*/  (for(I,1,LiczbaPreferencji),
/*10*/ fromto(Roznice,Wy,We,[]), % zbierz reifikacje
/*11*/ param(Preferencje,Pozycje)
/*12*/ do
/*13*/ P1 #= Pozycje[Preferencje[I,1]],
/*14*/ P2 #= Pozycje[Preferencje[I,2]],
/*15*/ Roznica #= P1-P2,
    % reifikacja warunku iż moduł zmiennej "Roznica" ma być równy 1:
/*16*/ Reif #= (Roznica #= 1 or Roznica #= -1),
/*17*/ Wy = [Reif|We]
/*18*/ ),
```

```
/*19*/ flatten_array(Preferencje, PreferencjeSplaszczone),
/*20*/ LiczbaPreferencjiSplaszczonych #= sum(PreferencjeSplaszczone),
/*21*/ Z :: 0..LiczbaPreferencjiSplaszczonych,
    % Z - liczba zaspokojonych preferencji:
/*22*/ Z #= sum(Roznice),
/*23*/ flatten_array(Pozycje, Zmienne),
/*24*/ ZNeg #= -Z,

/*25*/ minimize(search(Zmienne,0,first_fail,indomain,complete,[]),ZNeg),
/*26*/ writeln("Imiona: Basia, Czarek, Darek, Fred, Gucio, Majka, Piotr"),
/*27*/ writeln("Pozycje":Pozycje),
/*28*/ writeln("Liczba preferencji zasygnalizowanych": LiczbaPreferencji),
/*29*/ writeln("Liczba preferencji zaspokojonych":Z).

 % Komentarz do predykatu preferencje/1:
 %  [Basia, Czarek, Darek, Fred, Gucio, Majka, Piotr]
 % 1. Basia chce stać obok Gucia i Majki:
 %     1 obok 5, 1 obob 6
 % 2. Czarek chce stać obok Basi i Gucia:
 %     2 obok 1, 2 obok 5
 % 3. Fred chce stać obok Majki i Darka:
 %     4 obok 6, 4 obok 3
 % 4. Piotr chce stać obok Freda i Darka:
 %     7 obok 4, 7 obok 3

/*30*/ preferencje([](
               [](1,5),
               [](1,6),
               [](2,1),
               [](2,5),
               [](4,6),
               [](4,3),
               [](7,4),
               [](7,3))).
```

Program generuje następujący komunikat:

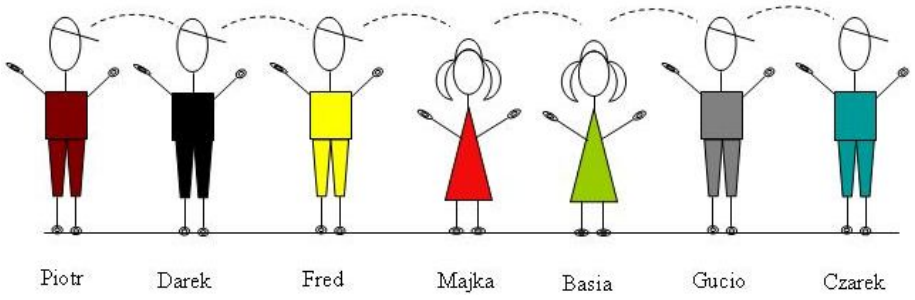
```
Found a solution with cost 0
Found a solution with cost -1
Found a solution with cost -2
Found a solution with cost -3
Found a solution with cost -4
Found a solution with cost -5
Found a solution with cost -6
Found no solution with cost -8.0 .. -7.0
```

```
Porządek imion (zmiennych)- komentarz autora:
          Basia, Czarek, Darek, Fred, Gucio, Majka, Piotr
Pozycje : [](5,     7,     2,     3,    6,     4,     1)
Liczba preferencji zasygnalizowanych : 8
Liczba preferencji zaspokojonych : 6
```

Rozwiązanie to odpowiada ustawieniu:

```
Piotr, Darek, Fred, Majka, Basia, Gucio, Czarek
```

Ustawienie to pokazano na rysunku 5.20. Jak widać, dwie preferencje nie zostały spełnione. Oczywiście, wyznaczone rozwiązanie optymalne nie jest unikatowe: program `5_38_photo_1.ecl`, dla którego linie `6` i `11` zostały usunięte, wygenerował również dwa rozwiązania z różnymi sześcioma spełnionymi preferencjami.



Rysunek 5.20: Ustawienie minimalizujące liczbę niespełnionych preferencji

5.11 Zadania

Pięć podręczników

Bookco Publishers rozważa publikację pięciu podręczników. Maksymalne zapotrzebowanie na podręczniki, zmienne i stałe koszty produkcji oraz ceny sprzedaży przedstawia tablica 5.9. I tak np. produkcja 2000 egzemplarzy podręcznika l daje przychód równy 2000*50 = 100000 JP, ale kosztuje 80000 + 25*2000 = 130000 JP. Zdolności produkcyjne Bookco to 10000 egzemplarzy podręczników. Napisz program wyjaśniający jak Bookco może

maksymalizować zysk z produkcji podręczników.

Podręcznik	1	2	3	4	5
Maksymalne zapotrzebowanie	5000	4000	3000	4000	3000
Cena sprzedaży (JP)	50	40	38	32	40
Zmienny koszt (JP)	25	20	15	18	22
Stałe koszty (w tysiącach JP)	80	50	60	30	40

Tablica 5.9: Dane dla pięciu podręczników

Zwiększamy fundusze emerytalne i produkujemy zieloną energię

Wzrastająca liczba bezdzietnych młodych małżeństw, kierująca swe poświęcenie i miłość ku psom, skłoniła Parlament Absurdolandii (zaniepokojony perspektywą malenia wpływów podatkowych, mogącą przyśpieszyć upadek piramidy finansowej państwowego systemu emerytalnego) do prac nad systemem generowania wpływów podatkowych przez rzeczone psy. Pośpiesznie utworzona Komisja Parlamentarna rozważała szereg możliwości, ale w swym *Raporcie Końcowym* przedstawiła szczegółowo tylko jeden z nich, opatrzony (ze względu na perspektywę uzyskania dla rozwiązania międzynarodowych patentów) angielskim akronimem `TET`, oznaczającym *Tail Energy Taps* i tłumaczącym się na *Pobór Energii Ogonów*. Istotą tego rozwiązania była konwersja naturalnego (i całkowicie dotąd bezużytecznego) merdania psiego ogona w użyteczną energię za pomocą małej, komputerowo-sterowanej i napędzanej psim ogonem prądnicy ładującej niewielki akumulatorek. Takie *TET*y zostaną przytwierdzone do psich zadków i zinterfejsowane z psimi ogonami. Wszyscy właściciele psów zostaną zobowiązani (na mocy specjalnej ustawy uchwalonej przez Parlament) do wyposażenia swych pupili w *TET*y i do cotygodniowego zgłaszania się z *TET*owymi akumulatorami odpowiednio w Dzielnicowych, Rejonowych lub Gromadzkich *Punktach Poboru Psich Energii*, gdzie energia z akumulatorów zostanie użyta do pompowania wody w kaskadzie szeregu zbiorników i w ten sposób zamieniona na grawitacyjną potencjalną energię dla przyszłych zastosowań.

Dla szczególnie leniwych psów *Raport Końcowy* przewidywał internetową wersję *TET*ów, kontrolujących aktywność psich ogonów i w okresach ich zaniku pobudzających ją za pomocą serii delikatnych mechanicznych i akustycznych sygnałów.

Raport Końcowy komisji parlamentarnej został entuzjastycznie przyjęty

przez większość parlamentarzystów. *Parlamentarzyści Zieloni* nie mogli nachwalić się genialnego odkrycia nowego źródła energii zrównoważonego rozwoju i jej przyjazności dla środowiska. *Parlamentarni Uszczęśliwiacze Ludzkości* różnej maści byli zachwyceni perspektywami tworzenia zupełnie nowych techno-zielonych miejsc pracy w przemyśle produkcji *TET*ów i w usługach konserwacji *TET*ów, jak również zarządczo-zielonych miejsc pracy w nowo-utworzonych Dzielnicowych, Rejonowych i Gromadzkich *Punktach Poboru Psich Energii* i *Ośrodkach Koordynacji Energii Psich Ogonów*, oraz w *Departamencie Psiej Energii* w *Ministerstwie Energetyki*. Przegłosowano również utworzenie *Specjalnych Oddziałów Psiej Policji* w *Ministerstwie Spraw Wewwnętrznych* w celu wyłapywania psów bez *TET*ów na zadkach. Obiekcje zgłaszane przez nieliczną *Opozycję*, zwracającą uwagę na koszty całego przedsięwzięcia i brak sensownego bilansu energetycznego, zostały wyszydzone przez *Rządzącą Koalicję* oraz *Media Głównego Nurtu*. *Ustawa o Psiej Energii* została szybko, bez zbędnej dyskusji i dzielenia włosa na czworo, uchwalona przez Parlament. Zostało to entuzjastycznie przyjęte przez szereg firm, dostrzegających możliwości pozbycia się mechanicznych i elektronicznych staroci, zalegających w magazynach, jak również możliwości pracy dla swych słabo zatrudnionych i silnie uzwiązkowionych załóg. Największą przeszkodą dla produkcji *TET*ów wydawał się być brak danych *kanigraficznych*[8]: nikt w Absurdolandii nie wiedział, ile dużych i ile małych psów tam żyje, ile z pośród nich to psy leniwe, a ile skłonnych do nieustannego merdania ogonem, lecz *Ustawa* wyróżniła wymienione cztery grupy, przypisując każdej z nich odmienny rodzaj *TET*a. Wszystkie zainteresowane firmy przestały się jednak zamartwiać brakiem danych o absurdolandzkiej *kanilacji*[9] i przystąpiły do maksymalizacji zysków z produkcji *TET*ów, wiedząc doskonale, że wszystko co wyprodukują zostanie przez rząd za pieniądze podatników wykupione po ustalonych przez rząd cenach.

Tak również postąpiła duża i znana *Junk Techno Company* (JTC), która szybko dostrzegła możliwość pozbycia się z zasłanych kurzem półek dwóch mechanicznych systemów `Type_A` i `Type_B`, których bardzo dużo kiedyś wyprodukowano licząc na lukratywne kontrakty wojskowe, oraz jednego systemu elektronicznego, `Inter`. Wymienione systemy można było niskim kosztem przekształcić w:

- `TET`y bezinternetowe dla małych piesków (potrzebne tylko systemy

[8] To jest psi odpowiednik danych "demograficznych".
[9] To jest psi odpowiednik "populacji".

Type_B),

- TETy internetowe dla małych piesków (potrzebne systemy Type_B oraz Inter),
- TETy bezinternetowe dla dużych psów (potrzebne systemy Type_A oraz Type_B)
- TETy internetowe dla dużych psów (potrzebne systemy Type_A, Type_B oraz Inter).

Szczęśliwym trafem okazało się, że koszty adaptacji i montażu TETów są głównymi składowymi kosztów, gdyż potrzebną elektronikę można otrzymać praktycznie za darmo na jednym z licznych *cmentarzy elektronicznych*.

Adaptacja systemów Type_A w JTC może się odbywać z wydajnością 60 sztuk dziennie, adaptacja systemów Type_B z wydajnością 50 sztuk dziennie, a adaptacja systemów Inter z wydajnością 30 sztuk dziennie. Montaż elektroniki dla każdego systemu może się odbywać z wydajnością 120 dziennie dla dowolnych systemów.

Na podstawie rządowo ustalonych cen JTC oceniło zysk ze sprzedaży jednego systemu Type_A jako równy 300 JP, jednego systemu Type_B - 500 JP i jednego systemu Inter - 300 JP. Napisz program obliczający, ile i jakich TET-ów powinno JTC wyprodukować, by zmaksymalizować swój zysk.

Kleje

Glueco produkuje trzy rodzaje kleju na dwóch różnych liniach produkcyjnych. Na każdej linii może pracować co najwyżej 7 robotników. Robotnicy zarabiają tygodniowo 500 JP na linii 1 i 900 JP na linii 2. Tygodniowy koszt utrzymania ruchu na linii 1 wynosi 1000 JP, a na linii 2 wynosi 2000 JP, W ciągu tygodnia każdy robotnik produkuje liczbę pojemników z klejem pokazaną w tablicy 5.10. Każdego tygodnia należy wyprodukować co najmniej 120 pojemników z klejem 1, co najmniej 150 pojemników z klejem 2 i co najmniej 200 pojemników z klejem 3. Napisz program wyznaczający taką obsadę linii produkcyjnych, która minimalizuje tygodniowy koszt produkcji przy zaspokojeniu tygodniowego zapotrzebowania.

Obciążenie maszyn

Pewien produkt można wykonać na dowolnej z czterech różnych maszyn. Każda maszyna ma stały koszt uzbrojenia, zmienny koszt produkcji na

Linia produkcyjna	Klej 1	Klej 2	Klej 3
Linia produkcyjna 1	20	30	40
Linia produkcyjna 2	50	35	45

Tablica 5.10: Tygodniowe wielkości produkcji klejów

jednostkę produktu i wydajność, przedstawione w tablicy 5.11. Należy wykonać 2000 sztuk wymienionego produktu. Napisz program wyznaczający obciążenie maszyn produkcją tak, by zminimalizować koszt całkowity.

Maszyna	Koszt stały (JP)	Zmienny koszt jednostkowy (JP)	Wydajność (liczba produktów)
1	1000	20	900
2	920	24	1000
3	800	16	1200
4	700	28	1600

Tablica 5.11: Dane czterech maszyn

Produkcja papieru

Papiernia produkuje papier w rolach o szerokości 105 cm. Jednakże hurtownie domagają się rol papieru o mniejszej szerokości, które należy wycinać z pierwotnych standardowych rol. Np. standardowa rola może być pocięta na dwie role o szerokości 35 cm i jedną rolę 30 cm. Papiernia otrzymała zamówienie pokazane w tablicy 5.12.

Szerokość cm	Liczba rol
25	100
30	125
35	80

Tablica 5.12: Zamówienie dla papierni

Napisz program wyznaczający minimalną liczbę rol papieru o szerokości 105 cm, którą należy wyprodukować dla realizacji zamówienie.

Pięć projektów

Pięć projektów o 3-letnim horyzoncie planowania zabiega o finansowanie. Tablica 5.13[10] przedstawia prognozowany zysk i roczne koszty każdego z projektów.

Projekt	Koszt (miliony JP rocznie)			Zysk (miliony JP)
	rok 1	rok 2	rok 3	
1	5	1	8	20
2	4	7	10	40
3	3	9	2	20
4	7	4	1	15
5	8	6	10	30
Dostępne finansowanie (miliony JP)	25	25	25	

Tablica 5.13: Roczne koszty projektów

Napisz program 1 dokonujący takiego wyboru finansowanych projektów, by maksymalizować całkowity zysk w horyzoncie 3-letnim. Zmodyfikuj program 1 tak, by uwzględnić dodatkowe ograniczenie: projekt 5 musi otrzymać finansowanie jeżeli projekt 1 albo 2 zostanie finansowany. Zmodyfikuj program 1 tak, by uwzględnić dodatkowe ograniczenie: z pośród projektów 2 i 3 tylko jeden może być finansowany.

Walczymy z napoleonofobią

Kontynuując pożyteczną misję walki z różnymi formami dyskryminacji i fobii, Parlament Absurdolandii zajął sie losem mniejszości obejmującej osoby uważające się za Napoleona Bonapartego, a pospolicie zwane *napoleonidami*. Asumpt do tego dała uchwała *Światowego Instytutu Zdrowia, Szczęścia i Pomyślności*, zgodnie z którą tzw. *napoleonidzenie* nie jest jednostką chorobową, lecz elementem społecznej dywersyfikacji, zasługującym na szacunek, godne finansowe wsparcie środkami publicznymi (*Różnorodność jest siłą!*) i promocję uroczystymi paradami w *Światowym Dniu Napoleonida*. Przedstawiciel napoleonidów wystapił na posiedzeniu *Komisji Parlamentarnej d/s Wykluczeń*, gdzie w gorzkich słowach przed-

[10] Zadanie pochodzi z podręcznika [Taha-08].

stawiał psychiczny dyskomfort absurdolandzkich napoleonidów, wynikający np. z faktu, że otoczenie zwraca się do nich per "Panie Kowalski" zamiast "Jego Cesarska Wielmożność", że różni nienawistnicy przezywają ich bonapartaczami, że do pracy muszą dojeżdżać tramwajem, autobusem lub pociągiem zamiast konno lub w karecie, w asyście generałów, adiutantów i małej grupy szwoleżerów, że wydatki związane z przestrzeganiem zasad ubioru i poruszania się, kompatybilnego z ich psychiką, prawie zawsze grubo przekraczają ich możliwości finansowe, że mało którego z nich stać na kosztowną operację plastyczno-kosmetyczną upodobniającą jego fizys do jego psychiki, a posiadanie zameczka z obsługą jest całkowicie poza ich zasięgiem. Opisał również tragiczne losy licznego grona bardzo społecznie aktywnych napoleonidów, którzy zostali ubezwłasnowolnieni i poddani przymusowemu leczeniu w Lecznicach Zamkniętych. *Komisja Parlamentarna d/s Wykluczeń* była głęboko wstrząśnięta wysłuchaną relacją i postanowiła natychmiast podjąć walkę z napoleonofobią i napoleonodyskryminacją, poczynając od szerokiego otwarcia drzwi owych Lecznic Zamkniętych dla wszelkich hospitalizowanych w nich napoleonidów i wprowadzenia kar więzienia za wszelką mowę nienawiści, natrząsanie się i naśmiewanie z napoleonidów, którzy zostali uznani za cenną mniejszość pozostającą pod specjalną opieką prawa. Znany lider partii *Ruch bez Ruchu* przysięgał - ze łzami w oczach - że na swoje listy wyborcze na nadchodzące wybory parlamentarne wprowadzi co najmniej dwóch napoleonidów. Aktywiści napoleonidzcy przedstawili listę 150 znanych zadeklarowanych absurdolandzkich napoleonidów (których liczba wzrosła poważnie w wyniku wzmiankowanej decyzji *Światowego Instytutu Zdrowia, Szczęścia i Pomyślności*). Rząd Absurdolandii szybko zmienił aktualny budżet tworząc *Fundusz Napoleonidzki* w wysokości 10 milionów JP, odebranych *Funduszowi Emerytalnemu*. Celem Funduszu Napoleoindzkiego jest udzielanie napoleonidom świadczeń pieniężnych zapewniających godziwe, niestresujące warunki bytowania. Równolegle aktywiści napoleonidzcy opracowali *Wskaźnik Szczęścia Napoleonida*, ujmujący w sposób ścisły świadczenia społeczne usuwające najbardziej dotkliwe cierpienia społeczności napoleonidów. Zdaniem aktywistów, *Komisja Parlamentarna d/s Wykluczeń* powinna dokonać podziału *Funduszu Napoleoindzkiego* w sposób optymalny, tak by ustrzec się przed zarzutem marnotrawienia środków publicznych. Za taki podział uznano zgodnie podział maksymalizujacy *Wskaźnik Szczęścia Napoleonida* dla rozpatrywanego zbioru 150 napoleonidów. Wskaźnik ten jest sumą iloczynów liczby beneficjentów korzystających z danego świadczenia i wartości szczęścia wynikającego z tego świadczenia. Maksy-

malizacji *Wskaźnika Szczęścia Napoleonida* trzeba było niestety dokonać uwzględniając szereg nie dających sie pokonać ograniczeń. I tak *Ministerstwo Technologii Medycznych* poinformowało, że rocznie można wykonać nie więcej niż 30 operacji korekty wzrostu i 15 operacji korekty twarzy, a *Minsterstwo Dziedzictwa Kulturowego* miało do dyspozycji tylko 6 zameczków z obsługą, mogących stanowić godziwe lokum dla napoleonidów. Również uznano niestety (ze względu na pogłębiający się kryzys), że napoleonidzi nie powinni równocześnie posiadać wierzchowców do jazdy konnej i karet z konnym zaprzęgiem. Jednostkowe koszty różnych zapomóg i wartości szczęścia z świadczeń dla pojedynczego beneficjenta przedstawia tablica 5.14.

Liczba beneficjentów	Rodzaj świadczenia	Wartość szczęścia dla beneficjenta	Koszt zapomogi dla beneficjenta
N1	Strój napoleoński	6	40
N2	Wierzchowiec	25	300
N3	Kareta z zaprzęgiem	50	1500
N4	Korekta twarzy	180	2000
N5	Korekta wzrostu	200	6500
N6	Zameczek	500	13000

Tablica 5.14: Dane dla alokacji świadczeń przysługujących napoleonidom

Wyznacz liczbę beneficjentów różnych świadczeń maksymalizującą całkowitą wartość szczęścia populacji napoleonidów przy uwzględnieniu wszystkich ograniczeń.

Produkcja samochodów

Znana firma *Clunker Motors Co* ma cztery fabryki samochodów. Każda z nich może produkować dowolny z trzech najbardziej popularnych modeli (Clunker SUV, Clunker Electric i Clunker Green). Podstawowe parametry ekonomiczne produkcji w każdej z fabryk przedstawia tablica 5.15.

Danymi tymi są koszty stałe funkcjonowania fabryki przez rok i koszty zmienne produkcji jednego samochodu. Produkcja musi spełniać następujące ograniczenia:
1. Każda fabryka może produkować tylko jeden z wymienionych trzech modeli.
2. Wszystkie operacje związane z produkcją dowolnego modelu muszą być zlokalizowane w jednej fabryce.

Fabryka	Koszty stałe	Koszty zmienne		
		SUV	Electric	Green
1	7 md JP	15000 JP	19000 JP	15000 JP
2	6 md JP	12000 JP	18000 JP	11000 JP
3	4 md JP	17000 JP	16000 JP	12000 JP
4	2 md JP	19000 JP	22000 JP	9000 JP

Tablica 5.15: Dane fabryk samochodów

3. Jeżeli fabryka nie produkuje samochodów (tzn. zostanie zamknięta), jej koszta stałe są pomijalnie małe.
4. Jeżeli fabryki 3 i 4 są czynne, to fabryka 1 również musi być czynna.

Clunker Motors Co musi produkować 600000 samochodów każdego typu rocznie. Napisz program, który określi rozkład produkcji minimalizujący całkowite koszty roczne przy zachowaniu wielkości produkcji.

Sieć barów szybkiej obsługi

Znana sieć barów szybkiej obsługi "Tasty Poison" podbija rynek Absurdolandii po błyskawicznym upadku państwowych punktów żywienia zbiorowego. Grupa specjalistów tej sieci analizuje sytuację w Starej Stolicy, koncentrując się na 6 możliwych lokalizacjach barów "Tasty Poison". Bary w wybranych lokalizacjach powinny obsługiwać 8 różnych dzielnic Starej Stolicy; zgodnie z zasadami działania sieci, każda dzielnica powinna być obsługiwana co najmniej przez jeden bar. Specjaliści dostarczyli - po ciężkiej pracy - wiarygodne oceny kosztów budowy każdego z barów wraz z propozycją dzielnic, które każdy bar mógłby obsługiwać, pokazane w tablicy 5.16. Możliwe przyporządkowania dzielnic barom zostały zaznaczone symbolami $\times$.

Napisz program minimalizujący koszt budowy sieci barów przy równoczesnej obsłudze wszystkich dzielnic.

Dzielnica	Bar szybkiej obsługi					
	1	2	3	4	5	6
1	×	×		×		
2		×		×		
3	×		×			
4	×			×		
5			×		×	
6			×		×	×
7		×		×		
8		×			×	×
Koszt budowy (miliony JP)	10	15	12	8	12	13

Tablica 5.16: Dane dla projektu sieci barów szybkiej obsługi

Skargi studentów [11]

Uniwersytet tworzy komisję dla rozpatrywania skarg studentów. Administracja pragnie, by w komisji była co najmniej jedna kobieta, co najmniej jeden mężczyzna, jeden student, jeden pracownik administracji i jeden przedstawiciel profesury. Dziesięć osób zostało wyłonionych do prac w wymienionej komisji, zob. tablica 5.17. Administracja pragnie utworzyć najmniejszą możliwie komisję z przedstawicielami wymienionych kategorii osób. Napisz program rozwiązujący ten problem.

Kategoria	Osoby
Kobiety	a,b,c,d,e
Mężczyźni	f,g,h,i,j
Studenci	a,b,c,j
Administratorzy	e,f
Profesura	d,g,h,i

Tablica 5.17: Kandydaci na członków komisji

[11] Przykład z [Taha-08].

Czterech programistów

Firma informatyczna dysponuje czterema programistami o jednakowych kwalifikacjach, którzy powinni napisać i przetestować siedem modułów programowych o różnej złożoności i różnej pracochłonności. Ocena pracochłonności opracowania modułów jest przedstawiona w tablicy 5.18.

Moduły programowe	1	2	3	4	5	6	7
Ocena pracochłonności (dni)	3	3	3	1	1	1	4

Tablica 5.18: Pracochłonność napisania i testowania modułów

Napisz program wyznaczający taki (niepodzielny) rozdział prac pomiędzy programistów, który umożliwi zakończenie napisania i testowania modułów w najkrótszym czasie.

Budowa pizzerii [12]

Dla danych przedstawionych w tablicy 5.19 napisz program wyznaczający ścieżkę krytyczną i czas jej trwania dla czynności związanych z zakładaniem pizzerii.

[12] Przykład z http://faculty.ksu.edu.sa/ialharkan/default.aspx

Numer czynności	Czynność	Poprzednik	Czas trwania w dniach
1	Opracowanie projektu		1
2	Wybór wykonawcy	1	5
3	Czyszczenie terenu	2	5
4	Doprowadzenie wody	3	5
5	Doprowadzenie elektryczności	3	12
6	Odprowadzenie ścieków	3	7
7	Wykonanie podłóg	4	7
8	Instalowanie wiatrołapu	7	1
9	Wykonanie ścian działowych	8	6
10	Kafelkowanie ścian	9	7
11	Wykonanie sufitów	5, 6, 7	4
12	Instalacja urządzeń kucharskich	10, 11	2
13	Instalowania oświetlenia i AC	12	3
14	Projekt frontu	3	5
15	Instalacja frontu	14	5
16	Lakierowanie 1	13, 15	2
17	Instalowanie mebli	6	4
18	Instalowanie internetu	13	1
19	Instalowanie okien	11	1
20	Lakierowanie 2	17, 19	1
21	Końcowa dekoracja	20	1
22	Projekt billboardu	3	2
23	Instalowanie billboardu	22	2
24	Akcja reklamowa		2
25	Test końcowy	18, 21, 23	1

Tablica 5.19: Czynności konieczne dla budowy pizzerii

Rozdział 6

CLP z predykatami globalnymi dla rozwiązań optymalnych

6.1 Uwagi wstępne

Niektóre *predykaty globalne* zostały już przedstawione i stosowane (w skromnym zakresie) w rozdziale 4 dla wyznaczania rozwiązań dopuszczalnych. Istnieje grupa "potężnych" predykatów globalnych, szczególnie użytecznych dla zadań optymalizacji. Są to predykaty `cumulative/4`, `cumulative/5`, `disjunctive/2`, `disjoint/1` i `cycle/3`. Wymienione predykaty mają przede wszystkim zastosowanie w trzech do tej pory nie omówionych zastosowaniach. Są nimi:

1. *Problemy harmonogramowania.* Harmonogramowanie (zwany w anglosaskiej literaturze *scheduling*) stanowi rozwinięcie idei szeregowania: można powiedzieć, że harmonogramowanie jest szeregowaniem z dodatkowym uwzględnieniem *ograniczenia resursów* potrzebnych dla realizacji szeregowanych operacji i obligatoryjnym wymogiem minimalizacji wskaźnika jakości, którym jest najczęściej całkowity czas wykonania szeregowanych operacji. Ograniczenie resursów można zdefiniować następująco:

```
Zapotrzebowanie_A_na wspólny_resurs + ... +
          Zapotrzebowanie_Z_na wspólny_resurs <=
                    Maksymalna_wielkość_wspólnego resursu
```

Ograniczenie to ma charakter *fundamentalny* i *uniwersalny*, prawie takie jak np. prawo ciążenia (*toutes proportion gardée*): bez uwzględnienia ograniczenia resursów nie można podejmować racjonalnych decyzji.

2. *Problemy upakowania*. Problem ten, zwany w anglosaskiej literaturze *bin packing*, polega na rozmieszczeniu (w zadanym "pojemniku" 1-wymiarowym, 2-wymiarowym lub 3-wymiarowym, największej liczby określonych elementów o tej samej wymiarowości. Elementarnym problemem upakowania był problem plecakowy omawiany w rozdziałach 5.6.3 i 5.6.6.

3. *Problemy marszrutyzacji*. Problem ten, zwany w anglosaskiej literaturze *vehicle routing problem*, polega na wyznaczeniu takich tras dla określonych środków transportu, by obsłużyć zbiór przestrzennie odległych klientów, spełnić ograniczenia dla stosowanych środków transportu i uzyskać minimum wskaźnika jakości, którym jest najczęściej całkowita długość trasy.

Wymienione predykaty znalazły również zastosowanie przy rozwiązywaniu niektórych już omówionych problemów, np. problemów szeregowania.

6.2 Predykat 'cumulative/4'

Jest ono ograniczeniem kumulatywnym *wspólnego resursu*: w dowolnej chwili całkowite obciążenie wspólnego resursu przez `n` zadań nie może przekroczyć maksymalnej wartości tego resursu, zwanej *limitem*. Ma ono postać:

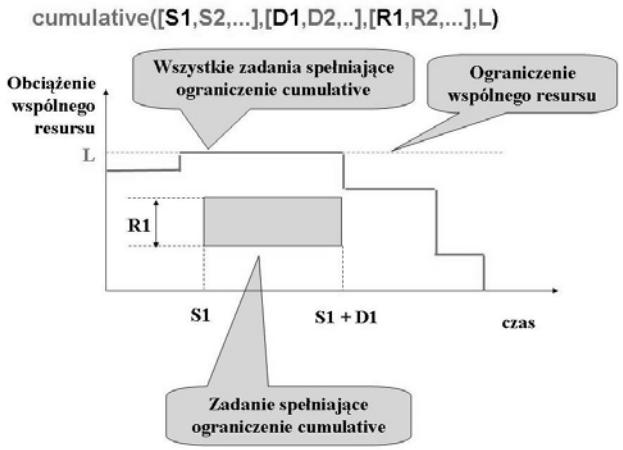
```
cumulative(+CzasyStartu, +CzasyTrwania, +Resursy, ++Limit)
```

gdzie:

```
CzasyStartu = [S1,...,Sn] jest listą czasów startu zadań,
CzasyTrwania = [D1,...,Dn] jest listą czasów trwania zadań,
Resursy = [R1,...,Rn] jest listą obciążeń wspólnego resursu
     przez kolejne zadania.
Dla wszystkich 1<=j<=n czasy końców zadań są równe:
      Sj+Dj=Kj
Limit  jest maksymalnym obciążeniem wspólnego resursu w dowolnej
```

```
chwili j, takiej że dla  a<=j<=b:
      max(Suma Rj) <= Limit
dla wszystkich j takich, że:  Sj <= i <= Sj+Dj-1,
 a = min(min(S1),...,min(Sn)) -
    najwcześniejszy możliwy czas startu zadań
 b = max((max(S1)+max(D1)),...,(max(Sn)+max(Dn))) -
    najpóźniejszy możliwy czas zakończenia zadań, gdzie:
 min(X) jest minimalną wartością w dziedzinie X
 max(X) jest maksymalną wartością w dziedzinie X
Koniec wszystkich zadań jest równy:
      Koniec =  max(Sj+Dj).
```

Podkreśla się, że elementem ograniczenia *cumulative/4* jest *jeden wspólny resurs* (np. całkowitą ilość dostępnych pieniędzy, całkowitą liczbę dostępnych maszyn określonego typu, całkowitą liczbę monterów o określonych umiejętnościach). Graficzną ilustrację zadania spełniającego to ograniczenie pokazano na rysunku 6.1.



Rysunek 6.1: Zadanie spełniające ograniczenie 'cumulative/4'

6.3 Kumulatywne harmonogramowanie 1

Rozwiążmy przykład z rozdziału 5.10.2 z zastosowaniem predykatu `cumulative/4`. Rozwiązaniem jest program `6_1_harm_cumu_1.ecl`:

```
/*1*/  :-lib(ic).
/*2*/   :- lib(ic_edge_finder3).
 /*3*/  :-lib(branch_and_bound).

/*4*/  top :-
/*5*/  harmonogram(_).

/*6*/  harmonogram([Z1,Z2,Z3,Z4,Koniec]):-
/*7*/  [Z1,Z2,Z3,Z4,Koniec] :: 0..15,
/*8*/  Z1 + 3 #=< Z2,
/*9*/  Z1 + 3 #=< Z3,
/*10*/ Z2 + 4 #=< Z4,
/*11*/ Z3 + 2 #=< Z4,
/*12*/ Z4 + 1 #= Koniec,

/*13*/ cumulative([Z2,Z3],[4,2],[1,1],1),
/*14*/ minimize(labeling([Z1,Z2,Z3,Z4,Koniec]),Koniec),
/*15*/ writeln("Z1 ":Z1),
/*16*/ writeln("Z2 ":Z2),
/*17*/ writeln("Z3 ":Z3),
/*18*/ writeln("Z4 ":Z4),
/*19*/ writeln("Koniec ":Koniec).
```

Program ten generuje komunikat:

```
Found a solution with cost 10
Found no solution with cost 8.0 .. 9.0
Z1 : 0
Z2 : 3
Z3 : 7
Z4 : 9
Koniec  : 10,
```

któremu odpowiada znany już wykres Gantta z rysunku 5.17a). Jeżeli zakomentować linię `13` i odkomentować linię `13a`, program wygeneruje komunikat:

```
Found a solution with cost 8
Z1  : 0
Z2  : 3
```

```
Z3  : 3
Z4  : 7
Koniec  : 8,
```

któremu odpowiada znany już wykres Gantta z rysunku 5.17b).

6.4 Kumulatywne harmonogramowanie 2

Rozpatrzmy bardziej złożony przykład kumulatywnego harmonogramowania.

Dla 7 zadań dane są czasy trwania i obciążenia wspólnego resursu:

Zadanie	Czas trwania	Obciążenie wspólnego resursu
1	16	2
2	6	9
3	13	3
4	7	7
5	5	10
6	18	1
7	4	11

W jakiej kolejności wykonać zadania, by całkowite obciążenie wspólnego resursu nie przekraczało nigdy 13, a czas wykonania wszystkich zadań był minimalny? Przykład ten rozwiązuje program `6_2_harm_cumu_2.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_edge_finder3).
 /*3*/  :- lib(branch_and_bound).

/*4*/top:-
/*5*/    LS = [S1,S2,S3,S4,S5,S6,S7],   %lista czasów startu zadań
/*6*/    LD = [16, 6,13, 7, 5,18, 4],   %lista czasów trwania zadań
/*7*/    LK = [K1,K2,K3,K4,K5,K6,K7],   %lista czasów końca zadań
/*8*/    LR = [2,9,3,7,10,1,11],   %lista obciążeń resursu zadaniami
/*9*/    LS :: 1..100,
/*10*/   Koniec :: 1..100,
/*11*/   LK :: 1..100,
/*12*/   Limit :: 1..13,

/*13*/   cumulative(LS,LD,LR,Limit),
```

```
/*14*/   K1 #= S1 + 16,
/*15*/   K2 #= S2 + 6,
/*16*/   K3 #= S3 + 13,
/*17*/   K4 #= S4 + 7,
/*18*/   K5 #= S5 + 5,
/*19*/   K6 #= S6 + 18,
/*20*/   K7 #= S7 + 4,

/*21*/   maxlist([K1,K2,K3,K4,K5,K6,K7],Koniec),
/*22*/   minimize(labeling([S1,S2,S3,S4,S5,S6,S7,
             K1,K2,K3,K4,K5,K6,K7]),Koniec),

/*23*/   write("LS = "), writeln(LS),
/*24*/   write("LK = "), writeln(LK),
/*25*/   write("Limit = "), writeln(Limit),
/*26*/   write("Koniec = "), writeln(Koniec).
```

Rozwiązaniem optymalnym jest:

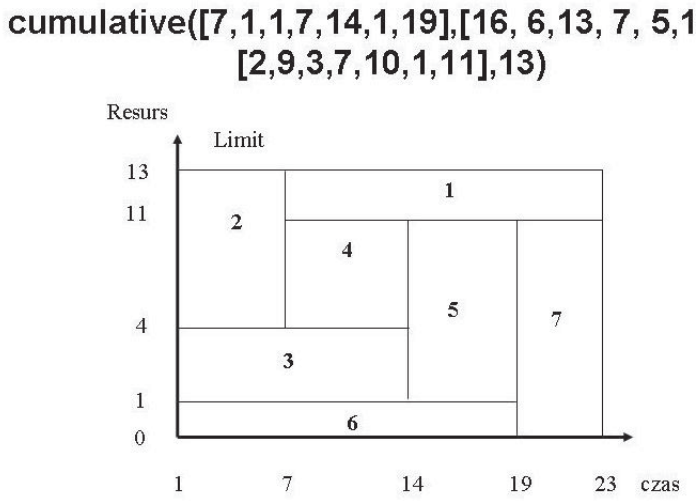
```
LS=[ 1,17,10,10, 5, 5,1]
LK=[17,23,23,17,10,23,5]
Limit=13
Koniec=23
```

Graficzną ilustrację rozwiązania pokazano na rysunku 6.2. Numery prostokątów są numerami zadań. Rysunek ten można interpretować nie tylko jako harmonogram. Można nań popatrzeć również jako na wynik rozkroju prostokąta o wymiarach 13×23 na mniejsze prostokąty odpowiadające zadaniom, lub jako na wynik upakowywania mniejszych prostokątów w większym prostokącie.

6.5 Kumulatywne szeregowanie

Predykat `cumulative/4` znajduje również zastosowanie dla zadań szeregowania. Ilustruje to przykład szeregowania zadań dla linii montażowej: należy wyznaczyć taką kolejność wykonywania zadań, która spełnia ograniczenia czasowe i kolejnościowe przy minimalizacji całkowitego czasu montażu. Zakłada się, że na linii montażowej są wykonywane zadania:

```
   A   B   C   D   E   F   G   H  I    J   K
o czasach trwania odpowiednio:
   45  11   9  50  15  12  12  12  12   8   9
```



Rysunek 6.2: Przykładu kumulatywnego harmonogramowania

przy zachowaniu następującej kolejności:

```
najpierw_potem(poprzednik, następnik)
najpierw_potem(A,B).
najpierw_potem(B,C).
najpierw_potem(C,F).
najpierw_potem(C,G).
najpierw_potem(F,J).
najpierw_potem(G,J).
najpierw_potem(J,K).
najpierw_potem(D,E).
najpierw_potem(E,H).
najpierw_potem(E,I).
najpierw_potem(H,J).
najpierw_potem(I,J).
```

Dla każdej pary (`poprzednik-następnik`), `poprzednik` nie może być wykonywany później aniżeli `następnik`. Program `6_3_szeregowanie_opty_cum.ecl` wyznacza (przy użyciu predykatu globalnego cumulative/4) taką sekwencję zadań, by całkowity czas montażu był minimalny:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(ic_edge_finder3).
 /*3*/ :- lib(branch_and_bound).

/*4*/     top:-
  %  czasy startów zadań:
/*5*/        LS=[As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
/*6*/        LS :: 0..250,
/*7*/        Koniec :: 0..250,

  %  ograniczenia kolejnościowe (zawierające czasy potrzebne na wykonanie zadań)
/*8*/       As + 45 #=< Bs,
/*9*/       Bs + 15 #=< Cs,
/*10*/      Cs + 9 #=< Fs,
/*11*/      Cs + 9 #=< Gs,
/*12*/      Fs + 12 #=< Js,
/*13*/      Gs + 12 #=< Js,
/*14*/      Js + 8 #=< Ks,
/*15*/      Ds + 50 #=< Es,
/*16*/      Es + 15 #=< Hs,
/*17*/      Es + 15 #=< Is,
/*18*/      Hs + 12 #=< Js,
/*19*/      Is + 12 #=< Js,
/*20*/      Ks + 9 #= Koniec,

/*21*/     cumulative([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
               [45,15,9,50,15,12,12,12,12,8,9],
                       [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],1),

/*22*/     minimize(labeling([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks]),Koniec),

/*23*/     writeln([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks,Koniec]).
```

Rozwiązanie jest (jak zwykle przy optymalizacji w ECL^iPS^e) pojedyncze i ma postać:

```
Found a solution with cost 199
Found no solution with cost 98.0 .. 198.0
[0, 45, 60, 69, 119, 134, 146, 158, 170, 182, 190, 199]
```

Intuicja podpowiada nam jednak, że tych rozwiązań optymalnych powinno być

więcej. Zweryfikujemy to za pomocą programu `6_4_szeregowanie_opty_cum_wszystkie.ecl`, w którym ustalono zmienną `Koniec` na optymalnej wartosci 199:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(ic_edge_finder3).

/*3*/     top:-
/*4*/        assert(licznik(0)),
  %  czasy startów zadań:
/*5*/        LS=[As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
/*6*/        LS :: 0..250,

  %  ograniczenia kolejnościowe (zawierające czasy potrzebne na wykonanie zadań)
/*7*/        As + 45 #=< Bs,
/*8*/        Bs + 15 #=< Cs,
/*9*/        Cs + 9 #=< Fs,
/*10*/       Cs + 9 #=< Gs,
/*11*/       Fs + 12 #=< Js,
/*12*/       Gs + 12 #=< Js,
/*13*/       Js + 8 #=< Ks,
/*14*/       Ds + 50 #=< Es,
/*15*/       Es + 15 #=< Hs,
/*16*/       Es + 15 #=< Is,
/*17*/       Hs + 12 #=< Js,
/*18*/       Is + 12 #=< Js,
/*19*/       Ks + 9 #= Koniec,
/*20*/       Koniec is 199,

/*21*/       cumulative([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
              [45,15,9,50,15,12,12,12,12,8,9],
                      [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],1),

/*22*/       labeling([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks]),

/*23*/       licz,
/*24*/       licznik(Liczba),

/*25*/       write("Rozwiazanie optymalne "), write(Liczba),write(":"),nl,
/*26*/       writeln([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks,Koniec]),
/*27*/       fail.

/*28*/    top:-
/*29*/       write("To wszystkie rozwiazania optymalne.").

/*30*/    licz:-
/*31*/       retract(licznik(Stary)),
/*32*/       Nowy is Stary + 1,
```

```
/*33*/      assert(licznik(Nowy)).
```

Nasza intuicja była poprawna: program generuje 504 rozwiązania optymalne. Tylko niektóre z nich zostaną pokazane poniżej:

```
Rozwiazanie optymalne 1:
[0, 45, 60, 69, 119, 134, 146, 158, 170, 182, 190, 199]
.............................................................
Rozwiazanie optymalne 61:
[0, 45, 110, 60, 119, 134, 146, 158, 170, 182, 190, 199]
.............................................................
Rozwiazanie  optymalne 141:
[0, 95, 110, 45, 119, 134, 146, 158, 170, 182, 190, 199]
.............................................................
Rozwiazanie optymalne  504:
[89, 134, 149, 0, 50, 170, 158, 77, 65, 182, 190, 199]
To wszystkie rozwiazania optymalne.
```

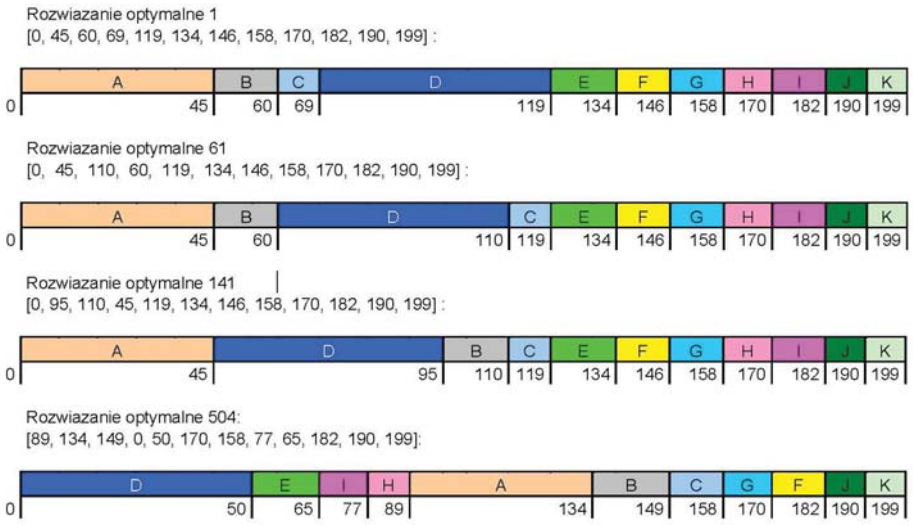
Ocena tych rozwiązań wymaga przedstawienia ich w postaci wykresów Gantta z rysunku 6.3. Rysunek ten umożliwia również zrozumienie poprawnej interpretacji wyników numerycznych generowanych przez program. Obfitość rozwiązań optymalnych dla szeregowania zadań ma pewną zaletę, której omówienie nie ma już charakteru elementarnego: umożliwia rozwiązanie zadania *równoważenia linii montażowej*, patrz rozdział 6.10.

6.6 Predykat 'disjunctive/2'

Predykat ten jest postaci:

```
disjunctive(+Czasy_Startu, +Czasy_Trwania)
```

Predykat ten jest spełniony, jeżeli zadania o `Czasach_Startu` i `Czasach_Trwania` nie nakładają się na siebie, jak to pokazano na rysunku 6.4. Liczby czasów startu i czasów trwania muszą oczywiście być jednakowe.



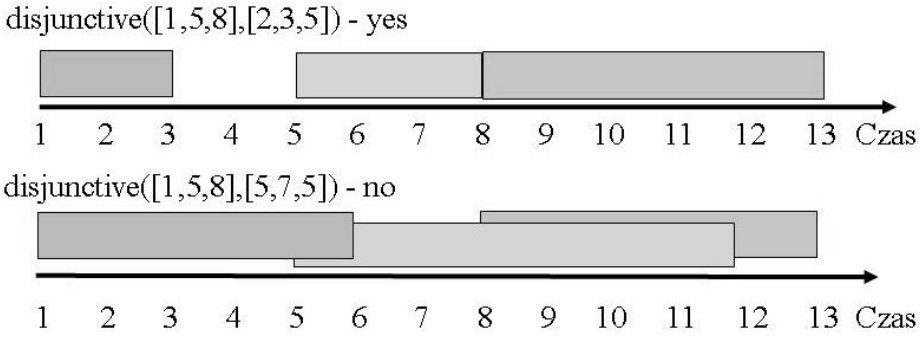
Rysunek 6.3: Wykresy Gantta wybranych optymalnych sekwencji zadań dla linii montażowej

W odróżnieniu od `cumulative/4`, które ogranicza zadania w przestrzeni resursu (na wykresach - w pionie), ograniczenie `disjunctive/2` ogranicza zadania w czasie (na wykresach - w poziomie). Przykład poniższy pokazuje jednak, że w pewnych sytuacjach ograniczenia te są wymienne.

6.7 Dyzjunktywne szeregowanie

Predykat `disjunctive/2` jest w zasadzie przeznaczony dla dyzjunktywnego szeregowania. Poniżej (`6_5_szeregowanie_opty_dis.ecl`) zostanie przedstawione rozwiązanie problemu szeregowania z rozdziału 6.5 przy użyciu tego predykatu:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(ic_edge_finder3).
/*3*/ :- lib(branch_and_bound).

/*4*/ top:-
  %  czasy startów zadań:
```



Rysunek 6.4: Właściwości ograniczenia 'disjunctive/2'

```
/*5*/        LS=[As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
/*6*/        LS :: 0..250,
/*7*/        Koniec :: 0..250,

  %  ograniczenia kolejnościowe (zawierające czasy potrzebne dla wykonanie zadań)
/*8*/       As + 45 #=< Bs,
/*9*/       Bs + 15 #=< Cs,
/*10*/      Cs + 9 #=< Fs,
/*11*/      Cs + 9 #=< Gs,
/*12*/      Fs + 12 #=< Js,
/*13*/      Gs + 12 #=< Js,
/*14*/      Js + 8 #=< Ks,
/*15*/      Ds + 50 #=< Es,
/*16*/      Es + 15 #=< Hs,
/*17*/      Es + 15 #=< Is,
/*18*/      Hs + 12 #=< Js,
/*19*/      Is + 12 #=< Js,
/*20*/      Ks + 9 #= Koniec,

/*21*/     disjunctive([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
                      [45,15,9,50,15,12,12,12,12,8,9]),

/*22*/     minimize(labeling([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks]),Koniec),

/*23*/     writeln([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks,Koniec]).
```

Rozwiązanie jest identyczne jak dla programu `6_3_szeregowanie_opty_cum.ecl`.

Wielokrotne rozwiązania optymalne można wyznaczyć za pomocą programu

6_6_szeregowanie_opty_dis_wszystkie.ecl:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(ic_edge_finder3).

/*3*/    top:-
/*4*/       assert(licznik(0)),
  %  czasy startów zadań:
/*5*/       LS=[As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
/*6*/       LS :: 0..250,

  %  ograniczenia kolejnościowe (zawierające czasy potrzebne dla wykonanie zadań)
/*7*/       As + 45 #=< Bs,
/*8*/       Bs + 15 #=< Cs,
/*9*/       Cs + 9 #=< Fs,
/*10*/      Cs + 9 #=< Gs,
/*11*/      Fs + 12 #=< Js,
/*12*/      Gs + 12 #=< Js,
/*13*/      Js + 8 #=< Ks,
/*14*/      Ds + 50 #=< Es,
/*15*/      Es + 15 #=< Hs,
/*16*/      Es + 15 #=< Is,
/*17*/      Hs + 12 #=< Js,
/*18*/      Is + 12 #=< Js,
/*19*/      Ks + 9 #= Koniec,
/*20*/      Koniec is 199,

/*21*/      disjunctive([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks],
                  [45,15,9,50,15,12,12,12,12,8,9]),

/*22*/     labeling([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks]),

/*23*/     licz,
/*24*/     licznik(Liczba),

/*25*/     write("Rozwiazanie optymalne "), write(Liczba),write(":"),nl,
/*26*/     writeln([As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is,Js,Ks,Koniec]),
/*27*/     fail.

/*28*/  top:-
/*29*/     write("To wszystkie rozwiazania optymalne.").

/*30*/  licz:-
/*31*/     retract(licznik(Stary)),
/*32*/     Nowy is Stary  1, +
/*33*/     assert(licznik(Nowy)).
```

Rozwiązania w liczbie 504 są identyczne jak dla programu 6_4_szeregowanie_

opty_cum_wszystkie.ecl. Wybrane rozwiązania można obejrzeć na rysunku 6.3.

6.8 Dyzjunktywne harmonogramowanie

Predykat disjunctive/2 znajduje również zastosowanie dla problemów harmonogramowania. Rozwiążmy przykład z rozdziału 5.10.2 z zastosowaniem predykatu disjunctive/2. Rozwiązaniem tym jest program 6_7_harm_dis.ecl:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_edge_finder3).
/*3*/  :- lib(branch_and_bound).

/*4*/  top :-
/*5*/       harmonogram(_).

/*6*/  harmonogram([Z1,Z2,Z3,Z4,Koniec]):-
/*7*/      [Z1,Z2,Z3,Z4,Koniec] :: 0..15,
/*8*/      Z1 + 3 #=< Z2,
/*9*/      Z1 + 3 #=< Z3,
/*10*/     Z2 + 4 #=< Z4,
/*11*/     Z3 + 2 #=< Z4,
/*12*/     Z4 + 1 #= Koniec,
/*13*/     disjunctive([Z2,Z3],[4,2]),
/*14*/     minimize(labeling([Z1,Z2,Z3,Z4,Koniec]),Koniec),
/*15*/     writeln("Z1 ":Z1),
/*16*/     writeln("Z2 ":Z2),
/*17*/     writeln("Z3 ":Z3),
/*18*/     writeln("Z4 ":Z4),
/*19*/     writeln("Koniec ":Koniec).
```

Program ten generuje komunikat:

```
Found a solution with cost 10
Found no solution with cost 8.0 .. 9.0
Z1 : 0
Z2 : 3
Z3 : 7
```

```
Z4 : 9
Koniec  : 10
```

któremu odpowiada znany już wykres Gantta z rysunku 5.10 a).

6.9 Predykat 'disjoint2/1'

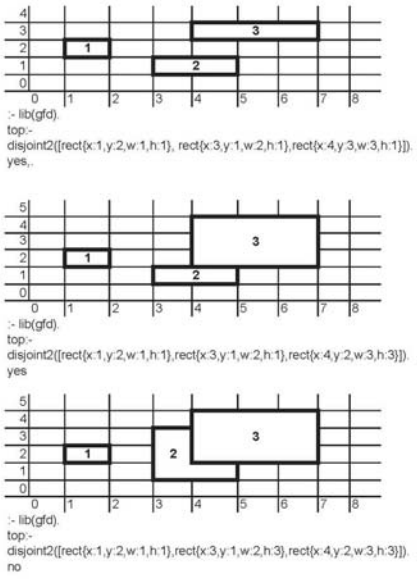
Predykat ten, będący uogólnieniem predykatu `disjunctive/2`, ogranicza położenie i rozmiary 2-wymiarowych prostokatów tak, by nie nachodziły na siebie. Prostokąty są opisywane strukturami postaci:

```
rect{x:X,y:Y,w:W,h:H}
```

zawierającymi pola:

- stała x: Współrzędna x lewego boku prostokąta, równa zmiennej X;
- stała y: Współrzędna y dolnej krawędzi prostokąta, równa zmiennej Y;
- stała w: Długość prostokąta równa zmiennej W;
- stała h: Wysokość prostokąta równa zmiennej H.

Stosowanie predykatu `disjunctive/2` jest możliwe po załadowaniu biblioteki `gfd`. Właściwości predykatu ilustruje program `6_8_disjoint.ecl`:



Rysunek 6.5: Trzy przykłady zastosowań predykatu 'disjoint2/1'

```
:- lib(gfd).
top_1:-
  disjoint2([rect{x:1,y:2,w:1,h:1},rect{x:3,y:1,w:2,h:1},
           rect{x:4,y:3,w:3,h:1}]).
top_2:-
   disjoint2([rect{x:1,y:2,w:1,h:1},rect{x:3,y:1,w:2,h:1},
           rect{x:4,y:2,w:3,h:3}]).
top_3:-
   disjoint2([rect{x:1,y:2,w:1,h:1},rect{x:3,y:1,w:2,h:3},
           rect{x:4,y:2,w:3,h:3}]).
```

Rozwiązaniem dla `top_1` i `top_2` jest `yes`, rozwiązaniem dla `top_3` jest `no`. Rysunek 6.5 przedstawia prostokąty występujące w programie.

6.10 Równoważenie linii montażowej

Równoważenie linii montażowej polega na takim przydzieleniu zadań poszczególnym stanowiskom linii, by przy zachowaniu zadanej kolejności zadań i ich czasu trwania, na każdym stanowisku była (w idealnej sytuacji) wykonywana praca o jednakowym czasie trwania. Czas ten nazywa się *czasem cyklu* linii montażowej.

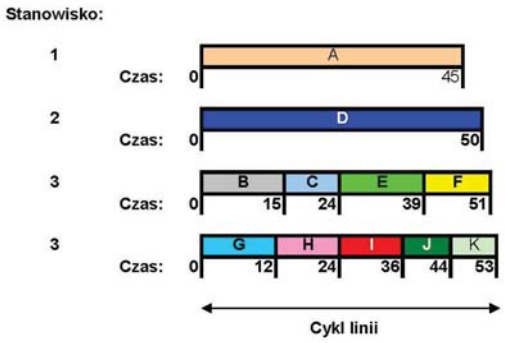
W niezrównoważonej linii montażowej mniej obciążone stanowiska muszą czekać bezczynnie na zakończenie zadań na stanowiskach bardziej obciążonych.

Ponieważ tego typu idealne rozważanie nie zawsze jest możliwe, dąży się do równoważenia przybliżonego, dla którego całkowity czas montażu jest minimalny.

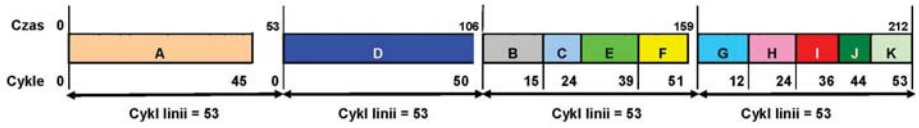
Rozpatrzmy rozwiązanie optymalne `141` szeregowania dla przykładu z rozdziału 6.5, pokazane na rysunku 6.3. Korzystając z tego rozwiązania można wstępnie dokonać rozdziału zadań pomiędzy cztery stanowiska montażu, co zrobiono w programie `6_9_disjoint_balance.ecl`:

```
/*1*/ :-lib(gfd).
/*2*/ :- lib(branch_and_bound).

/*3*/ top:-
  %czasy startów zadań:
/*4*/      LSZ=[ A, B, C, D, E, F, G, H, I, J, K],
  %stanowiska, na których będą wykonywane zadania A,B,...
/*5*/      Lst=[Ast,Bst,Cst,Dst,Est,Fst,Gst,Hst,Ist,Jst,Kst],
  %czasy trwania zadań
/*6*/      LD=[Ad,Bd,Cd,Dd,Ed,Fd,Gd,Hd,Id,Jd,Kd],
  %cza7sy zakończenia zadań
/*4*/      LE=[EA,EB,EC,ED,EE,EF,EG,EH,EI,EJ,EK],
  %obciążenia resource'ów
/*8*/      LR=[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

  %deklaracja domen
/*9*/      LSZ #:: 0..100,
/*10*/     Lst #:: 1..4,
/*11*/     LE #:: 0..100,
/*12*/     Limit #:: 1..4,

  %deklaracja czasów trwania zadań
/*13*/     Ad is 45, Bd is 15, Cd is 9, Dd is 50, Ed is 15, Fd is 12,
                     Gd is 12, Hd is 12, Id is 12, Jd is 8, Kd is 9,
  %przydział  zadań do czterech stanowisk
/*14*/     Ast is 1,  Bst is 3, Cst is 3, Dst is 2, Est is 3, Fst is 3,
                     Gst is 4, Hst is 4, Ist is 4, Jst is 4, Kst is 4,
```

```
  % ograniczenia czasów końców zadań:
/*15*/     gfd_gac: (EA #>= A + Ad),
/*16*/     gfd_gac: (EB #>= B + Bd),
/*17*/     gfd_gac: (EC #>= C + Cd),
/*18*/     gfd_gac: (ED #>= D + Dd),
/*19*/     gfd_gac: (EE #>= E + Ed),
/*20*/     gfd_gac: (EF #>= F + Fd),
/*21*/     gfd_gac: (EG #>= G + Gd),
/*22*/     gfd_gac: (EH #>= H + Hd),
/*23*/     gfd_gac: (EI #>= I + Id),
/*24*/     gfd_gac: (EJ #>= J + Jd),
/*25*/     gfd_gac: (EK #>= K + Kd),

  %przydziały zadań do stanowisk:
/*26*/     disjoint2([
             rect{x:A,y:Ast,w:Ad,h:1},rect{x:B,y:Bst,w:Bd,h:1},
             rect{x:C,y:Cst,w:Cd,h:1},rect{x:D,y:Dst,w:Dd,h:1},
             rect{x:E,y:Est,w:Ed,h:1},rect{x:F,y:Fst,w:Fd,h:1},
             rect{x:G,y:Gst,w:Gd,h:1},rect{x:H,y:Hst,w:Hd,h:1},
             rect{x:I,y:Ist,w:Id,h:1}, rect{x:J,y:Jst,w:Jd,h:1},
             rect{x:K,y:Kst,w:Kd,h:1}]),

  %obliczenie wskaźnika jakości (czasu montażu)
/*27*/     cumulative(LSZ,LD,LR,Limit),

/*28*/     append(LSZ,LE,LSZ_E),
/*29*/     append(LSZ_E, Lst,LSZ_E_Lst),

  %minimalizacja wskaźnika jakości
/*30*/    gfd_gac: (max(LE, M)),
/*31*/    bb_min(labeling(LSZ_E_Lst), M, bb_options with
              [strategy:continue,from:0,to:100]),
/*32*/    write("Czasy końców zadań = "),write(LE),nl,
/*33*/    write("Minimalny czas cyklu = "),write(M),nl,nl,

/*34*/write("Stanowisko dla A: "),write(Ast),write(", Start A = "),write(A),nl,
/*35*/write("Stanowisko dla B: "),write(Bst),write(", Start B = "),write(B),nl,
/*36*/write("Stanowisko dla C: "),write(Cst),write(", Start C = "),write(C),nl,
/*37*/write("Stanowisko dla D: "),write(Dst),write(", Start D = "),write(D),nl,
/*38*/write("Stanowisko dla E: "),write(Est),write(", Start E = "),write(E),nl,
/*39*/write("Stanowisko dla F: "),write(Fst),write(", Start F = "),write(F),nl,
/*40*/write("Stanowisko dla G: "),write(Gst),write(", Start G = "),write(G),nl,
/*41*/write("Stanowisko dla H: "),write(Hst),write(", Start H = "),write(H),nl,
/*42*/write("Stanowisko dla I: "),write(Ist),write(", Start I = "),write(I),nl,
/*43*/write("Stanowisko dla J: "),write(Jst),write(", Start J = "),write(J),nl,
/*43*/write("Stanowisko dla K: "),write(Kst),write(", Start K = "),write(K),nl.
```



Rysunek 6.6: Wynik spełnienia 'cumulative/4' dla równoważenia linii



Rysunek 6.7: Wykres Gantta dla zrównoważonej linii montażowej

Rozwiązanie ma postać pokazaną na rysunkach 6.6 i 6.7, odpowiadających komunikatowi:

```
Found a solution with cost 53
Found no solution with cost 50.0 .. 52.0
Czasy końców zadań = [45, 15, 24, 50, 39, 51, 12, 24, 36, 44, 53]
Minimalny czas cyklu = 53

Stanowisko dla A: 1 Start A = 0
Stanowisko dla B: 3 Start B = 0
Stanowisko dla C: 3 Start C = 15
Stanowisko dla D: 2 Start D = 0
Stanowisko dla E: 3 Start E = 24
Stanowisko dla F: 3 Start F = 39
Stanowisko dla G: 4 Start G = 0
```

```
  Stanowisko dla H: 4 Start H = 12
  Stanowisko dla I: 4 Start I = 24
  Stanowisko dla J: 4 Start J = 36
  Stanowisko dla K: 4 Start K = 44

  Delayed goals:
gfd : gfd_do_propagate(gfd_prob(nvars(35)))
  Yes (1258.02s cpu)
```

6.11 Czytamy gazety 1

Przejdziemy obecnie do bardziej złożonego przykładu harmonogramowania:

Czterech studentów nauk politycznych najlepszego uniwersytetu Absurdolandii, Alek, Bolek, Czarek i Darek, wynajmuje wspólne mieszkanie. Codziennie rano otrzymują cztery najbardziej opiniotwórcze dzienniki głównego nurtu: "Przeinaczenia i Manipulacje" (PiM), "Absurdzielec Codzienny" (AC), "Poranny Brainwashing" (PB) i "Der Rynsztok" (DR). Każdy student czyta zawsze wszystkie gazety w określonej - przez siebie ulubionej - kolejności i przez określony czas, zgodnie z tablicą 6.1[1].

Studenci	Czytanie 1	Czytanie 2	Czytanie 3	Czytanie 4
Arek budzi się o 8:30, ma kolejność czytania: i czas czytania:	PiM 60 min	AC 30 min	PB 2 min	DR 5 min
Bolek budzi się o 8:45, ma kolejność czytania: i czas czytania:	AC 75 min	PB 3 min	PiM 25 min	DR 10 min
Czarek budzi się o 8:45, ma kolejność czytania: i czas czytania:	PB 5 min	AC 15 min	PiM 10 min	DR 30 min
Darek budzi się o 9:45 ma kolejność czytania: i czas czytania:	DR 90 min	PiM 5 min	AC 5 min	PB 5 min

Tablica 6.1: Kolejność i czasy czytania gazet

[1] Temat zainspirowany raportem [Duncan-90].

Jeżeli Alek, Bolek, Czarek i Darek rozpoczynają lekturę gazet natychmiast po obudzeniu, o której godzinie mogą wszyscy pójść najwcześniej na uczelnię? Rozwiązanie zadania jest dane programem `6_10_gazety_1.ecl`, w którym:
1) ograniczenia rozdzielności czytania gazet są deklarowane za pomocą *disjunctive/2*;
2) ograniczenia liczby gazet czytanych równocześnie przez studenta są deklarowane za pomocą *cumulative/4*;
3) czasy są minutami po godzinie 08:00.

Program `6_10_gazety_1.ecl` ma postać:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(ic_edge_finder3).
 /*3*/ :- lib(branch_and_bound).

/*4*/ top:-
/*5*/  A=[APiM,AAC,APB,ADR], % czasy początków czytania przez Alka
/*6*/  B=[BAC,BPB,BPiM,BDR], % czasy początków czytania przez Bolka
/*7*/  C=[CPB,CAC,CPiM,CDR], % czasy początków czytania przez Czarka
/*8*/  D=[DDR,DPiM,DAC,DPB], % czasy początków czytania przez Darka
/*9*/  Koniec=[A_koniec,B_koniec,C_koniec,D_koniec],
/*10*/              % czasy końców czytania przez studentów

/*11*/  A :: 30..360,
/*12*/  B :: 45..360,
/*13*/  C :: 45..360,
/*14*/  D :: 105..360,
/*15*/  Koniec :: 90..360,
/*16*/  Koniec_koncow :: 90..360,

/*17*/ APiM#>=30,
/*18*/ AAC#>=APiM+60,        % ograniczenia kolejnościowe dla Alka
/*19*/ APB#>=AAC+30,
/*20*/ ADR#>=APB+2,
/*21*/ A_koniec#>=ADR+5,

/*22*/ BAC#>=45,
/*23*/ BPB#>=BAC+75,         % ograniczenia kolejnościowe dla Bolka
/*24*/ BPiM#>=BPB+3,
/*25*/ BDR#>=BPiM+25,
/*26*/ B_koniec#>=BDR+10,

/*27*/ CPB#>=45,
/*28*/ CAC#>=CPB+5,          % ograniczenia kolejnościowe dla Czarka
/*29*/ CPiM#>=CAC+15,
/*30*/ CDR#>=CPiM+10,
```

```
/*31*/ C_koniec#>=CDR+30,

/*32*/ DDR#>=105,
/*33*/ DPiM#>=DDR+90,         % ograniczenia kolejnościowe dla Darka
/*34*/ DAC#>=DPiM+5,
/*35*/ DPB#>=DAC+5,
/*36*/ D_koniec#>=DPB+5,

    % Ograniczenia liczby studentów czytających gazetę.
    % Jedna gazeta może być czytana tylko przez jednego studenta.
    % czytanie "Przeinaczeń i Manipulacji":
/*37*/ disjunctive([APiM,BPiM,CPiM,DPiM],[60,25,10,5]),
    % czytanie "Absurdzielca Codziennego":
/*38*/ disjunctive([AAC,BAC,CAC,DAC],[30,75,15,5]),
    % czytanie "Porannego Brainwashingu":
/*39*/ disjunctive([APB,BPB,CPB,DPB],[2,3,5,9]),
    % czytanie "Der Rynsztoku":
/*40*/ disjunctive([ADR,BDR,CDR,DDR],[5,10,30,90]),

    % Ograniczenia liczby gazet czytanych przez studenta.
    % Każdy student może czytać tylko jedną gazetę.
    % czyta Alek:
/*41*/ cumulative([APiM,AAC,APB,ADR],[60,30,2,5],[1,1,1,1],1),
    % czyta Bolek:
/*42*/ cumulative([BAC,BPB,BPiM,BDR],[75,3,25,10],[1,1,1,1],1),
    % czyta Czarek:
/*43*/ cumulative([CPB,CAC,CPiM,CDR],[5,15,10,30],[1,1,1,1],1),
    % czyta Darek:
/*44*/ cumulative([DDR,DPiM,DAC,DPB],[90,5,5,5],[1,1,1,1],1),

/*45*/ maxlist(Koniec,Koniec_koncow),
/*46*/ minimize(labeling([APiM,AAC,APB,ADR,BAC,BPB,BPiM,BDR,
          CPB,CAC,CPiM,CDR,DDR,DPiM,DAC,DPB,A_koniec,B_koniec,
              C_koniec,D_koniec]), Koniec_koncow),nl,

/*47*/ write("A = "),write(A),nl,
/*48*/ write("B = "),write(B),nl,
/*49*/ write("C = "),write(C),nl,
/*50*/ write("D = "),write(D),nl,

/*51*/ przedstaw_harmonogram([APiM,AAC,APB,ADR,BAC,BPB,BPiM,BDR,
         CPB,CAC,CPiM,CDR,DDR,DPiM,DAC,DPB],
         ["Alek","Przeinaczenia i Manipulacje",60,
         "Alek","Absurdzielec Codzienny",30,
         "Alek","Poranny Brainwashing",2,
         "Alek","Der Rynsztok",5,
         "Bolek","Absurdzielec Codzienny",75,
         "Bolek","Poranny Brainwashing",3,
         "Bolek","Przeinaczenia i Manipulacje",25,
```

```
        "Bolek","Der Rynsztok",10,
        "Czarek","Poranny Brainwashing",5,
        "Czarek","Absurdzielec Codzienny",15,
        "Czarek","Przeinaczenia i Manipulacje",10,
        "Czarek","Der Rynsztok",10,
        "Darek","Der Rynsztok",90,
        "Darek","Przeinaczenia i Manipulacje",5,
        "Darek","Absurdzielec Codzienny",5,
        "Darek","Poranny Brainwashing",5]).

/*52*/ przedstaw_harmonogram([],[]):-nl.
/*53*/ przedstaw_harmonogram([H1|T1],[H21,H22,H23|T2]) :-
/*54*/    przelicz_czas(H1, FG, FM),
/*55*/    HH is H1+H23,
/*56*/    przelicz_czas(HH, TG, TM),nl,
/*57*/    write(H21),write(" ----> "),write(H22),write(" od "),
/*58*/    write(FG),write(":"),write(FM),
/*59*/    write(" do "),write(TG),write(":"),write(TM),
/*60*/ przedstaw_harmonogram(T1,T2).

/*61*/ przelicz_czas(Czas,Godziny,Minuty) :-
/*62*/    div(Czas, 60, G),
/*63*/    Godziny is G + 8,
/*64*/    mod(Czas,60,Minuty).
```

Program generuje następujące rozwiązanie:

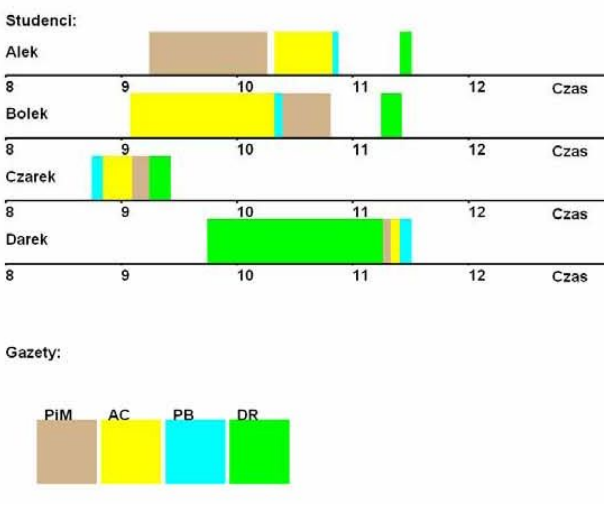
```
Found a solution with cost 338
Found a solution with cost 263
Found a solution with cost 262
Found a solution with cost 257
Found a solution with cost 240
Found a solution with cost 238
Found a solution with cost 235
Found a solution with cost 210

A = [75, 140, 170, 195]
B = [65, 140, 143, 200]
C = [45, 50, 65, 75]
D = [105, 195, 200, 205]


Alek ----> Przeinaczenia i Manipulacje od 9:15 do 10:15
Alek ----> Absurdzielec Codzienny od 10:20 do 10:50
Alek ----> Poranny Brainwashing od 10:50 do 10:52
Alek ----> Der Rynsztok od 11:15 do 11:20
```

```
Bolek ----> Absurdzielec Codzienny od 9:5 do 10:20
Bolek ----> Poranny Brainwashing od 10:20 do 10:23
Bolek ----> Przeinaczenia i Manipulacje od 10:23 do 10:48
Bolek ----> Der Rynsztok od 11:20 do 11:30
Czarek ----> Poranny Brainwashing od 8:45 do 8:50
Czarek ----> Absurdzielec Codzienny od 8:50 do 9:5
Czarek ----> Przeinaczenia i Manipulacje od 9:5 do 9:15
Czarek ----> Der Rynsztok od 9:15 do 9:25
Darek ----> Der Rynsztok od 9:45 do 11:15
Darek ----> Przeinaczenia i Manipulacje od 11:15 do 11:20
Darek ----> Absurdzielec Codzienny od 11:20 do 11:25
Darek ----> Poranny Brainwashing od 11:25 do 11:30
```

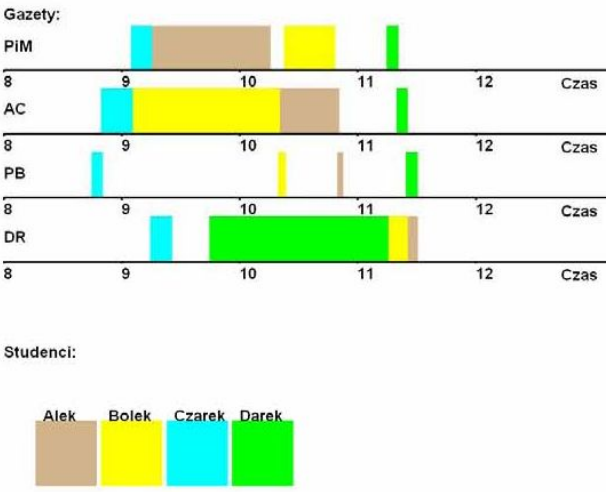
Jak widać, czytanie gazet skończy się o godzinie 11:25 i wtedy wszyscy czterej studenci mogą pójść na uczelnię.

Powyższy wydruk jest słabo czytelny. Poprawienie czytelności rozwiązania uzyskuje się tworząc dwa wykresy Gantta; wykres przedstawiający czynności studentów w czasie (zob. rysunek 6.8) oraz wykres przedstawiający losy gazet w czasie (zob. rysunek 6.9). Sposób kolorowania *klocków* wykresu Gantta dla gazet jest odmienny niż dla studentów. Gdybyśmy zastosowali w wykresie Gantta dla gazet sposób kolorowania *klocków* z wykresu dla studentów, wszystkie *klocki* w danym wierszu (dla tej samej gazety) byłyby tego samego koloru, co uczyniłoby wykres słabo czytelnym.

Z wykresów tych widać, że optymalny ciąg czytań nie jest jednoznaczny: np. czytania PB przez Alka, PiM-u przez Bolka i DR przez Czarka mogą rozpocząć się nieco później bez wpływu na koniec wszystkich czytań.



Rysunek 6.8: Wykres Gantta dla studentów



Rysunek 6.9: Wykres Gantta dla gazet

6.12 Czytamy gazety 2

Czytanie gazet można również rozwiązać korzystając wyłącznie z predykatu globalnego `cumulative/4`. Odpowiedni program `6_11_gazety_2.ecl` ma postać:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(ic_edge_finder3).
 /*3*/ :- lib(branch_and_bound).

/*4*/ top:-
/*5*/  A=[APiM,AAC,APB,ADR], % czasy początków czytania przez Alka
/*6*/  B=[BAC,BPB,BPiM,BDR], % czasy początków czytania przez Bolka
/*7*/  C=[CPB,CAC,CPiM,CDR], % czasy początków czytania przez Czarka
/*8*/  D=[DDR,DPiM,DAC,DPB], % czasy początków czytania przez Darka
/*9*/  Koniec=[A_koniec,B_koniec,C_koniec,D_koniec],
              % czasy końców czytania przez studentów

/*10*/  A :: 30..360,
/*11*/  B :: 45..360,
/*12*/  C :: 45..360,
/*13*/  D :: 105..360,
/*14*/  Koniec :: 90..360,
/*15*/  Koniec_koncow :: 90..360,

/*16*/ APiM#>=30,
/*17*/ AAC#>=APiM+60,    % ograniczenia kolejnościowe dla Alka
/*18*/ APB#>=AAC+30,
/*19*/ ADR#>=APB+2,
/*20*/ A_koniec#>=ADR+5,

/*21*/ BAC#>=45,
/*22*/ BPB#>=BAC+75,    % ograniczenia kolejnościowe dla Bolka
/*23*/ BPiM#>=BPB+3,
/*24*/ BDR#>=BPiM+25,
/*25*/ B_koniec#>=BDR+10,

/*26*/ CPB#>=45,
/*27*/ CAC#>=CPB+5,    % ograniczenia kolejnościowe dla Czarka
/*28*/ CPiM#>=CAC+15,
/*29*/ CDR#>=CPiM+10,
/*30*/ C_koniec#>=CDR+30,

/*31*/ DDR#>=105,
/*32*/ DPiM#>=DDR+90,    % ograniczenia kolejnościowe dla Darka
/*33*/ DAC#>=DPiM+5,
/*34*/ DPB#>=DAC+5,
/*35*/ D_koniec#>=DPB+5,
```

```
    % Ograniczenia liczby studentów czytających gazetę.
    % Jedna gazeta może być czytana tylko przez jednego studenta:
    % czytanie "Przeinaczeń i Manipulacji":
/*36*/ cumulative([APiM,BPiM,CPiM,DPiM],[60,25,10,5],[1,1,1,1],1),
    % czytanie "Absurdzielca Codziennego":
/*37*/ cumulative([AAC,BAC,CAC,DAC],[30,75,15,5],[1,1,1,1],1),
    % czytanie "Porannego Brainwashingu":
/*38*/ cumulative([APB,BPB,CPB,DPB],[2,3,5,5],[1,1,1,1],1),
    % czytanie "Der Rynsztoku":
/*39*/ cumulative([ADR,BDR,CDR,DDR],[5,10,30,90],[1,1,1,1],1),

    % Ograniczenia liczby gazet czytanych przez studenta.
    % Każdy student może czytać tylko jedną gazetę:
    % czyta Alek:
/*40*/ cumulative([APiM,AAC,APB,ADR],[60,30,2,5],[1,1,1,1],1),
    % czyta Bolek:
/*41*/ cumulative([BAC,BPB,BPiM,BDR],[75,3,25,10],[1,1,1,1],1),
    % czyta Czarek:
/*42*/ cumulative([CPB,CAC,CPiM,CDR],[5,15,10,30],[1,1,1,1],1),
    % czyta Darek:
/*43*/ cumulative([DDR,DPiM,DAC,DPB],[90,5,5,5],[1,1,1,1],1),

/*44*/ maxlist(Koniec,Koniec_koncow),
/*45*/ minimize(labeling([APiM,AAC,APB,ADR,BAC,BPB,BPiM,BDR,
          CPB,CAC,CPiM,CDR,DDR,DPiM,DAC,DPB,A_koniec,B_koniec,
              C_koniec,D_koniec]), Koniec_koncow),nl,

/*46*/ write("A = "),write(A),nl,
/*47*/ write("B = "),write(B),nl,
/*48*/ write("C = "),write(C),nl,
/*49*/ write("D = "),write(D),nl,

/*50*/ przedstaw_harmonogram([APiM,AAC,APB,ADR,BAC,BPB,BPiM,BDR,
         CPB,CAC,CPiM,CDR,DDR,DPiM,DAC,DPB],
         ["Alek","Przeinaczenia i Manipulacje",60,
         "Alek","Absurdzielec Codzienny",30,
         "Alek","Poranny Brainwashing",2,
         "Alek","Der Rynsztok",5,
         "Bolek","Absurdzielec Codzienny",75,
         "Bolek","Poranny Brainwashing",3,
         "Bolek","Przeinaczenia i Manipulacje",25,
         "Bolek","Der Rynsztok",10,
         "Czarek","Poranny Brainwashing",5,
         "Czarek","Absurdzielec Codzienny",15,
         "Czarek","Przeinaczenia i Manipulacje",10,
         "Czarek","Der Rynsztok",10,
         "Darek","Der Rynsztok",90,
         "Darek","Przeinaczenia i Manipulacje",5,
         "Darek","Absurdzielec Codzienny",5,
```

```
        "Darek","Poranny Brainwashing",5]).

/*51*/ przedstaw_harmonogram([],[]):-nl.
/*52*/ przedstaw_harmonogram([H1|T1],[H21,H22,H23|T2]) :-
/*53*/    przelicz_czas(H1, FG, FM),
/*54*/    HH is H1+H23,
/*55*/    przelicz_czas(HH, TG, TM),nl,
/*56*/    write(H21),write(" ----> "),write(H22),write(" od "),
/*57*/    write(FG),write(":"),write(FM),
/*58*/    write(" do "),write(TG),write(":"),write(TM),
/*59*/ przedstaw_harmonogram(T1,T2).

    % Czasy sa minutami po godzinie 08:00

/*60*/ przelicz_czas(Czas,Godziny,Minuty) :-
/*61*/    div(Czas, 60, G),
/*62*/    Godziny is G + 8,
/*63*/    mod(Czas,60,Minuty).
```

Program generuje komunikat:

```
Found a solution with cost 338
Found a solution with cost 263
Found a solution with cost 262
Found a solution with cost 257
Found a solution with cost 240
Found a solution with cost 238
Found a solution with cost 235
Found a solution with cost 210

A = [75, 140, 170, 195]
B = [65, 140, 143, 200]
C = [45, 50, 65, 75]
D = [105, 195, 200, 205]

Alek ----> Przeinaczenia i Manipulacje od 9:15 do 10:15
Alek ----> Absurdzielec Codzienny od 10:20 do 10:50
Alek ----> Poranny Brainwashing od 10:50 do 10:52
Alek ----> Der Rynsztok od 11:15 do 11:20
Bolek ----> Absurdzielec Codzienny od 9:5 do 10:20
Bolek ----> Poranny Brainwashing od 10:20 do 10:23
Bolek ----> Przeinaczenia i Manipulacje od 10:23 do 10:48
Bolek ----> Der Rynsztok od 11:20 do 11:30
Czarek ----> Poranny Brainwashing od 8:45 do 8:50
```

```
Czarek ----> Absurdzielec Codzienny od 8:50 do 9:5
Czarek ----> Przeinaczenia i Manipulacje od 9:5 do 9:15
Czarek ----> Der Rynsztok od 9:15 do 9:25
Darek ----> Der Rynsztok od 9:45 do 11:15
Darek ----> Przeinaczenia i Manipulacje od 11:15 do 11:20
Darek ----> Absurdzielec Codzienny od 11:20 do 11:25
Darek ----> Poranny Brainwashing od 11:25 do 11:30
```

Jest ono tym razem nieco inne od poprzedniego, co wynika z istnienia kilku rozwiązań optymalnych: tym razem (dla tego samego czasu końca `11:30`), Alek i Bolek zamienili się kolejnością czytania dziennika "Der Rynsztok".

6.13 Czytamy gazety 3

Sposób wprowadzania danych stosowany w poprzednich dwóch przykładach był bardzo rozwlekły i uniemożliwiał szybką zmianę danych.
W programie `6_12_gazety_3.ecl` zrobiono to bardziej profesjonalnie, niestety kosztem przejrzystości i zrozumiałości programu. Program korzysta z następujących ważnych "prywatnych" predykatów:

```
dane(Dane)
Dane = [student(Imie,Czas_wstawania,Gazety)|Reszta]
Gazety = [Gazeta,Czas_czytania|Reszta] =
               = [Nazwa_gazety1, Czas_czytania1,
                       Nazwa_gazety2, Czas_czytania2, itd]
ogranicz(Dane,Czytania,Czasy_startu,Koniec)
Czytania = [czytanie(Imie,Gazeta,Start,Czas_czytania)|Reszta]
ograniczenie_pojedynczej_gazety(Gazeta,Czytania)
ogranicz_z_aku(Dane,Aku_czytania,Czytania,
              Aku_czasow_startu,Czasy_startu,Koniec)
tworz_czytania(Gazety,Imie,Czas_wstawania,Aku_Czytania,Czytania,
              Aku_czasow_startu,Czasy_startu)
kompletuj_gazety(Czytania,Gazeta,Aku_czasow_startu,Czasy_startu,
              Aku_czasow_czytania,Czasy_czytania)
ukonkretniaj(Czasy_startu,Koniec) - prywatna odmiana labelingu
```

Termin `aku` oznacza `akumulator` odpowiedniej listy.

Program `6_12_gazety_3.ecl` jest następujący:

```
/*1*/ :- lib(ic).
/*2*/ :- lib(ic_edge_finder3).
```

```
 /*3*/ :- lib(branch_and_bound).

/*4*/ top:-
/*5*/  dane(Dane),
/*6*/  ogranicz(Dane,Czytania,Czasy_startu,Koniec),
/*7*/  minimize(ukonkretniaj(Czasy_startu,Koniec),Koniec),
/*8*/  przedstaw_harmonogram(Czytania).

 % Dane problemu. Czas wstawania jest liczbą całkowitą,
 %  będącą liczbą minut, które upłynęły po godzinie 08:00:
/*9*/ dane([
  student("Alek",30,["PiM",60,"AC",30,"PB",2,"DR",5]),
  student("Bolek",45,["AC",75,"PB",3,"PiM",25,"DR",10]),
  student("Czarek",45,["PB",5,"AC",15,"PiM",10,"DR",30]),
  student("Darek",104,["DR",90,"PiM",1,"AC",1,"PB",1])]).

/*10*/ogranicz(Dane,Czytania,Czasy_startu,Koniec):-
/*11*/ Koniec :: 0..363,
/*12*/ ogranicz_z_aku(Dane,[],Czytania,[],Czasy_startu,Koniec),
/*13*/ ograniczenie_pojedynczej_gazety("PiM",Czytania),
/*14*/ ograniczenie_pojedynczej_gazety("AC",Czytania),
/*15*/ ograniczenie_pojedynczej_gazety("PB",Czytania),
/*16*/ ograniczenie_pojedynczej_gazety("DR",Czytania).

 % Twórz serię "czytań" dla każdego studenta:
/*17*/ogranicz_z_aku([],Czytania,Czytania,Czasy_startu,
                 Czasy_startu,_).
/*18*/ogranicz_z_aku([student(Imie,Czas_wstawania,
         [Gazeta,Czas_czytania|Reszta])|Pozostali],Aku_czytania,
                 Czytania,Aku_czasow_startu,Czasy_startu,Koniec):-

 % Twórz czytanie dla "Imie":
/*19*/ Start :: Czas_wstawania..363,
/*20*/ Czas_kolejny is Czas_wstawania + Czas_czytania,
/*21*/ tworz_czytania(Reszta,Imie,Czas_kolejny,
         [czytanie(Imie,Gazeta,Start,Czas_czytania)|Aku_czytania],
               Czyt1,[Start|Aku_czasow_startu],Starty1),
/*22*/ Czyt1 = [czytanie(Imie,_,S1,D1)|_],

 % Uczyń "Koniec" niemniejszym niż czas zakończenia ostatniego
 % czytania przez danego studenta. W ten sposób "Koniec" będzie
 % ostatecznie równy czasowi końca ostatniego czytania
 % najpóźniejszego studenta:
/*23*/ Koniec #>= S1+D1,
/*24*/ ogranicz_z_aku(Pozostali,Czyt1,Czytania,Starty1,
           Czasy_startu,Koniec).

 % Twórz "czytania" dla studenta "Imie" z równoczesnym
 % ograniczaniem kolejności czytania gazet:
```

```
/*25*/tworz_czytania([],_,_,Czytania,Czytania,Czasy_startu,
          Czasy_startu).
/*26*/tworz_czytania([Gazeta,Czas_czytania|Reszta],Imie,
          Czas_kolejny,Aku_czytania,Czyt,Aku_czasow_startu,Starty):-
/*27*/ Start :: Czas_kolejny..363,
/*28*/ Aku_czytania = [czytanie(Imie,_,S1,D1)|_],
/*29*/ Start #>= S1+D1,
/*30*/ KolejnyKoniec is Czas_kolejny+Czas_czytania,
/*31*/ tworz_czytania(Reszta,Imie,KolejnyKoniec,
         [czytanie(Imie,Gazeta,Start,Czas_czytania)|Aku_czytania],
              Czyt,[Start|Aku_czasow_startu],Starty).

 % Kompletuj wszystkie "czytania" gazety "Gazeta"
 % i wymuś nienakładanie się "czytań" za pomocą
 % predykatu globalnego "cumulative/4":
/*32*/ograniczenie_pojedynczej_gazety(Gazeta,Czytania):-
/*33*/ kompletuj_gazety(Czytania,Gazeta,[],Starty,[],
          Czasy_czytania),
/*34*/ cumulative(Starty,Czasy_czytania,[1,1,1,1],1).

 % Kompletuj czasy startu i czasy czytania
 % wszystkich czytań dla "Gazeta":
/*35*/kompletuj_gazety([],_,S,S,D,D).
/*36*/kompletuj_gazety([czytanie(_,Gazeta,S,D)|Reszta],
          Gazeta,S0,S1,D0,D1):- !,
/*37*/ kompletuj_gazety(Reszta,Gazeta,[S|S0],S1,[D|D0],D1).
/*38*/kompletuj_gazety([_|Reszta],Gazeta,S0,S1,D0,D1):-
/*39*/ kompletuj_gazety(Reszta,Gazeta,S0,S1,D0,D1).

 % Generuj rozwiązanie korzystając z heurystyki "first_fail"
/*40*/ukonkretniaj([],Koniec):-
/*41*/ indomain(Koniec,min),
/*42*/ przelicz_czas(Koniec,Godziny,Minuty),
/*43*/ write("Znaleziono czytania dla konca:  "),write(Godziny),
          write(":"),write(Minuty),nl.

/*44*/ukonkretniaj(Czasy_startu,Koniec):-
/*45*/ wybierz(Zmienna,Czasy_startu,Reszta,0,first_fail),
/*46*/ indomain(Zmienna),
/*47*/ ukonkretniaj(Reszta,Koniec).

 % Predykat "prywatny" wybierz(Zmienna,Lista,Reszta,Flaga,
 % Heurystyka) korzysta ze standardowego predykatu "delete/5".
 % Jeżeli "Lista" nie jest pusta, możliwe są nawroty.
/*48*/wybierz(_, [], _, _, _):-
/*49*/ !,
/*50*/ fail.
/*51*/wybierz(Zmienna, Lista, Reszta, Flaga, Heurystyka):-
/*52*/ delete(Zmienna,Lista,Reszta,Flaga, Heurystyka).
```

```
 % Przedstaw harmonogram:
/*53*/przedstaw_harmonogram([]).
/*54*/przedstaw_harmonogram([czytanie(Imie,Gazeta,Start,
         Czas_czytania)|Reszta]):-
/*55*/ przelicz_czas(Start, FG, FM),
/*56*/ HH is Start+Czas_czytania,
/*57*/ przelicz_czas(HH, TH, TM),
/*58*/ write(Imie),write(" ----> "),write(Gazeta),write(" od "),
         write(FG),write(":"),write(FM),
/*59*/ write(" do "),write(TH),write(":"),write(TM),nl,
/*60*/ przedstaw_harmonogram(Reszta).

 % Czasy są minutami po godzinie 08:00:
/*61*/przelicz_czas(Czas,Godziny,Minuty) :-
/*62*/ div(Czas, 60, G),
/*63*/ Godziny is G + 8,
/*64*/ mod(Czas,60,Minuty).
```

Program generuje komunikat:

```
Znaleziono czytania dla konca:  11:59
Found a solution with cost 239
Znaleziono czytania dla konca:  11:30
Found a solution with cost 210
Found no solution with cost 197.0 .. 209.0

Darek ----> PB od 11:17 do 11:18
Darek ----> AC od 11:16 do 11:17
Darek ----> PiM od 11:15 do 11:16
Darek ----> DR od 9:45 do 11:15
Czarek ----> DR od 9:15 do 9:45
Czarek ----> PiM od 9:5 do 9:15
Czarek ----> AC od 8:50 do 9:5
Czarek ----> PB od 8:45 do 8:50
Bolek ----> DR od 11:15 do 11:25
Bolek ----> PiM od 10:23 do 10:48
Bolek ----> PB od 10:20 do 10:23
Bolek ----> AC od 9:5 do 10:20
Alek ----> DR od 11:25 do 11:30
Alek ----> PB od 10:50 do 10:52
Alek ----> AC od 10:20 do 10:50
```

```
Alek ----> PiM od 9:15 do 10:15
```

Rozwiązanie różni się od rozwiązań dla przykładów `6_10_gazety_1.ecl` i `6_11_gazety_2.ecl`, minimalny czas czytania jest jednak taki sam.

6.14 Montujemy rowery

Ronald Graham z `AT&T` napisał ciekawy esej o fikcyjnej fabryce rowerów *ACME*, zob. [Graham-83], zamieszczony w książce [Steen-83]. Esej ten, nieco rozszerzony i zmodyfikowany dla potrzeb książki, będzie podstawą pouczającego programu harmonogramowania. Pouczającego w tym sensie, że pokazującego jak bardzo jest zawodna intuicja nawet w przypadku prostego zadania harmonogramowania. Zmodyfikowana i rozszerzona wersja eseju jest następująca:
"W *ACME* od sześciu miesięcy nie wykonuje się planu i dlatego zaczęto zmieniać kierownictwo. Nowy kierownik działu montażu ma zaradzić temu złemu stanowi rzeczy. Zdaje sobie sprawę z tego, że jest to jego wielka szansa zwrócenia na siebie uwagi dyrekcji, więc od pierwszego dnia zakasuje rękawy i próbuje dowiedzieć się o wszystkim, co się na montażu dzieje.
Po pierwsze, dowiaduje się, że proces montażu dzieli się zwyczajowo na pewną liczbę mniejszych zadań montażowych:

```
A - przygotowanie ramy i montaż przednich widełek i błotników
B - montaż i scentrowanie przedniego koła
C - montaż i scentrowanie tylnego koła
D - założenie przerzutki na ramie
E - montaż  wolnego koła
F - przymocowanie koła łańcuchowego do korby pedału
G - montaż korby pedału i koła łańcuchowego na ramie
H - montaż prawego pedału
I - montaż lewego pedału
J - montaż kierownicy, siodełka i hamulców[2]
```

Nowy kierownik dowiaduje się także, że jego (dopiero co zwolniony) poprzednik zebrał całe stosy danych o tym, jak długo (średnio, w minutach) prace te są

[2]Parafrazując Benedykta Chmielowskiego (1700-1763), który w pierwszej polskiej encyklopedii "Nowe Ateny" napisał przy haśle *"Koń"* tylko jedno słynne zdanie: *"Koń jaki jest każdy widzi."*, można napisać *"Rower jaki jest każdy widzi."*, rezygnując z pięknego rysunku roweru firmy *ACME* zamieszczonego w eseju [Graham-83].

wykonywane przez doświadczonego montera; dane te zostały zebrane w następującej tablicy:

Czynność	A	B	C	D	E	F	G	H	I	J
Czas trwania	7	7	7	2	2	2	2	8	8	18

Łączny czas trwania wszystkich czynności montażowych dla pojedynczego roweru wynosi więc 63 minuty. Ze względu na ograniczenia przestrzenne i wyposażeniowe tworzy się z zatrudnionych w fabryce 20 monterów 10 zespołów dwuosobowych, z których każdy montuje jeden rower. Kierownik dokonuje szybkiego przeliczenia zakładając, że skoro montaż jednego roweru zajmuje 63 minuty, zatem dwuosobowy zespół *powinien* dać temu radę w 31,5 minuty. Oznacza to, że podczas 8-godzinnej zmiany każdy zespół mógłby zmontować 15.23 rowerów, czyli 10 zespołów zmontuje dziennie 152 rowery. Nowy kierownik zaczyna sobie już wyobrażać swój awans.

Ten entuzjazm zredukuje się znacznie, gdy przekona się, że rowerów nie można składać w sposób przypadkowy: pewne czynności muszą być wykonane przed innymi. Np. przed montażem pedałów korba pedałów musi być zamontowana do ramy. Po długich naradach z doświadczonymi monterami zostaje opracowany następujący zestaw *ograniczeń kolejnościowych*:

A, B, C, D, E	należy wykonać przed czynnością	J
D, E, A	należy wykonać przed czynnością	C
D	należy wykonać przed czynnościami	E, F
E, F, G	należy wykonać przed czynnościami	H, I
F	należy wykonać przed czynnością	G
A	należy wykonać przed czynnością	B

Ponadto należy jeszcze uwzględnić dwie zasady "zajętości" forsowane przez właścicieli fabryki:

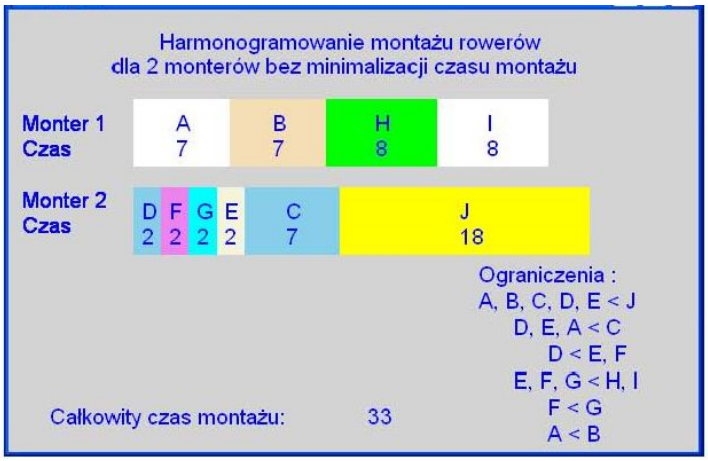
1. Żaden monter nie może być bezczynny.

2. Każdy montaż rozpoczęty przez montera musi być doprowadzony do końca.

Nowy kierownik wyznaczył metodą prób następujący harmonogram montażu dla każdej pary monterów:

```
Czynność                A  B  C  D  E  F  G  H  I  J
Czasy startu czynności  1, 8, 9, 1, 7, 3, 5, 15, 23, 16.
```

Odpowiada mu wykres Gantta przedstawiony na rysunku 6.10.



Rysunek 6.10: Pierwszy harmonogram montażu

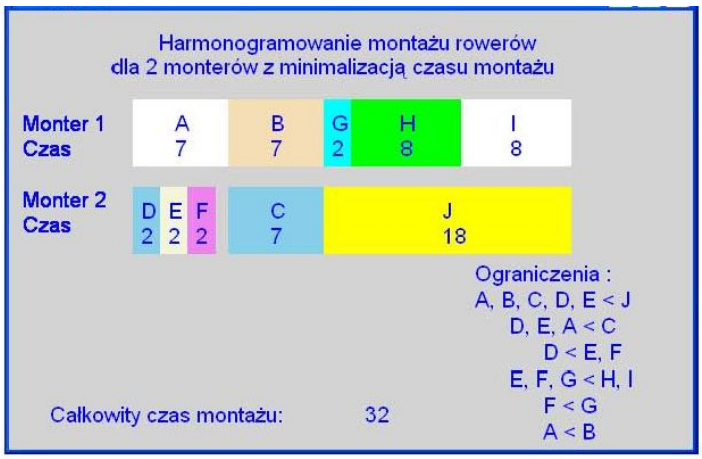
Niestety, całkowity czas trwania montażu roweru to 33 minuty, w związku z czym na 8-godzinnej zmianie można będzie zmontować tylko 14,5 rowera, a więc 10 zespołów zmontuje 145 rowerów, co jest znacznie poniżej założonych w planie 152 rowerów.

Po bezskutecznych próbach napisania lepszego harmonogramu kierownik zlecił jego opracowanie znanemu specjaliście od programowania w logice z ograniczeniami, który przedstawił szereg programów zawartych w przykładzie `6_5_rowery.clp`. Pierwszym z nich jest program minimalizujący całkowity czas montażu, przedstawiony w części `top1`. Wygenerowanemu harmonogramowi, przedstawionemu za pomocą następujących list, których pozycjom odpowiadają kolejne czynności `A`, `B`,...`J`:

```
Czasy startu =  [1, 8,  8,  1, 3, 5, 15, 17, 25, 15]
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,   8, 18]
Czasy konca =   [8, 15, 15, 3, 5, 7, 17, 25, 33, 33]
Koniec = 33
```

```
Calkowity czas montazu = 32,
```

odpowiada wykres Gantta przedstawiony na rysunku 6.11.

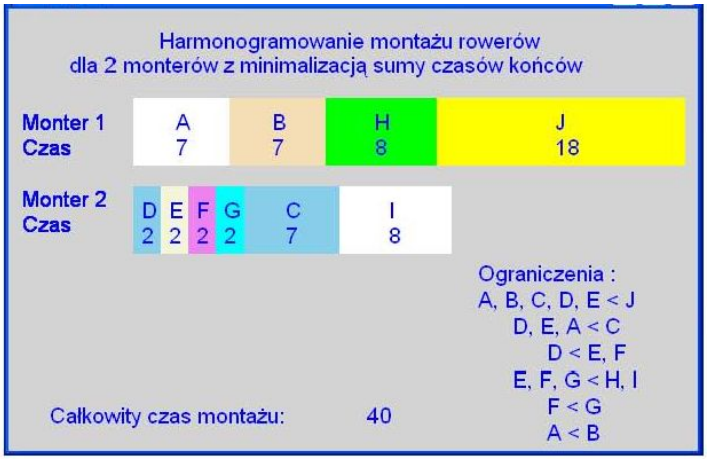


Rysunek 6.11: Drugi harmonogram montażu

Niestety, całkowity czas montażu jest ciągle dłuższy od oczekiwanych 31,5 minut. Ponadto naruszono pierwszą zasadę "zajętości", gdyż pomiędzy czynnościami `F` a `C` jest nielegalna minutowa przerwa w pracy drugiego montera. Kierownik pragnie ją wyeliminować na drodze tworzenia takiego harmonogramu, który minimalizuje sumę czasów końców wykonania wszystkich czynności. Odpowiedni opracowany przez specjalistę program jest wywoływany w przykładzie `6_5_rowery.clp` w części `top2`. W wyniku uzyskuje się harmonogram:

```
Czasy startu =  [1, 8,  9,  1, 3, 5, 7, 15, 16, 23]
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,  8, 18]
Czasy konca =   [8, 15, 16, 3, 5, 7, 9, 23, 24, 41]
Koniec = 41
Calkowity czas montażu = 40,
```

któremu odpowiada wykres Gantta przedstawiony na rysunku 6.12.
Nowy harmonogram jest więc zdecydowanie gorszy od poprzedniego. Aby zwiększyć wydajność montażu, kierownik wypożyczyl elektronarzędzia skracające o



Rysunek 6.12: Trzeci harmonogram montażu

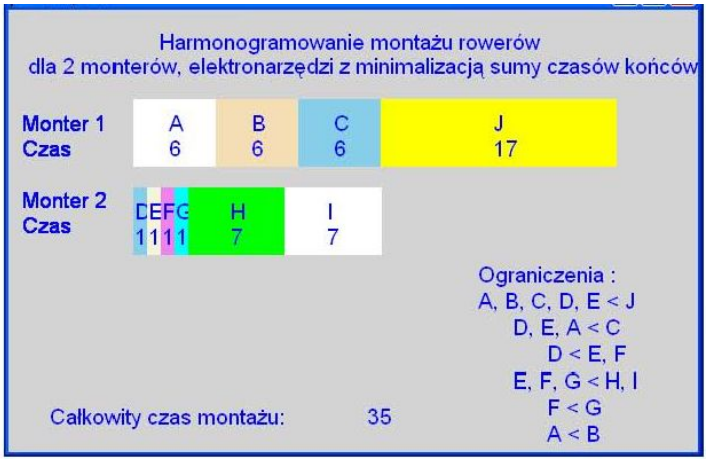
1 minutę czas trwania każdej czynności. Nadal jednak domaga się, aby montaż odbywał się z minimalizacją sumy czasów końców wykonania zadań. Jak to wpłynie na czas montażu? Odpowiedni program jest wywoływany w przykładzie `6_7_rowery.clp` w części `top3`. W wyniku uzyskuje się harmonogram:

```
Czasy startu =  [1, 7,  12, 1, 2, 3, 4,  5, 13, 18]
Czasy montazu = [6, 6,  6,  1, 1, 1, 1,  7,  7, 17]
Czasy konca =   [7, 13, 18, 2, 3, 4, 5, 12, 20, 35]
Koniec = 35
Calkowity czas montazu = 34,
```

któremu odpowiada wykres Gantta przedstawiony na rysunku 6.13.

Gdyby kierownik po wypożyczeniu elektronarzędzi nie upierał się przy minimalizacji sumy czasów końców zadań, lecz zaakceptował minimalizację czasu montażu, można by ów czas wydatnie skrócić. Przedstawia to część `top4` programu, generująca harmonogram:

```
Czasy startu =  [1, 7,  7,  1, 2, 3, 4, 13, 20, 13]
Czasy montazu = [6, 6,  6,  1, 1, 1, 1,  7,  7, 17]
Czasy konca =   [7, 13, 13, 2, 3, 4, 5, 20, 27, 30]
```



Rysunek 6.13: Czwarty harmonogram montażu

```
Koniec = 30
Calkowity czas montazu = 29,
```

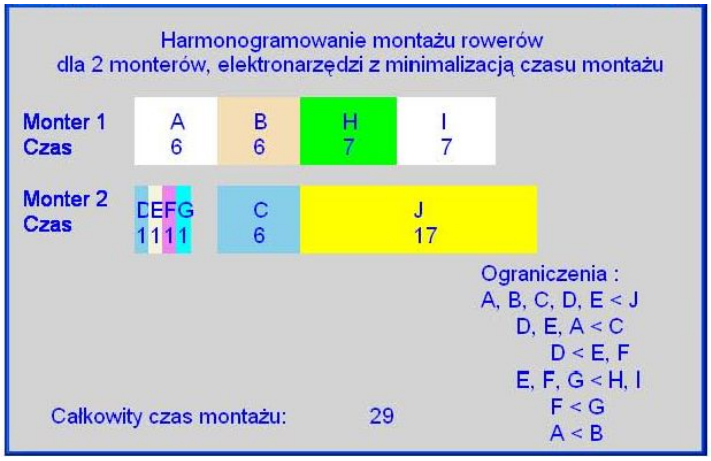
któremu odpowiada wykres Gantta przedstawiony na rysunku 6.14.

Niestety, harmonogram ten zawiera aż 2-minutową przerwę pomiędzy czynnościami `G` i `C`. Zrezygnowany kierownik oddał elektronarzędzia i w akcie desperacji - dla zwiększenia wydajności - zaangażowal dodatkowo 10 monterów. Teraz każdy rower będzie montowany przez trzech monterów. Specjalista od *CLP* ostrzega kierownika, że to nie zmniejszy wcale - z powodu ograniczeń kolejnościowych - czasu montażu. Kierownik nie wierzy jego zapewnieniom, wynikającym z części `top4` programu, która generuje harmonogram:

```
Czasy startu =  [1, 8,  8,  1, 3, 3, 5,  7, 15, 15]
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,  8, 18]
Czasy konca =   [8, 15, 15, 3, 5, 5, 7, 15, 23, 33]
Koniec = 33
Calkowity czas montazu = 32,
```

któremu odpowiada wykres Gantta przedstawiony na rysunku 6.15.

Okazuje się, że czas trwania montażu jest dokładnie taki sam, jak dla dwóch



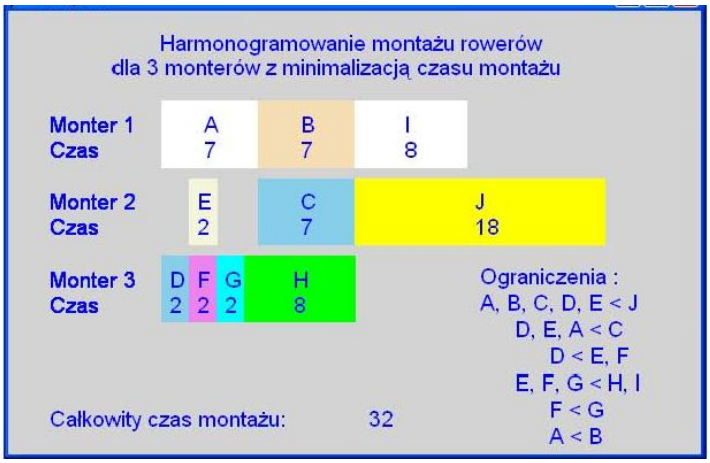
Rysunek 6.14: Piąty harmonogram montażu

monterów pracujących z niewielka przerwą, porównaj harmonogram z rysunku 6.11. Kierownik - w akcie skrajnej desperacji - angażuje kolejnych 10 monterów. Teraz każdy rower będzie montowany przez 4 monterów. Specjalista od *CLP* bezskutecznie ostrzega, że to nic nie da. Kierownik nie wierzy jego zapewnieniom, wynikającym z części `top5` programu, która generuje harmonogram:

```
Czasy startu =  [1, 8,  8,  1, 3, 3, 5,  7,  7, 15]
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,  8, 18]
Czasy konca =   [8, 15, 15, 3, 5, 5, 7, 15, 15, 33]
Koniec = 33
Calkowity czas montazu = 32,
```

któremu odpowiada wykres Gantta przedstawiony na rysunku 6.16.

Kierownik, który swymi nieprzemyślanymi angażami doprowadził fabrykę *ACME* do bankructwa, po kilku nieprzespanych nocach (o czym Ronald Graham już nie wspomina) zaoferował swe usługi (lobbowanie na rzecz dotacji, z budżetu państwowego Absurdolandii, dla przedsiębiorstw trudniących się montażem) znanej nam już z rozdziału 4.5.3 centrali partii "Wszystko dla Wszystkich", gdzie - jako wybitny specjalista od spraw przemysłu - został entuzjastycznie



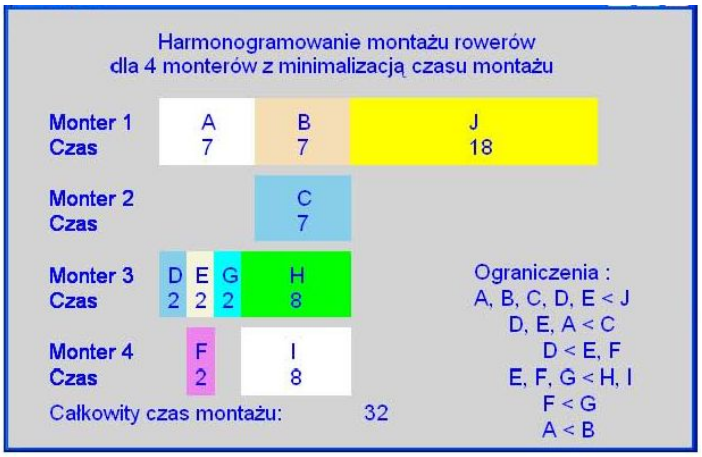
Rysunek 6.15: Szósty harmonogram montażu

przyjęty[3].

Program `6_13_rowery.ecl` jest następujący:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_edge_finder3).
 /*3*/  :- lib(branch_and_bound).

/*4*/  top:-
/*5*/    top1,
/*6*/    top2,
/*7*/    top3,
/*8*/    top4,
/*9*/    top5,
/*10*/   top6.
```

[3]Już dawno temu Alexis de Tocqueville (1805–1859) w swej słynnej książce "Demokracja w Ameryce" napisał: "W Ameryce jest tyle sposobów zarabiania na życie, że mało kto zaczyna zajmować się polityką, jeżeli nie poniósł porażek w innych działalnościach." Czy to tylko w Ameryce? I czy tylko tak dawno temu?



Rysunek 6.16: Siódmy harmonogram montażu

```
 % Minimalizacja czasu montażu dla dwóch monterów:
/*11*/ top1:-
/*12*/   deklaruj_dziedziny(Czasy_Startu,Resurs),
/*13*/   Czasy_Montazu = [7, 7, 7, 2, 2, 2, 2, 8, 8,18],
/*14*/   Czasy_Konca =   [_, _, _, _, _, _, _, _, _, _],
/*15*/   Czasy_Konca :: 1..200,
/*16*/   Limit :: 2,

/*17*/   ograniczenia(Czasy_Startu),
/*18*/   cumulative(Czasy_Startu,Czasy_Montazu,Resurs,Limit),
/*19*/   czasy_konca(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
          Koniec),

/*20*/   bb_min(search(Czasy_Startu,0,smallest,indomain_min,
         bbs(1),[]),Koniec, bb_options{delta:1,timeout:60}),

/*21*/   Czas_montazu is Koniec - 1,
/*22*/   write("Minimalizacja czasu montazu dla dwoch
              monterow:"),
/*23*/   pisz_wyniki(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
              Koniec,Czas_montazu).
```

```
 % Minimalizacja sumy czasów końców montażu dla dwóch monterów:
```

```
/*24*/ top2:-
/*25*/   deklaruj_dziedziny(Czasy_Startu,Resurs),
/*26*/   Czasy_Montazu = [7, 7, 7, 2, 2, 2, 2, 8, 8, 18],
/*27*/   Czasy_Konca =   [K1,K2,K3,K4,K5,K6,K7,K8,K9,K10],
/*28*/   Czasy_Konca :: 1..200,
/*29*/   Limit :: 2,

/*30*/   ograniczenia(Czasy_Startu),
/*31*/   cumulative(Czasy_Startu,Czasy_Montazu,Resurs,Limit),
/*32*/   czasy_konca(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
         Koniec),

/*33*/   Suma_czasow #= K1+K2+K3+K4+K5+K6+K7+K8+K9+K10,

/*34*/   bb_min(search(Czasy_Startu,0,smallest,indomain_min,
        bbs(1),[]),Suma_czasow, bb_options{delta:1,timeout:60}),

/*35*/   Czas_montazu is Koniec - 1,
/*36*/   write("Minimalizacja sumy czasow koncow montazu dla
             dwoch monterow:"),
/*37*/   pisz_wyniki(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
             Koniec,Czas_montazu).

 % Minimalizacja sumy czasów końców montażu dla dwóch
 % monterów i elektronarzędzi:
/*38*/ top3:-
/*39*/   deklaruj_dziedziny(Czasy_Startu,Resurs),
/*40*/   Czasy_Montazu = [6, 6, 6, 1, 1, 1, 1, 7, 7, 17],
/*41*/   Czasy_Konca =   [K1,K2,K3,K4,K5,K6,K7,K8,K9,K10],
/*42*/   Czasy_Konca :: 1..200,
/*43*/   Limit :: 2,

/*44*/   ograniczenia_elektronarzedzi(Czasy_Startu),
/*45*/   cumulative(Czasy_Startu,Czasy_Montazu,Resurs,Limit),
/*46*/   czasy_konca(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
         Koniec),

/*47*/   Suma_czasow #= K1+K2+K3+K4+K5+K6+K7+K8+K9+K10,

/*48*/   bb_min(search(Czasy_Startu,0,smallest,indomain_min,
        bbs(1),[]),Suma_czasow, bb_options{delta:1,timeout:60}),

/*49*/   Czas_montazu is Koniec - 1,
/*50*/   write("Minimalizacja sumy czasow koncow montazu dla
             dwoch monterow i elektronarzedzi:"),
/*51*/   pisz_wyniki(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
```

```
            Koniec,Czas_montazu).

 % Minimalizacja czasu montażu dla dwóch monterów i
 % elektronarzędzi:
/*52*/ top4:-
/*53*/   deklaruj_dziedziny(Czasy_Startu,Resurs),
/*54*/   Czasy_Montazu = [6, 6, 6, 1, 1, 1, 1, 7, 7, 17],
/*55*/   Czasy_Konca =   [_, _, _, _, _, _, _, _, _, _],
/*56*/   Czasy_Konca :: 1..200,
/*57*/   Limit :: 2,

/*58*/   ograniczenia_elektronarzedzi(Czasy_Startu),
/*59*/   cumulative(Czasy_Startu,Czasy_Montazu,Resurs,Limit),
/*60*/   czasy_konca(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
         Koniec),

/*61*/   bb_min(search(Czasy_Startu,0,first_fail,indomain,
        bbs(1),[]),Koniec, bb_options{delta:1,timeout:60}),

/*62*/   Czas_montazu is Koniec - 1,
/*63*/   write("Minimalizacja czasu montazu dla dwoch monterow
            i elektronarzedzi:"),
/*64*/   pisz_wyniki(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
            Koniec,Czas_montazu).

 % Minimalizacja czasu montażu dla trzech monterów:
/*65*/ top5:-
/*66*/   deklaruj_dziedziny(Czasy_Startu,Resurs),
/*67*/   Czasy_Montazu = [7, 7, 7, 2, 2, 2, 2, 8, 8,18],
/*68*/   Czasy_Konca =   [_, _, _, _, _, _, _, _, _, _],
/*69*/   Czasy_Konca :: 1..200,
/*70*/   Limit :: 3,

/*71*/   ograniczenia(Czasy_Startu),
/*72*/   cumulative(Czasy_Startu,Czasy_Montazu,Resurs,Limit),
/*73*/   czasy_konca(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
         Koniec),

/*74*/   bb_min(search(Czasy_Startu,0,first_fail,indomain,
        bbs(1),[]),Koniec, bb_options{delta:1,timeout:60}),

/*75*/   Czas_montazu is Koniec - 1,
/*76*/   write("Minimalizacja czasu montazu dla trzech
            monterow:"),
/*77*/   pisz_wyniki(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
```

```
            Koniec,Czas_montazu).

 % Minimalizacja czasu montażu dla czterech monterów:
/*78*/ top6:-
/*79*/   deklaruj_dziedziny(Czasy_Startu,Resurs),
/*80*/   Czasy_Montazu = [7, 7, 7, 2, 2, 2, 2, 8, 8,18],
/*81*/   Czasy_Konca =   [_, _, _, _, _, _, _, _, _, _],
/*82*/   Czasy_Konca :: 1..200,
/*83*/   Limit :: 4,

/*84*/   ograniczenia(Czasy_Startu),
/*85*/   cumulative(Czasy_Startu,Czasy_Montazu,Resurs,Limit),
/*86*/   czasy_konca(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
         Koniec),

/*87*/   bb_min(search(Czasy_Startu,0,first_fail,indomain,
        bbs(1),[]),Koniec, bb_options{delta:1,timeout:60}),

/*88*/   Czas_montazu is Koniec - 1,
/*89*/   write("Minimalizacja czasu montazu dla czterech
             monterów:"),
/*90*/   pisz_wyniki(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
             Koniec,Czas_montazu).

/*91*/ deklaruj_dziedziny(Czasy_Startu,Resurs):-
/*92*/   Czasy_Startu =  [_, _, _, _, _, _, _, _, _, _],
/*93*/   Resurs =       [_, _, _, _, _, _, _, _, _, _],
/*94*/   Czasy_Startu :: 1..100,
/*95*/   Resurs :: 1.

/*96*/ ograniczenia([A,B,C,D,E,F,G,H,I,J]):-
/*97*/    A + 7 #=< J,
/*98*/    B + 7 #=< J,
/*99*/    C + 7 #=< J,
/*100*/   D + 2 #=< J,
/*101*/   E + 2 #=< J,
/*102*/   A + 7 #=< C,
/*103*/   D + 2 #=< C,
/*104*/   E + 2 #=< C,
/*105*/   D + 2 #=< E,
/*106*/   D + 2 #=< F,
/*107*/   E + 2 #=< H,
/*108*/   F + 2 #=< H,
/*109*/   G + 2 #=< H,
/*110*/   E + 2 #=< I,
/*111*/   F + 2 #=< I,
/*112*/   G + 2 #=< I,
/*113*/   F + 2 #=< G,
```

```
/*114*/   A + 7 #=< B.

/*78*/ ograniczenia_elektronarzedzi([A,B,C,D,E,F,G,H,I,J]):-
/*115*/   A + 6 #=< J,
/*116*/   B + 6 #=< J,
/*117*/   C + 6 #=< J,
/*118*/   D + 1 #=< J,
/*119*/   E + 1 #=< J,
/*120*/   A + 6 #=< C,
/*121*/   D + 1 #=< C,
/*122*/   E + 1 #=< C,
/*123*/   D + 1 #=< E,
/*124*/   D + 1 #=< F,
/*125*/   E + 1 #=< H,
/*126*/   F + 1 #=< H,
/*127*/   G + 1 #=< H,
/*128*/   E + 1 #=< I,
/*129*/   F + 1 #=< I,
/*130*/   G + 1 #=< I,
/*131*/   F + 1 #=< G,
/*132*/   A + 6 #=< B.

/*133*/ czasy_konca(Czasy_Startu,Czasy_Montazu,Czasy_Konca,Koniec):-
/*134*/   ( foreach(S,Czasy_Startu),
/*135*/     foreach(D,Czasy_Montazu),
/*136*/     foreach(K,Czasy_Konca)
/*137*/   do
/*138*/   K #= S + D
/*139*/   ),
/*140*/   Koniec #= max(Czasy_Konca).

/*141*/ pisz_wyniki(Czasy_Startu,Czasy_Montazu,Czasy_Konca,
             Koniec,Czas_montazu):-
/*142*/   printf("%2n        Czasy startu =
    [%d, %d,  %d,  %d, %d, %d, %d, %d, %d, %d].%n", Czasy_Startu),
/*143*/   printf("        Czasy montazu =
    [%d, %d,  %d,  %d, %d, %d, %d,  %d,  %d, %d].%n", Czasy_Montazu),
/*144*/   printf("        Czasy konca =
    [%d, %d, %d, %d, %d, %d, %d, %d, %d, %d].%n", Czasy_Konca),
/*145*/   printf("        Koniec = %d%2n", Koniec),
/*146*/   printf("        Calkowity czas montazu: =
    %d%2n", Czas_montazu).
```

Program generuje komunikat:

```
Found a solution with cost 41
Found a solution with cost 34
Found a solution with cost 33
Minimalizacja czasu montazu dla dwoch monterow:
Czasy startu = [1, 8,  8,  1, 3, 5, 15, 17, 25, 15].
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,  8, 18].
Czasy konca = [8, 15, 15, 3, 5, 7, 17, 25, 33, 33].
Koniec = 33
Calkowity czas montazu: = 32

Found a solution with cost 151
Found no solution with cost 121.0 .. 150.0
Minimalizacja sumy czasow koncow montazu dla dwoch monterow:
Czasy startu = [1, 8,  9,  1, 3, 5, 7, 15, 16, 23].
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,  8, 18].
Czasy konca = [8, 15, 16, 3, 5, 7, 9, 23, 24, 41].
Koniec = 41
Calkowity czas montazu: = 40

Found a solution with cost 119
Found no solution with cost 97.0 .. 118.0
Minimalizacja sumy czasów koncow montazu dla dwoch monterow
        i elektronarzedzi:
Czasy startu = [1, 7,  12,  1, 2, 3, 4, 5, 13, 18].
Czasy montazu = [6, 6,  6,  1, 1, 1, 1,  7,  7, 17].
Czasy konca = [7, 13, 18, 2, 3, 4, 5, 12, 20, 35].
Koniec = 35
Calkowity czas montazu: = 34

Found a solution with cost 37
Found a solution with cost 30
Minimalizacja czasu montazu dla dwoch monterow
        i elektronarzedzi:
Czasy startu = [1, 7,  7,  1, 2, 3, 4, 13, 20, 13].
Czasy montazu = [6, 6,  6,  1, 1, 1, 1,  7,  7, 17].
Czasy konca = [7, 13, 13, 2, 3, 4, 5, 20, 27, 30].
Koniec = 30
Calkowity czas montazu: = 29

Found a solution with cost 33
Minimalizacja czasu montazu dla trzech monterow:
Czasy startu = [1, 8,  8,  1, 3, 3, 5, 7, 15, 15].
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,  8, 18].
Czasy konca = [8, 15, 15, 3, 5, 5, 7, 15, 23, 33].
Koniec = 33
Calkowity czas montazu: = 32

Found a solution with cost 33
```

```
Minimalizacja czasu montazu dla czterech monterow:
Czasy startu = [1, 8,  8,  1, 3, 3, 5, 7, 7, 15].
Czasy montazu = [7, 7,  7,  2, 2, 2, 2,  8,  8, 18].
Czasy konca = [8, 15, 15, 3, 5, 5, 7, 15, 15, 33].
Koniec = 33
Calkowity czas montazu: = 32
```

6.15 Rozładowujemy i załadowujemy statek

Predykat globalny `cumulative/4` jest dostępny również w pożytecznej odmianie 5-argumentowej `cumulative/5`:

```
cumulative(+CzasyStartu,+CzasyTrwania,+Resurs,+Powierzchnie,++Limit+)
```

gdzie `Powierzchnie` są listą iloczynów czasów trwania przez przyporządkowaną im wielkość potrzebnego resursu. Niestety, w odróżnieniu od innych języków *CLP* (np. w odróżnieniu od języka CHIP), iloczyny te musi zaprogramować użytkownik. Zastosujemy ten predykat dla rozwiązania następującego przykładu[4]:

Rozładunek i załadunek statku wymaga wykonania 32 zadań, dla każdego z których jest określona liczba roboczogodzin oraz kolejność wykonania, zgodnie z tablicą 6.2.

Jak widać, nie są znane obciążenia resursów (liczba dokerów potrzebnych dla wykonania zadań) i czasy potrzebne dla wykonania zadań, lecz jedynie ich iloczyny w postaci roboczogodzin.

Do dyspozycji jest 12-osobowy zespół dokerów, każdy z których może pracować conajwyżej 8 godzin. Należy wyznaczyć harmonogram rozładunku i załadunku, minimalizujący całkowity czas potrzebny na te czynności. Odpowiedni program przedstawia `6_14_statek.ecl`:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_edge_finder3).
 /*3*/  :- lib(branch_and_bound).

/*4*/  top:-
```

[4]Temat przykładu został zaczerpnięty z publikacji [Aggoun-93].

Zadanie	Liczba roboczogodzin	Zadanie następne
1	12	2, 4
2	16	3
3	12	5, 7
4	24	5
5	25	6
6	10	8
7	12	8
8	12	9
9	12	10, 14
10	16	11, 12
11	12	13
12	10	13
13	4	15,16
14	15	15
15	6	18
16	9	17
17	12	18
18	14	19, 20, 21
19	4	23
20	4	23
21	4	22
22	8	23
23	28	24
24	40	25
25	16	26,30,31,32
26	3	27
27	3	28
28	12	29
29	8	koniec
30	9	28
31	6	28
32	3	28
33	6	34
34	6	koniec

Tablica 6.2: Rozładunek i załadunek statku

```
  % Lista czasów startu zadań:
/*5*/    LS = [S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,
               S12,S13,S14,S15,S16,S17,S18,S19,S20,S21,S22,
                   S23,S24,S25,S26,S27,S28,S29,S30,S31,S32,S33,S34],

  % Lista czasów trwania zadań:
/*6*/    LD = [D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,
               D12,D13,D14,D15,D16,D17,D18,D19,D20,D21,D22,
                   D23,D24,D25,D26,D27,D28,D29,D30,D31,D32,D33,D34],

  % Lista obciążeń resursów przez zadania - lista liczb dokerów
  % potrzebnych dla wykonania kolejnych zadań:
/*7*/    LR = [R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,
               R12,R13,R14,R15,R16,R17,R18,R19,R20,R21,R22,
                   R23,R24,R25,R26,R27,R28,R29,R30,R31,R32,R33,R34],

  % Lista powierzchni zadań - lista roboczogodzin
  % potrzebnych dla wykonania kolejnych zadań:
/*8*/    LF = [12,16,12,24,25,10,12,12,12,16,12,
               10,4,15,6,9,12,14,4,4,4,8,
                   28,40,16,3,3,12,8,9,6,3,6,6],

/*9*/    LS   :: 1..400,
/*10*/   LD :: 1..40,
/*11*/   LR :: 1..12,
/*12*/   Koniec :: 1..400,
/*13*/   Limit :: 1..12,

/*14*/   cumulative(LS,LD,[R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,
               R12,R13,R14,R15,R16,R17,R18,R19,R20,R21,R22,
                   R23,R24,R25,R26,R27,R28,R29,R30,R31,R32,R33,R34],
                      LF,Limit),

/*15*/   S1 + D1 #=< S2,
/*16*/   S1 + D1 #=< S4,
/*17*/   S2 + D2 #=< S3,
/*18*/   S3 + D3 #=< S5,
/*19*/   S3 + D3 #=< S7,
/*20*/   S4 + D4 #=< S5,
/*21*/   S5 + D5 #=< S6,
/*22*/   S6 + D6 #=< S8,
/*23*/   S7 + D7 #=< S8,
/*24*/   S8 + D8 #=< S9,
/*25*/   S9 + D9 #=< S10,
/*26*/   S9 + D9 #=< S14,
/*27*/   S10 + D10 #=< S11,
/*28*/   S10 + D10 #=< S12,
/*29*/   S11 + D11 #=< S13,
/*30*/   S12 + D12 #=< S13,
```

```
/*31*/   S13 + D13 #=< S15,
/*32*/   S13 + D13 #=< S16,
/*33*/   S14 + D14 #=< S15,
/*34*/   S15 + D15 #=< S18,
/*35*/   S16 + D16 #=< S17,
/*36*/   S17 + D17 #=< S18,
/*37*/   S18 + D18 #=< S19,
/*38*/   S18 + D18 #=< S20,
/*39*/   S18 + D18 #=< S21,
/*40*/   S19 + D19 #=< S23,
/*41*/   S20 + D20 #=< S23,
/*42*/   S21 + D21 #=< S22,
/*43*/   S22 + D22 #=< S23,
/*44*/   S23 + D23 #=< S24,
/*45*/   S24 + D24 #=< S25,
/*46*/   S25 + D25 #=< S26,
/*47*/   S25 + D25 #=< S30,
/*48*/   S25 + D25 #=< S31,
/*49*/   S25 + D25 #=< S32,
/*50*/   S26 + D26 #=< S27,
/*51*/   S27 + D27 #=< S28,
/*52*/   S28 + D28 #=< S29,
/*53*/   S29 + D29 #=< 400,
/*54*/   S30 + D30 #=< S28,
/*55*/   S31 + D31 #=< S28,
/*56*/   S32 + D32 #=< S33,
/*57*/   S33 + D33 #=< S34,
/*58*/   S34 + D34 #=< 400,

  % Wyznaczanie listy końców zadań:
/*59*/   (
/*60*/   foreach(I,LS),
/*61*/   foreach(J,LD),
/*62*/   foreach(K,LK)
/*63*/   do
/*64*/   K #= I + J
/*65*/   ),

  % Wyznaczanie listy powierzchni zadań:
/*66*/   (
/*67*/   foreach(I,LD),
/*68*/   foreach(J,LR),
/*69*/   foreach(F,LF)
/*70*/   do
/*71*/   F #= I * J
/*72*/   ),

/*73*/   maxlist(LK,Koniec),
```

```
/*74*/   minimize(labeling(LS,LD,LR),Koniec),nl,

/*75*/   writeln("Liczba dokerow ":Limit),
/*76*/   writeln("Koniec rozladunku i zaladunku ":Koniec),nl,

/*77*/   writeln("Zadanie   Start  Trwanie Dokerzy  Koniec"),
/*78*/   wypisz(LS,LD,LR,LK,01).

/*79*/ labeling([X1|X],[Y1|Y],[Z1|Z]):-
/*80*/   indomain(X1),
/*81*/   indomain(Y1),
/*82*/   indomain(Z1),
/*83*/   labeling(X,Y,Z).
/*84*/ labeling([],[],[]).

/*85*/ wypisz([Sg|Sk],[Dg|Dk],[Rg|Rk],[Kg|Kk],N):-
/*86*/   printf("%d\t%d\t%d\t%d\t%d\t",[N,Sg,Dg,Rg,Kg]),nl,
/*87*/   N1 is N+1,
/*88*/   wypisz(Sk,Dk,Rk,Kk,N1).
/*89*/ wypisz([],[],[],[],_).
```

Program generuje następujący komunikat:

```
Found a solution with cost 54
Found a solution with cost 53
Found a solution with cost 44
Found a solution with cost 43
Found no solution with cost 37.0 .. 42.0

Liczba dokerow  : 12
Koniec rozladunku i zaladunku : 43

Zadanie Start Trwanie Dokerzy Koniec
   1      1       1       12     2
   2      2       2       8      4
   3      4       1       12     5
   4      5       2       12     7
   5      7       5       5      12
   6      12      1       10     13
   7      7       2       6      9
   8      13      1       12     14
   9      14      1       12     15
   10     15      2       8      17
```

```
11      17      1       12      18
12      18      1       10      19
13      19      1       4       20
14      19      3       5       22
15      22      1       6       23
16      20      3       3       23
17      23      1       12      24
18      24      2       7       26
19      26      1       4       27
20      26      1       4       27
21      26      1       4       27
22      27      1       8       28
23      28      4       7       32
24      32      4       10      36
25      36      2       8       38
26      38      1       3       39
27      39      1       3       40
28      40      1       12      41
29      41      2       4       43
30      38      1       9       39
31      39      1       6       40
32      39      1       3       40
33      41      1       6       42
34      42      1       6       43,
```

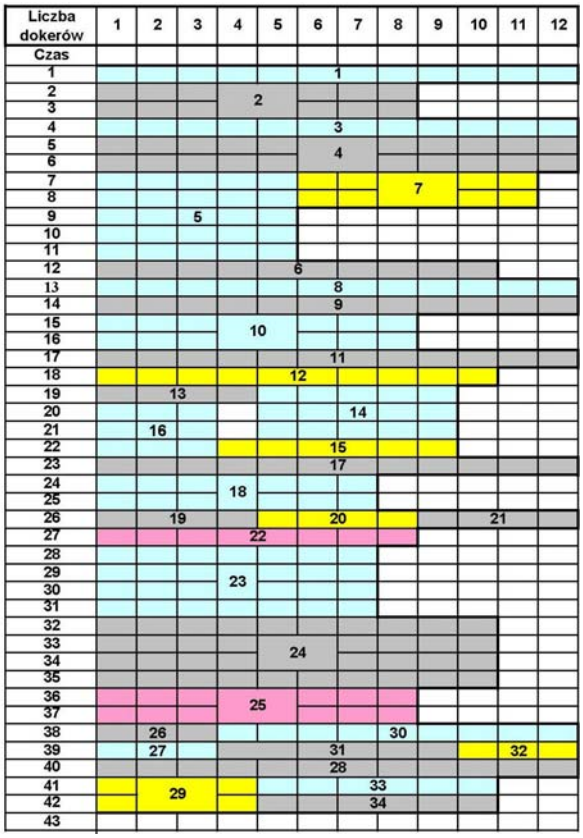
gdzie `Start` oznacza czas rozpoczęcia zadania, `Trwanie` jest czasem potrzebnym dla wykonania zadania, `Dokerzy` to liczba dokerów potrzebnych do wykonania zadania, a `Koniec` jest czasem zakończenia zadania.

Powyższy harmonogram jest "słabo czytelny". Można go jednak przedstawić w postaci graficznej z rysunku 6.17, gdzie dla poprawy czytelności zadania sąsiadujące ze sobą oznaczono różnymi kolorami.

6.16 Czym jest problem "linii zadań"?

Omawiane przykłady czytania gazet i montowania rowerów są przykładami ogólnego problemu harmonogramowania znanego pod nazwą *linii zadań* (ang. *job-shop*). Można go zdefiniować następująco:

Należy dokonać rozdziału `n` zadań (zadanie = *job*) pomiędzy `m` maszyn znajdu-



Rysunek 6.17: Optymalny harmonogram rozładowania i ładowania statku

jących się w hali produkcyjnej (hala produkcyjna = *shop*). Każde zadanie składa się ze zbioru operacji (*tasks*) które należy wykonać w ściśle określonej kolejności. Każda operacja ma całkowitoliczbowy czas trwania i musi być wykonana na określonej maszynie ze zbioru dostępnych maszyn. Zakłada się, że:

- w dowolnej chwili dowolna maszyna może wykonywać tylko jedną operację;
- znane są czasy wykonania wszystkich operacji na wszystkich maszynach;
- dla wszystkich zadań znana jest kolejność wykonywania operacji;

- wykonywanie operacji nie może być przerwane;
- każda maszyna jest albo w dostępna albo niedostępna;
- dopuszczalne są przerwy pomiędzy kolejnymi dwoma operacjami danego zadania;
- żadna maszyna nie może zostać zastąpiona inną;
- operacje żadnej z maszyn nie zależą od którejkolwiek z pozostałych maszyn;
- żadne z zadań nie zależy od któregokolwiek z pozostałych zadań.

Celem problemu *linii zadań* bywa najczęściej określenie takich czasów rozpoczynania kolejnych operacji dla poszczególnych zadań, by - przy zachowaniu ograniczeń kolejnościowych i czasów trwania operacji - uzyskać minimum czasu wykonania wszystkich operacji. Czas ten nazywa się *długością uszeregowania*, ale bywa bardzo często nazywany z angielska *makespan*'em.

Terminologia ta ma charakter historyczny i wynika z najwcześniejszych prac nad harmonogramowaniem zadań w manufakturach. Zadania te (ang. *job*) były wykonywane w halach fabrycznych (ang. *shop*) przy pomocy odpowiednio przeszkolonych ludzi lub za pomocą pewnych maszyn. Obecnie, aczkolwiek harmonogramowanie jest ciągle bardzo ważną czynnością w zarządzaniu procesami produkcji przemysłowej (w szczególności procesami montażu), pojawiła się duża liczba zastosowań nie-przemysłowych, np. biznesowych i wojskowych, dla których stara terminologia, ze względu na swą adekwatność, jest wciąż stosowana. I tak np.:

- W przykładach z czytaniem gazet były cztery zadania (przeczytanie przez każdego studenta kompletu gazet w określonej kolejności), "maszynami" (resursami) były owe cztery gazety, operacjami zaś czytanie tych gazet.
- W towarzystwach ubezpieczeniowych można dopatrzeć się "hali", w której ubezpieczyciele ("maszyny") przetwarzają polisy ubezpieczeniowe ("zadania") na drodze wypełniania szeregu formularzy.

Spróbujmy sformułować ogólną definicję problemu *job-shop*. Niech $M = \{M_1, M_2, \ldots, M_m\}$ będzie zbiorem maszyn, $Z = \{Z_1, Z_2, \ldots, Z_n\}$ zbiorem zadań. Dla każdego zadania Z_i definiuje się ciąg m kolejno wykonywanych operacji

$O_i = \{O_{i,1}, O_{i,2}, \ldots, O_{i,m}\}$, każda z których jest wykonywana na innej maszynie:

$$O_{i,1} \rightarrow M_{i,1}$$
$$O_{i,2} \rightarrow M_{i,2}$$
$$\ldots\ldots\ldots\ldots$$
$$O_{i,m} \rightarrow M_{i,m}$$

i każda z których trwa określony czas:

$$O_{i,1} \rightarrow T_{i,1}$$
$$O_{i,2} \rightarrow T_{i,2}$$
$$\ldots\ldots\ldots\ldots$$
$$O_{i,m} \rightarrow T_{i,m}.$$

Niech $O = \{Ope(1), Ope(2), \ldots, Ope(m)\} = \{1, 2, \ldots, m\}$ będzie uporządkowanym zbiorem liczb naturalnych odpowiadających kolejno wykonywanym operacjom. Jeżeli każdemu elementowi (i, j) iloczynu kartezjańskiego $Z \times O$ przyporządkować parę $M_{i,j}, T_{i,j}$, to otrzymane tą drogą dwie funkcje (*funkcja maszyn* i *funkcja czasów operacji*) definiują problem typu *linii zadań*.

Dla ilustracji tych pojęć dobrze jest spojrzeć raz jeszcze na tabelę 6.1, która definiuje wymienione dwie funkcje dla zadania czytania gazet: wiersze odpowiadają poszczególnym studentom czytającym gazety (to są "zadania"), kolumny odpowiadają kolejnym czytaniom (to są "operacje"), a w środku każdej komórki są nazwy gazet ("maszyn") i czasy czytania ("czasy trwania operacji").

Powróćmy do ogólnej definicji. Jeżeli założyć, że wszystkie zadania są wykonywane na każdej z maszyn i pominąć ograniczenia kolejnościowe, to liczba możliwych harmonogramów jest równa $(n!)^m$. Z tablicy 6.3 widać, jak szybko rośnie liczba możliwych harmonogramów.

m maszyny	n zadania	$(n!)^m$
1	5	120
3	5	1.7 million
5	5	25000 million

Tablica 6.3: Przykładowy wzrost liczby możliwych harmonogramów

A więc jeżeli chcieć rozwiązać problem minimalizacji długości uszeregowania metodą przeglądu zupełnego, to należy wygenerować *wszystkie* $(n!)^m$ harmonogramy, testować spełnienie ograniczeń kolejnościowych i w przypadku pozytywnego testu wyznaczyć długość uszeregowania[5]

Poniżej zobaczymy, że ECL^iPS^e umożliwia deklaratywne rozwiązanie problemu minimalizacji długości uszeregowania dla linii zadań bez uciekania sie do przeglądu zupełnego. Zostanie to zilustrowane dwoma standardowymi *benchmarkami*, często stosowanymi dla oceny efektywności metody rozwiązywania.

6.17 Harmonogramowanie linii zadań - benchmark MT6

Funkcje maszyn (M) i czasów operacji (T) tego benchmarku są definiowane tablicą z rysunku 6.18.

	Op 0		Op 1		Op 2		Op 3		Op 4		Op 5	
	M	T	M	T	M	T	M	T	M	T	M	T
Zadanie 0	2	1	0	3	1	6	3	7	5	3	4	6
Zadanie 1	1	8	2	5	4	10	5	10	0	10	3	4
Zadanie 2	2	5	3	4	5	8	0	9	1	1	4	7
Zadanie 3	1	5	0	5	2	5	3	3	4	8	5	9
Zadanie 4	2	9	1	3	4	5	5	4	0	3	3	1
Zadanie 5	1	3	3	3	5	9	0	10	4	4	2	1

Rysunek 6.18: Funkcje maszyn (M) i czasów operacji (T) benchmarku MT6

W problemie tym jest `6` *zadań*, każde z których wymaga wykonania `6` *operacji* w zadanej kolejności na `6` *maszynach*. Minimalną długość uszeregowania dla tego problemu wyznacza program `6_15_MT6.ecl`:

```
:-lib(ic).
:-lib(ic_edge_finder3).
```

[5]Podkreśla się potrzebę wyznaczania *wszystkich* harmonogramów; nie wiemy bowiem, czy minimalny czas uszeregowania nie wystąpi dla ostatniego z wyznaczonych harmonogramów.

```
    :-lib(branch_and_bound).
    :-lib(lists).

top:-
 %     Sji - czas startu dla operacji  i-tej  zadania j-tego:
   S = [
 S00, S01, S02, S03, S04, S05,
 S10, S11, S12, S13, S14, S15,
 S20, S21, S22, S23, S24, S25,
 S30, S31, S32, S33, S34, S35,
 S40, S41, S42, S43, S44, S45,
 S50, S51, S52, S53, S54, S55
 ],
 S :: 0..70,

 % Czasy końców zadań:
 E = [E0, E1, E2, E3, E4, E5],
 E :: 0..100,

 % Resurs dla zadań:
 R = [1, 1, 1, 1, 1, 1],

 % Ograniczenia kolejnościowe dla operacji:
 S01 #>= S00 + 1, S02 #>= S01 + 3, S03 #>= S02 + 6, S04 #>= S03 + 7,
       S05 #>= S04 + 3, E0 #>= S05 + 6,
 S11 #>= S10 + 8, S12 #>= S11 + 5, S13 #>= S12 + 10, S14 #>= S13 + 10,
       S15 #>= S14 + 10, E1 #>= S15 + 4,
 S21 #>= S20 + 5, S22 #>= S21 + 4, S23 #>= S22 + 8, S24 #>= S23 + 9,
      S25 #>= S24 + 1, E2 #>= S25 + 7,
 S31 #>= S30 + 5, S32 #>= S31 + 5, S33 #>= S32 + 5, S34 #>= S33 + 3,
      S35 #>= S34 + 8, E3 #>= S35 + 9,
 S41 #>= S40 + 9, S42 #>= S41 + 3, S43 #>= S42 + 5, S44 #>= S43 + 4,
       S45 #>= S44 + 3, E4 #>= S45 + 1,
   S51 #>= S50 + 3, S52 #>= S51 + 3, S53 #>= S52 + 9, S54 #>= S53 + 10,
      S55 #>= S54 + 4, E5 #>= S55 + 1,

   %     Każda maszyna istnieje tylko w jednym egzemplarzu;
   %     dlatego może każdorazowo wykonywać tylko jedną operację:

   % maszyna 0 może w każdej chwili  wykonywać tylko jedną operację:
   cumulative([S01, S14, S23, S31, S44, S53], [3, 10, 9, 5, 3, 10], R, 1),

   % maszyna 1 może w każdej chwili  wykonywać tylko jedną operację:
   cumulative([S02, S10, S24, S30, S41, S50],[6, 8, 1, 5, 3, 3], R, 1),

   % maszyna 2 może w każdej chwili  wykonywać tylko jedną operację:
   cumulative([S00, S11, S20, S32, S40, S55],[1, 5, 5, 5, 9, 1], R, 1),

   % maszyna 3 może w każdej chwili  wykonywać tylko jedną operację:
```

```
    cumulative([S03, S15, S21, S33, S45, S51],[7, 4, 4, 3, 1, 3], R, 1),

    % maszyna 4 może w każdej chwili  wykonywać tylko jedną operację:
    cumulative([S05, S12, S25, S34, S42, S54],[6, 10, 7, 8, 5, 4], R, 1),

    % maszyna 5 może w każdej chwili  wykonywać tylko jedną operację:
    cumulative([S04, S13, S22, S35, S43, S52],[3, 10, 8, 9, 4, 9], R, 1),

%     Każde zadanie "istnieje" tylko w jednym egzemplarzu; dlatego
%     każdorazowo można wykonywać tylko jedną  operację tego zadania:

    % dla zadania 0 można w każdej chwili wykonywać tylko jedną operację:
    cumulative([S00, S01, S02, S03, S04, S05],[1, 3, 6, 7, 3, 6], R, 1),

    % dla zadania 1 można w każdej chwili wykonywać tylko jedną operację:
    cumulative([S10, S11, S12, S13, S14, S15],[8, 5, 10, 10, 10, 4], R, 1),

    % dla zadania 2 można w każdej chwili wykonywać tylko jedną operację:
    cumulative([S20, S21, S22, S23, S24, S25],[5, 4, 8, 9, 1, 7], R, 1),

    % dla zadania 3 można w każdej chwili wykonywać tylko jedną operację:
    cumulative([S30, S31, S32, S33, S34, S35],[5, 5, 5, 3, 8, 9], R, 1),

    % dla zadania 4 można w każdej chwili wykonywać tylko jedną operację:
    cumulative([S40, S41, S42, S43, S44, S45],[9, 3, 5, 4, 3, 1], R, 1),

    % dla zadania 5 można w każdej chwili wykonywać tylko jedną operację:
    cumulative([S50, S51, S52, S53, S54, S55],[3, 3, 9, 10, 4, 1], R, 1),

    append(S, E, SE),
    maxlist(E, M),
    bb_min(uziemienie(SE), M, bb_options with [strategy:continue,from:0,to:100]),
    write("Czasy koncow zadan = "),write(E),nl,
    write("Minimalna długosc uszeregowania = "),write(M),nl,nl,

    write("  S00="),write(S00),
    write("  S01="),write(S01),
    write("  S02="),write(S02),
    write("  S03="),write(S03),
    write("  S04="),write(S04),
    write("  S05="),write(S05),nl,

    write("  S10="),write(S10),
    write("  S11="),write(S11),
    write("  S12="),write(S12),
    write("  S13="),write(S13),
    write("  S14="),write(S14),
    write("  S15="),write(S15),nl,
```

```
write("  S20="),write(S20),
write("  S21="),write(S21),
write("  S22="),write(S22),
write("  S23="),write(S23),
write("  S24="),write(S24),
write("  S25="),write(S25),nl,

write("  S30="),write(S30),
write("  S31="),write(S31),
write("  S32="),write(S32),
write("  S33="),write(S33),
write("  S34="),write(S34),
write("  S35="),write(S35),nl,

write("  S40="),write(S40),
write("  S41="),write(S41),
write("  S42="),write(S42),
write("  S43="),write(S43),
write("  S44="),write(S44),
write("  S45="),write(S45),nl,

write("  S50="),write(S50),
write("  S51="),write(S51),
write("  S52="),write(S52),
write("  S53="),write(S53),
write("  S54="),write(S54),
write("  S55="),write(S55),nl.

uziemienie(Wszystkie_Zmienne):-
najpierw_srodek(Wszystkie_Zmienne, Wszystkie_Zmienne_Przemieszane),
(fromto(Wszystkie_Zmienne_Przemieszane, Zmienne, Zmienne_Pozostale, []) do
delete(Zmienna, Zmienne, Zmienne_Pozostale, 0, max_regret),
indomain(Zmienna, min)
).

najpierw_srodek(Lista, Przemieszana):-
halve(Lista, F, B),
reverse(F, RF),
splice(B, RF, Przemieszana).
```

Program generuje komunikat :

```
Found a solution with cost 67
Found a solution with cost 64
Found a solution with cost 61
Found a solution with cost 60
Found a solution with cost 59
```

```
Found a solution with cost 58
Found a solution with cost 57
Found a solution with cost 56
Found a solution with cost 55
Found no solution with cost 47.0 .. 54.0
Czasy koncow zadan = [55, 52, 45, 54, 53, 50]
Minimalna długosc uszeregowania = 55

S00=5  S01=6  S02=16  S03=30  S04=42  S05=49
S10=0  S11=8  S12=13  S13=28  S14=38  S15=48
S20=0  S21=5  S22=9  S23=18  S24=27  S25=38
S30=8  S31=13  S32=22  S33=27  S34=30  S35=45
S40=13  S41=22  S42=25  S43=38  S44=48  S45=52
S50=13  S51=16  S52=19  S53=28  S54=45  S55=49
```

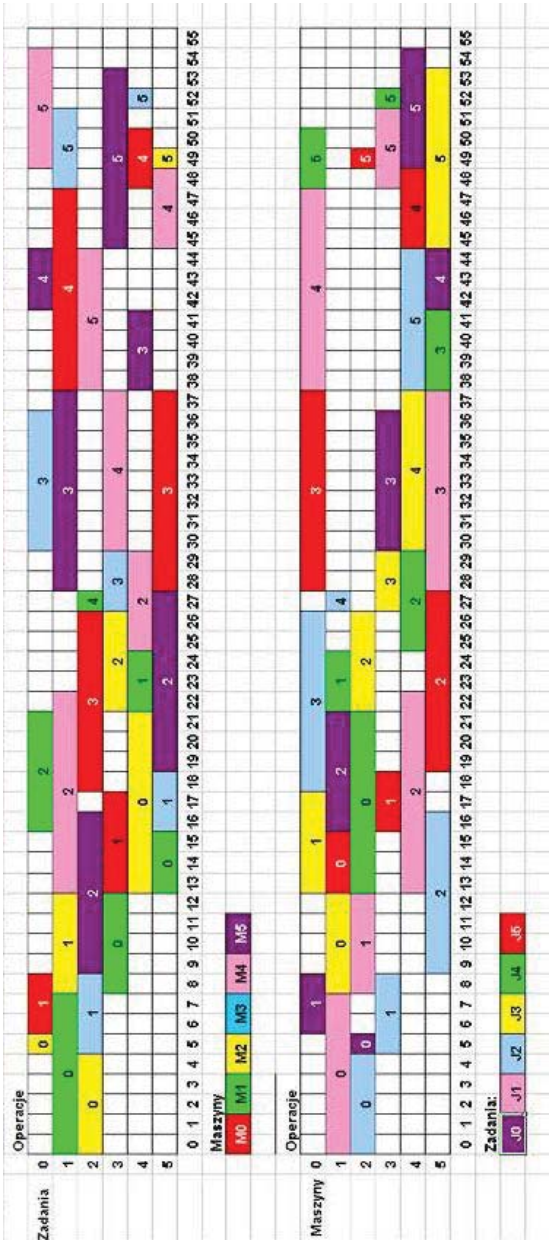
Rozwiązanie przedstawia wykres Gantta z rysunku 6.19.

6.18 Harmonogramowanie linii zadań - benchmark MT10

Na rysunku 6.20 przedstawiono definicję słynnego *benchmarku* MT10 dla rozwiązywania zadań typu job-shop. *Job-shop* tego *benchmarku* obejmuje 10 zadań dla 10 maszyn, wymaga więc rozwiązania problemu optymalizacji całkowitoliczbowej dla 100 zmiennych. Benchmark MT10 został sformułowany w 1963 r. przez J.F. Muth'a i G.L. Thompson'a w książce [Muth-63]. Benchmark ten okazał się przez długi czas zdumiewająco odporny na bardzo liczne próby wyznaczenia rozwiązania optymalnego. Pierwsze „rozsądne" (nieoptymalne) rozwiązanie uzyskano dopiero w latach 1981-1982 dla *makespan*'u równego `935`. Optymalne rozwiązanie z *makespan*'em równym `930` uzyskali w 1987 r. Carlier i Pinson, patrz [Carlier-89]. Do chwili obecnej MT10 stanowi wyzwanie dla konstruktorów algorytmów optymalizacji całkowitoliczbowej i programów *CLP*.

Złożoność problemu skłania nas do sprawdzenia - przed próbą wyznaczenia rozwiązania - czy rozwiązanie takie wogóle istnieje. Posłużymy się w tym celu programam `6_16_mt10_test.ecl` o następującej postaci:

```
/*1*/  :- lib(ic).

/*2*/  top:-
/*3*/    S = [S11,S12,S13,S14,S15,S16,S17,S18,S19,S1A,
              S21,S22,S23,S24,S25,S26,S27,S28,S29,S2A,
              S31,S32,S33,S34,S35,S36,S37,S38,S39,S3A,
              S41,S42,S43,S44,S45,S46,S47,S48,S49,S4A,
```



Rysunek 6.19: Wykresy Gantta dla benchmarku MT6

	Op. 1		Op. 2		Op. 3		Op. 4		Op. 5		Op. 6		Op. 7		Op. 8		Op. 9		Op. A	
	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T
Zad. 1	1	29	2	78	3	9	4	36	5	49	6	11	7	62	8	56	9	44	10	21
Zad. 2	1	43	3	90	5	75	10	11	4	69	2	28	7	46	6	46	8	72	9	30
Zad. 3	2	91	1	85	4	39	3	74	9	90	6	10	8	12	7	89	10	45	5	33
Zad. 4	2	81	3	95	1	71	5	99	7	9	9	52	8	85	4	98	10	22	6	43
Zad. 5	3	14	1	6	2	22	6	61	4	26	5	69	9	21	8	49	10	72	7	53
Zad. 6	3	84	2	2	6	52	4	95	9	48	10	72	1	47	7	65	5	6	8	25
Zad. 7	2	46	1	37	4	61	3	13	7	32	6	21	10	32	9	89	8	30	5	55
Zad. 8	3	31	1	86	2	46	66	74	5	32	7	88	9	19	10	48	8	36	4	79
Zad. 9	1	76	2	69	4	76	6	51	3	85	10	11	7	40	8	89	5	26	9	74
Zad. A	2	85	1	13	3	61	7	7	9	64	10	76	6	47	4	52	5	90	8	45

Rysunek 6.20: Funkcje maszyn (M) i czasów operacji (T) benchmarku MT10

```
            S51,S52,S53,S54,S55,S56,S57,S58,S59,S5A,
            S61,S62,S63,S64,S65,S66,S67,S68,S69,S6A,
            S71,S72,S73,S74,S75,S76,S77,S78,S79,S7A,
            S81,S82,S83,S84,S85,S86,S87,S88,S89,S8A,
            S91,S92,S93,S94,S95,S96,S97,S98,S99,S9A,
            SA1,SA2,SA3,SA4,SA5,SA6,SA7,SA8,SA9,SAA],

    %     SIJ - czas startu dla J-tej operacji I-tego zadania

/*4*/    E = [E1,E2,E3,E4,E5,E6,E7,E8,E9,EA],
/*5*/    E :: 655..1000,
/*6*/    S :: 0..1000,

 %     Ograniczenia kolejności operacji: w ramach każdego zadania,
 %     operacja następna nie może wyprzedzać operacji poprzedniej:

/*7*/     S12 #>= S11+29,   /*8*/     S13 #>= S12+78,
/*9*/     S14 #>= S13+9,    /*10*/    S15 #>= S14+36,
/*11*/    S16 #>= S15+49,   /*12*/    S17 #>= S16+11,
/*13*/    S18 #>= S17+62,   /*14*/    S19 #>= S18+56,
/*15*/    S1A #>= S19+44,   /*16*/    E1  #>= S1A+21,

/*17*/    S22 #>= S21+43,   /*18*/    S23 #>= S22+90,
/*19*/    S24 #>= S23+75,   /*20*/    S25 #>= S24+11,
/*21*/    S26 #>= S25+69,   /*22*/    S27 #>= S26+28,
/*23*/    S28 #>= S27+46,   /*24*/    S29 #>= S28+46,
/*25*/    S2A #>= S29+72,   /*26*/    E2  #>= S2A+30,
```

```
/*27*/    S32 #>= S31+91,   /*28*/    S33 #>= S32+85,
/*29*/    S34 #>= S33+39,   /*30*/    S35 #>= S34+74,
/*31*/    S36 #>= S35+90,   /*32*/    S37 #>= S36+10,
/*33*/    S38 #>= S37+12,   /*34*/    S39 #>= S38+89,
/*35*/    S3A #>= S39+45,   /*36*/    E3  #>= S3A+33,

/*37*/    S42 #>= S41+81,   /*38*/    S43 #>= S42+95,
/*39*/    S44 #>= S43+71,   /*40*/    S45 #>= S44+99,
/*41*/    S46 #>= S45+9,    /*42*/    S47 #>= S46+52,
/*43*/    S48 #>= S47+85,   /*44*/    S49 #>= S48+98,
/*45*/    S4A #>= S49+22,   /*46*/    E4  #>= S4A+43,


/*47*/    S52 #>= S51+14,   /*48*/    S53 #>= S52+6,
/*49*/    S54 #>= S53+22,   /*50*/    S55 #>= S54+61,
/*51*/    S56 #>= S55+26,   /*52*/    S57 #>= S56+69,
/*53*/    S58 #>= S57+21,   /*54*/    S59 #>= S58+49,
/*55*/    S5A #>= S59+72,   /*56*/    E5  #>= S5A+53,

/*57*/    S62 #>= S61+84,   /*58*/    S63 #>= S62+2,
/*59*/    S64 #>= S63+52,   /*60*/    S65 #>= S64+95,
/*61*/    S66 #>= S65+48,   /*62*/    S67 #>= S66+72,
/*63*/    S68 #>= S67+47,   /*64*/    S69 #>= S68+65,
/*65*/    S6A #>= S69+6,    /*66*/    E6  #>= S6A+25,

/*67*/    S72 #>= S71+46,   /*68*/    S73 #>= S72+37,
/*69*/    S74 #>= S73+61,   /*70*/    S75 #>= S74+13,
/*71*/    S76 #>= S75+32,   /*72*/    S77 #>= S76+21,
/*73*/    S78 #>= S77+32,   /*74*/    S79 #>= S78+89,
/*75*/    S7A #>= S79+30,   /*76*/    E7  #>= S7A+55,

/*77*/    S82 #>= S81+31,   /*78*/    S83 #>= S82+86,
/*79*/    S84 #>= S83+46,   /*80*/    S85 #>= S84+74,
/*81*/    S86 #>= S85+32,   /*82*/    S87 #>= S86+88,
/*83*/    S88 #>= S87+19,   /*84*/    S89 #>= S88+48,
/*85*/    S8A #>= S89+36,   /*86*/    E8  #>= S8A+79,

/*87*/    S92 #>= S91+76,   /*88*/    S93 #>= S92+69,
/*89*/    S94 #>= S93+76,   /*90*/    S95 #>= S94+51,
/*91*/    S96 #>= S95+85,   /*92*/    S97 #>= S96+11,
/*93*/    S98 #>= S97+40,   /*94*/    S99 #>= S98+89,
/*95*/    S9A #>= S99+26,   /*96*/    E9  #>= S9A+74,

/*97*/    SA2 #>= SA1+85,   /*98*/    SA3 #>= SA2+13,
/*99*/    SA4 #>= SA3+61,   /*100*/   SA5 #>= SA4+7,
/*101*/   SA6 #>= SA5+64,   /*102*/   SA7 #>= SA6+76,
/*103*/   SA8 #>= SA7+47,   /*104*/   SA9 #>= SA8+52,
/*105*/   SAA #>= SA9+90,   /*106*/   EA  #>= SAA+45,
```

```
/*107*/    append(S,E,SE),
/*108*/    labeling(SE).
```

Program ten zawiera wyłącznie dane o kolejności operacji i czasie ich trwania, zaczerpnięte z treści benchmarku. Generuje on komunikat:

"Yes (0.02s cpu, solution 1, maybe more)",

świadczący o istnieniu rozwiązania i skłaniający nas do jego wyznaczenia.

Efektywny program rozwiązywania benchmarku MT10 (i innych podobnych benchmarków) na platformie ECL^iPS^e, opracowany przez J. Schimpf'a (patrz [Schimpf-10] i bazujący na alorytmie przedstawionym w [Baptiste-95], można znaleźć w witrynie `http://www.eclipseclp.org/eclipse/examples`, w części `Planning and Scheduling`, w podrozdziale `Jobshop Scheduling`.

Poniżej zostanie przedstawiony program `6_17_mt10.ecl`, którego celem jest wyznaczenie optymalnych czasów startu operacji i wykazanie, że minimalny *makespan* to `930`. Skorzystano przy tym z pewnej informacji *a priori* odnośnie do dziedzin poszczególnych czasów startu, która to informacja została w dużej części uzyskana na drodze wyznaczenia najwcześniejszego początku i najpóźniejszego początku każdej z operacji. *Najwcześniejsze początki* każdej operacji zadania zostały wyznaczone jako suma czasów trwania wszystkich operacji poprzedzających w tym zadaniu. *Najpóźniejsze początki* każdej operacji zostały wyznaczone jako różnica górnego krańca dziedziny końców (= 1000) i sumy czasów następnych operacji zadania. Wyjątkiem są dziedziny opatrzone komentarzem `korekta`, które - w celu przyśpieszenia wyznaczenia rozwiązania - zostały dodatkowo zawężone. Program `6_17_mt10.ecl` jest więc programem pozbawionym cech ogólności, które ma program Schimpf'a: zmiana danych benchmarku wymaga "dostrojenia" programu `6_17_mt10.ecl`, który ma następującą postać:

```
/*1*/  :- lib(ic).
/*2*/  :- lib(ic_edge_finder3).
 /*3*/  :- lib(branch_and_bound).
 /*4*/  :- lib(lists).

/*5*/  top:-
/*6*/    S = [S11,S12,S13,S14,S15,S16,S17,S18,S19,S1A,
             S21,S22,S23,S24,S25,S26,S27,S28,S29,S2A,
```

```
            S31,S32,S33,S34,S35,S36,S37,S38,S39,S3A,
            S41,S42,S43,S44,S45,S46,S47,S48,S49,S4A,
            S51,S52,S53,S54,S55,S56,S57,S58,S59,S5A,
            S61,S62,S63,S64,S65,S66,S67,S68,S69,S6A,
            S71,S72,S73,S74,S75,S76,S77,S78,S79,S7A,
            S81,S82,S83,S84,S85,S86,S87,S88,S89,S8A,
            S91,S92,S93,S94,S95,S96,S97,S98,S99,S9A,
            SA1,SA2,SA3,SA4,SA5,SA6,SA7,SA8,SA9,SAA],

    %     SIJ - czas startu dla J-tej operacji I-tego zadania

/*7*/    E = [E1,E2,E3,E4,E5,E6,E7,E8,E9,EA],
/*8*/    E :: 655..1000,
/*9*/    R = [1,1,1,1,1,1,1,1,1,1],

/*10*/     S11 :: 0..605,     /*11*/     S21 :: 0..490,
/*12*/     S31 :: 0..432,     /*13*/     S41 :: 0..345,
/*14*/     S51 :: 0..607,     /*15*/     S61 :: 0..504,
/*16*/     S71 :: 0..584,     /*17*/     S81 :: 0..461,
/*18*/     S91 :: 0..403,     /*19*/     SA1 :: 0..460,

/*20*/     S12 :: 29..634,    /*21*/     S22 :: 43..533,
/*22*/     S32 :: 91..523,    /*23*/     S42 :: 81..426,
/*24*/     S52 :: 14..621,    /*25*/     S62 :: 84..588,
/*26*/     S72 :: 46..630,    /*27*/     S82 :: 31..492,
/*28*/     S92 :: 76..479,    /*29*/     SA2 :: 85..370,

/*30*/     S13 :: 107..712,   /*31*/     S23 :: 133..628,
/*32*/     S33 :: 176..608,   /*33*/     S43 :: 176..521,
/*34*/     S53 :: 20..627,    /*35*/     S63 :: 86..590,
/*36*/     S73 :: 83..667,    /*37*/     S83 :: 117..578,
/*38*/     S93 :: 145..548,   /*39*/     SA3 :: 98..548,

/*40*/     S14 :: 116..721,   /*41*/     S24 :: 208..698,
/*42*/     S34 :: 215..647,   /*43*/     S44 :: 247..592,
/*44*/     S54 :: 42..649,    /*45*/     S64 :: 138..642,
          % korekta:                        % korekta:
/*46*/     S74 :: 100..450,   /*47*/     S84 :: 100..450,
/*48*/     S94 :: 221..624,   /*49*/     SA4 :: 159..619,


/*50*/     S15 ::152..757,   /*51*/     S25 ::219..709,
/*52*/     S35 ::289..721,   /*53*/     S45 ::346..691,
/*54*/     S55 ::103..710,   /*55*/     S65 ::233..737,
/*56*/     S75 ::157..741,   /*57*/     S85 ::237..698,
/*58*/     S95 ::272..675,   /*59*/     SA5 ::166..626,

/*60*/     S16 ::201..806,   /*61*/     S26 ::288..778,
/*62*/     S36 ::300..700,   /*63*/     S46 ::355..700,
```

```
/*64*/     S56 ::129..736,   /*65*/     S66 ::281..736,
/*66*/     S76 ::189..736,   /*67*/     S86 ::269..730,
/*68*/     S96 ::250..600,   /*69*/     SA6 ::230..690,


/*70*/     S17 :: 212..817,   /*71*/     S27 :: 316..806,
/*72*/     S37 :: 389..821,   /*73*/     S47 :: 407..821,
/*74*/     S57 :: 198..805,   /*75*/     S67 :: 353..857,
/*76*/     S77 :: 210..793,   /*77*/     S87 :: 357..818,
/*78*/     S97 :: 368..771,   /*79*/     SA7 :: 306..766,

/*80*/     S18 :: 274..879,   /*81*/     S28 :: 362..852,
/*82*/     S38 :: 401..833,   /*83*/     S48 :: 492..837,
          % korekta:
/*84*/     S58 :: 450..800,   /*85*/     S68 :: 400..904,
/*86*/     S78 :: 242..826,   /*87*/     S88 :: 376..837,
/*88*/     S98 :: 408..811,   /*89*/     SA8 :: 353..813,


/*90*/     S19 :: 330..935,   /*91*/     S29 :: 408..898,
/*92*/     S39 :: 490..922,   /*93*/     S49 :: 590..935,
/*94*/     S59 :: 268..875,   /*95*/     S69 :: 450..800,
/*96*/     S79 :: 450..800,   /*97*/     S89 :: 424..885,
/*98*/     S99 :: 497..900,   /*99*/     SA9 :: 405..865,

/*100*/    S1A :: 374..979,   /*101*/    S2A :: 480..970,
/*102*/    S3A :: 535..967,   /*103*/    S4A :: 612..957,
/*104*/    S5A :: 340..947,   /*105*/    S6A :: 471..975,
/*106*/    S7A :: 361..945,   /*107*/    S8A :: 460..921,
/*108*/    S9A :: 523..921,   /*109*/    SAA :: 495..955,

 %     Każda maszyna istnieje tylko w jednym egzemplarzu;
 %     dlatego może każdorazowo wykonywać tylko jedną operację:

 %     Maszyna  1 może w każdej chwili  wykonywać tylko jedną operację:
/*110*/    cumulative([S11,S21,S32,S43,S52,S67,S72,S82,S91,SA2],
              [29,43,85,71,6,47,37,86,76,13],R,1),
 %     Maszyna  2 może w każdej chwili  wykonywać tylko jedną operację:
/*111*/    cumulative([S12,S26,S31,S41,S53,S62,S71,S83,S92,SA1],
              [78,28,91,81,22,2,46,46,69,85],R,1),
 %     Maszyna  3 może w każdej chwili  wykonywać tylko jedną operację:
/*112*/    cumulative([S13,S22,S34,S42,S51,S61,S74,S81,S95,SA3],
              [9,90,74,95,14,84,13,31,85,61],R,1),
 %     Maszyna  4 może w każdej chwili  wykonywać tylko jedną operację:
/*113*/    cumulative([S14,S25,S33,S48,S55,S64,S73,S8A,S93,SA8],
              [36,69,39,98,26,95,61,79,76,52],R,1),
 %     Maszyna  5 może w każdej chwili  wykonywać tylko jedną operację:
/*114*/    cumulative([S15,S23,S3A,S44,S56,S69,S7A,S85,S99,SA9],
              [49,75,33,99,69,6,55,32,26,90],R,1),
```

```
 %     Maszyna  6 może w każdej chwili  wykonywać tylko jedną operację:
/*115*/    cumulative([S16,S28,S36,S4A,S54,S63,S76,S84,S94,SA7],
              [11,46,10,43,61,52,21,74,51,47],R,1),
 %     Maszyna  7 może w każdej chwili  wykonywać tylko jedną operację:
/*116*/    cumulative([S17,S27,S38,S45,S5A,S68,S75,S86,S97,SA4],
              [62,46,89,9,53,65,32,88,40,7],R,1),
 %     Maszyna  8 może w każdej chwili  wykonywać tylko jedną operację:
/*117*/    cumulative([S18,S29,S37,S47,S58,S6A,S79,S89,S98,SAA],
              [56,72,12,85,49,25,30,36,89,45],R,1),
 %     Maszyna  9 może w każdej chwili  wykonywać tylko jedną operację:
/*118*/    cumulative([S19,S2A,S35,S46,S57,S65,S78,S87,S9A,SA5],
              [44,30,90,52,21,48,89,19,74,64],R,1),
 %     Maszyna  10 może w każdej chwili  wykonywać tylko jedną operację:
/*119*/    cumulative([S1A,S24,S39,S49,S59,S66,S77,S88,S96,SA6],
              [21,11,45,22,72,72,32,48,11,76],R,1),

 %     Ograniczenia kolejności operacji: w ramach każdego zadania,
 %     operacja następna nie może wyprzedzać operacji poprzedniej:

/*120*/    S12 #>= S11+29,   /*121*/    S13 #>= S12+78,
/*122*/    S14 #>= S13+9,    /*123*/    S15 #>= S14+36,
/*124*/    S16 #>= S15+49,   /*125*/    S17 #>= S16+11,
/*126*/    S18 #>= S17+62,   /*127*/    S19 #>= S18+56,
/*128*/    S1A #>= S19+44,   /*129*/    E1  #>= S1A+21,

/*130*/    S22 #>= S21+43,   /*131*/    S23 #>= S22+90,
/*132*/    S24 #>= S23+75,   /*133*/    S25 #>= S24+11,
/*134*/    S26 #>= S25+69,   /*135*/    S27 #>= S26+28,
/*136*/    S28 #>= S27+46,   /*137*/    S29 #>= S28+46,
/*138*/    S2A #>= S29+72,   /*139*/    E2  #>= S2A+30,

/*140*/    S32 #>= S31+91,   /*141*/    S33 #>= S32+85,
/*142*/    S34 #>= S33+39,   /*143*/    S35 #>= S34+74,
/*144*/    S36 #>= S35+90,   /*145*/    S37 #>= S36+10,
/*146*/    S38 #>= S37+12,   /*147*/    S39 #>= S38+89,
/*148*/    S3A #>= S39+45,   /*149*/    E3  #>= S3A+33,

/*150*/    S42 #>= S41+81,   /*151*/    S43 #>= S42+95,
/*152*/    S44 #>= S43+71,   /*153*/    S45 #>= S44+99,
/*154*/    S46 #>= S45+9,    /*155*/    S47 #>= S46+52,
/*156*/    S48 #>= S47+85,   /*157*/    S49 #>= S48+98,
/*158*/    S4A #>= S49+22,   /*159*/    E4  #>= S4A+43,


/*160*/    S52 #>= S51+14,   /*161*/    S53 #>= S52+6,
/*162*/    S54 #>= S53+22,   /*163*/    S55 #>= S54+61,
/*164*/    S56 #>= S55+26,   /*165*/    S57 #>= S56+69,
/*166*/    S58 #>= S57+21,   /*167*/    S59 #>= S58+49,
/*168*/    S5A #>= S59+72,   /*169*/    E5  #>= S5A+53,
```

```
/*170*/    S62 #>= S61+84,   /*171*/    S63 #>= S62+2,
/*172*/    S64 #>= S63+52,   /*173*/    S65 #>= S64+95,
/*174*/    S66 #>= S65+48,   /*175*/    S67 #>= S66+72,
/*176*/    S68 #>= S67+47,   /*177*/    S69 #>= S68+65,
/*178*/    S6A #>= S69+6,    /*179*/    E6  #>= S6A+25,

/*180*/    S72 #>= S71+46,   /*181*/    S73 #>= S72+37,
/*182*/    S74 #>= S73+61,   /*183*/    S75 #>= S74+13,
/*184*/    S76 #>= S75+32,   /*185*/    S77 #>= S76+21,
/*186*/    S78 #>= S77+32,   /*187*/    S79 #>= S78+89,
/*188*/    S7A #>= S79+30,   /*189*/    E7  #>= S7A+55,

/*190*/    S82 #>= S81+31,   /*191*/    S83 #>= S82+86,
/*192*/    S84 #>= S83+46,   /*193*/    S85 #>= S84+74,
/*194*/    S86 #>= S85+32,   /*195*/    S87 #>= S86+88,
/*196*/    S88 #>= S87+19,   /*197*/    S89 #>= S88+48,
/*198*/    S8A #>= S89+36,   /*199*/    E8  #>= S8A+79,


/*200*/    S92 #>= S91+76,   /*201*/    S93 #>= S92+69,
/*202*/    S94 #>= S93+76,   /*203*/    S95 #>= S94+51,
/*204*/    S96 #>= S95+85,   /*205*/    S97 #>= S96+11,
/*206*/    S98 #>= S97+40,   /*207*/    S99 #>= S98+89,
/*208*/    S9A #>= S99+26,   /*209*/    E9  #>= S9A+74,

/*210*/    SA2 #>= SA1+85,   /*211*/    SA3 #>= SA2+13,
/*212*/    SA4 #>= SA3+61,   /*213*/    SA5 #>= SA4+7,
/*214*/    SA6 #>= SA5+64,   /*215*/    SA7 #>= SA6+76,
/*216*/    SA8 #>= SA7+47,   /*217*/    SA9 #>= SA8+52,
/*218*/    SAA #>= SA9+90,   /*219*/    EA  #>= SAA+45,


 %     Każde zadanie "istnieje" tylko w jednym egzemplarzu; dlatego
 %     każdorazowo można wykonywać tylko jedną  operację tego zadania:

     % dla zadania 1 można w każdej chwili wykonywać tylko jedną operację:
/*219*/    cumulative([S11,S12,S13,S14,S15,S16,S17,S18,S19,S1A],
              [29,78,9,36,49,11,62,56,44,21],R,1),
     % dla zadania 2 można w każdej chwili wykonywać tylko jedną operację:
/*220*/    cumulative([S21,S22,S23,S24,S25,S26,S27,S28,S29,S2A],
              [43,90,75,11,69,28,46,46,72,30],R,1),
     % dla zadania 3 można w każdej chwili wykonywać tylko jedną operację:
/*221*/    cumulative([S31,S32,S33,S34,S35,S36,S37,S38,S39,S3A],
              [91,85,39,74,90,10,12,89,45,33],R,1),
     % dla zadania 4 można w każdej chwili wykonywać tylko jedną operację:
/*222*/    cumulative([S41,S42,S43,S44,S45,S46,S47,S48,S49,S4A],
              [81,95,71,99,9,52,85,98,22,43],R,1),
     % dla zadania 5 można w każdej chwili wykonywać tylko jedną operację:
```

```
/*223*/    cumulative([S51,S52,S53,S54,S55,S56,S57,S58,S59,S5A],
               [14,6,22,61,26,69,21,49,72,53],R,1),
      % dla zadania 6 można w każdej chwili wykonywać tylko jedną operację:
/*224*/    cumulative([S61,S62,S63,S64,S65,S66,S67,S68,S69,S6A],
               [84,2,52,95,48,72,47,65,6,25],R,1),
      % dla zadania 7 można w każdej chwili wykonywać tylko jedną operację:
/*225*/    cumulative([S71,S72,S73,S74,S75,S76,S77,S78,S79,S7A],
               [46,37,61,13,32,21,32,89,30,55],R,1),
      % dla zadania 8 można w każdej chwili wykonywać tylko jedną operację:
/*226*/    cumulative([S81,S82,S83,S84,S85,S86,S87,S88,S89,S8A],
               [31,86,46,74,32,88,19,48,36,79],R,1),
      % dla zadania 9 można w każdej chwili wykonywać tylko jedną operację:
/*227*/    cumulative([S91,S92,S93,S94,S95,S96,S97,S98,S99,S9A],
               [76,69,76,51,85,11,40,89,26,74],R,1),
      % dla zadania 10 można w każdej chwili wykonywać tylko jedną operację:
/*228*/    cumulative([SA1,SA2,SA3,SA4,SA5,SA6,SA7,SA8,SA9,SAA],
               [85,13,61,7,64,76,47,52,90,45],R,1),

/*229*/    append(S,E,SE),
/*230*/    maxlist(E,M),

/*231*/    bb_min(moje_ukonkretnienie(SE), M, bb_options with
               [strategy:continue,from:900,to:930]),

/*232*/    write("E  = "),write(E),nl,
/*233*/    write("Minimalny makespan = "),write(M),nl,nl,

/*234*/    write("S11="),write(S11),write("  S12="),write(S12),
           write("  S13="),write(S13),write("  S14="),write(S14),
           write("  S15="),write(S15),nl,write("S16="),write(S16),
           write("  S17="),write(S17),write("  S18="),write(S18),
           write("  S19="),write(S19),write("  S1A="),write(S1A),
           nl,nl,

/*235*/    write("S21="),write(S21),write("  S22="),write(S22),
           write("  S23="),write(S23),write("  S24="),write(S24),
           write("  S25="),write(S25),nl,write("S26="),write(S26),
           write("  S27="),write(S27),write("  S28="),write(S28),
           write("  S29="),write(S29),write("  S2A="),write(S2A),
           nl,nl,

/*236*/    write("S31="),write(S31),write("  S32="),write(S32),
           write("  S33="),write(S33),write("  S34="),write(S34),
           write("  S35="),write(S35),nl,write("S36="),write(S36),
           write("  S37="),write(S37),write("  S38="),write(S38),
           write("  S39="),write(S39),write("  S3A="),write(S3A),
           nl,nl,

/*237*/    write("S41="),write(S41),write("  S42="),write(S42),
```

```
          write("  S43="),write(S43),write("  S44="),write(S44),
          write("  S45="),write(S45),nl,write("S46="),write(S46),
          write("  S47="),write(S47),write("  S48="),write(S48),
          write("  S49="),write(S49),write("  S4A="),write(S4A),
          nl,nl,

/*238*/   write("S51="),write(S51),write("  S52="),write(S52),
          write("  S53="),write(S53),write("  S54="),write(S54),
          write("  S55="),write(S55),nl,write("S56="),write(S56),
          write("  S57="),write(S57),write("  S58="),write(S58),
          write("  S59="),write(S59),write("  S5A="),write(S5A),
          nl,nl,

/*239*/   write("S61="),write(S61),write("  S62="),write(S62),
          write("  S63="),write(S63),write("  S64="),write(S64),
          write("  S65="),write(S65),nl,write("S66="),write(S66),
          write("  S67="),write(S67),write("  S68="),write(S68),
          write("  S69="),write(S69),write("  S6A="),write(S6A),
          nl,nl,

/*240*/   write("S71="),write(S71),write("  S72="),write(S72),
          write("  S73="),write(S73),write("  S74="),write(S74),
          write("  S75="),write(S75),nl,write("S76="),write(S76),
          write("  S77="),write(S77),write("  S78="),write(S78),
          write("  S79="),write(S79),write("  S7A="),write(S7A),
          nl,nl,

/*241*/   write("S81="),write(S81),write("  S82="),write(S82),
          write("  S83="),write(S83),write("  S84="),write(S84),
          write("  S85="),write(S85),nl,write("S86="),write(S86),
          write("  S87="),write(S87),write("  S88="),write(S88),
          write("  S89="),write(S89),write("  S8A="),write(S8A),
          nl,nl,

/*242*/   write("S91="),write(S92),write("  S92="),write(S92),
          write("  S93="),write(S93),write("  S94="),write(S94),
          write("  S95="),write(S95),nl,write("S96="),write(S96),
          write("  S97="),write(S97),write("  S98="),write(S98),
          write("  S99="),write(S99),write("  S9A="),write(S9A),
          nl,nl,

/*243*/   write("SA1="),write(SA1),write("  SA2="),write(SA2),
          write("  SA3="),write(SA3),write("  SA4="),write(SA4),
          write("  SA5="),write(SA5),nl,write("SA6="),write(SA6),
          write("  SA7="),write(SA7),write("  SA8="),write(SA8),
          write("  SA9="),write(SA9),write("  SAA="),write(SAA),nl.

/*244*/ moje_ukonkretnienie(Wszystkie_Zmienne):-
/*245*/   najpierw_srodek(Wszystkie_Zmienne,Wszystkie_Zmienne_Przemieszane),
```

```
/*246*/    ( fromto(Wszystkie_Zmienne_Przemieszane, Zmienne,
                            Zmienne_Pozostale, []) do
/*247*/    delete(Zmienna, Zmienne, Zmienne_Pozostale, 0, max_regret),
/*248*/    indomain(Zmienna,min)
/*249*/    ).

/*250*/ najpierw_srodek(Lista,Przemieszana):-
/*251*/    halve(Lista,F,B),
/*252*/    reverse(F,RF),
/*253*/    splice(B,RF,Przemieszana).
```

Program generuje następujący komunikat:

```
Found a solution with cost 930
Found no solution with cost 900.0 .. 929.0
E  = [908, 915, 920, 842, 895, 655, 753, 892, 792, 930]
Minimalny makespan = 930

S11=119  S12=445  S13=523  S14=532  S15=568
S16=617  S17=645  S18=721  S19=792  S1A=887

S21=76  S22=224  S23=355  S24=430  S25=568
S26=637  S27=707  S28=753  S29=813  S2A=885

S31=308  S32=408  S33=493  S34=532  S35=609
S36=699  S37=709  S38=753  S39=842  S3A=887

S41=0  S42=84  S43=185  S44=256  S45=359
S46=368  S47=420  S48=637  S49=766  S4A=799

S51=179  S52=256  S53=286  S54=308  S55=370
S56=430  S57=499  S58=530  S59=593  S5A=842

S61=0  S62=84  S63=86  S64=138  S65=233
S66=281  S67=361  S68=408  S69=499  S6A=505

S71=86  S72=148  S73=233  S74=314  S75=327
S76=421  S77=442  S78=520  S79=668  S7A=698

S81=193  S82=275  S83=399  S84=445  S85=519
S86=557  S87=699  S88=718  S89=777  S8A=813

S91=217  S92=217  S93=294  S94=370  S95=421
S96=506  S97=517  S98=579  S99=668  S9A=718

SA1=132  SA2=262  SA3=327  SA4=388  SA5=420
```

```
SA6=517  SA7=628  SA8=735  SA9=787  SAA=885
```

Komunikat ten jest skrajnie niekomunikatywny. Należy koniecznie przedstawić go w postaci odpowiednich dwóch wykresów Gantta, tak jak to zostało zrobione dla przykładu 6.11.
Rysunek 6.21 przedstawia wykres Gantta dla zadań tego benchmarku, a rysunek 6.22 stosowany sposób kodowania kolorów maszyn.

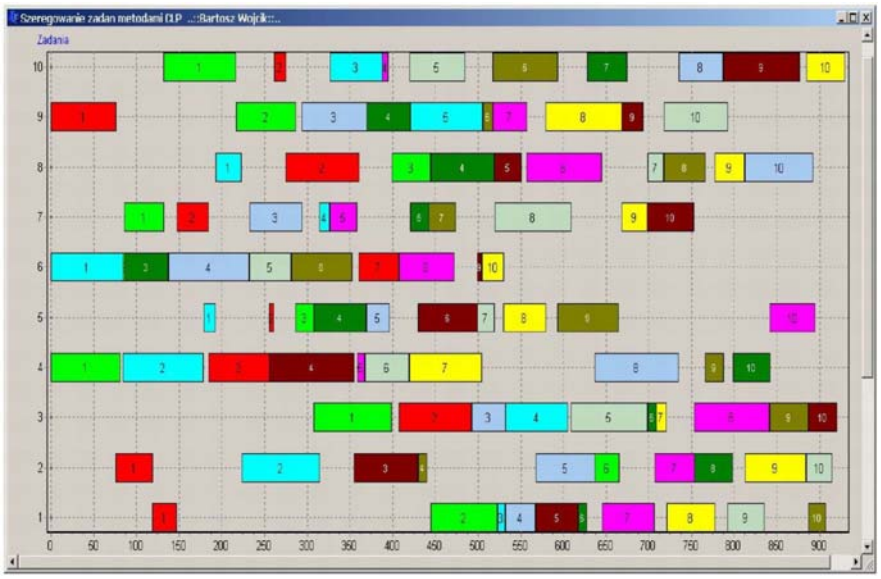
Na rysunku 6.21 trudno dostrzec operację 2 dla zadania 6, gdyż - jak wynika to z wiersza 170 programu, czas trwania tej operacji jest równy 2, co przy zastosowanej skali osi czasu jest nie do przedstawienia. Sens rysunku 6.21 jest oczywisty. I tak dla zadania 4 mamy kolejno operację wykonywaną przez maszynę jasno-zieloną (maszyna 2), maszynę jasno-niebieską (maszyna 3), maszynę czerwoną (maszyna 1) itd. itd.

Aby w przejrzysty sposób przedstawić wykres Gantta dla maszyn, należy niestety zrezygnować ze sposobu kodowania kolorów, zastosowanego dla wykresu Gantta dla zadań; w przeciwnym przpadku wykres dla każdej maszyny zawierałby *klocki* tego samego koloru, co czyniłoby go maksymalnie nieczytelnym. Dlatego też zastosowano odmienny sposób kolorowania. Rysunek 6.23 przedstawia wykres Gantta dla maszyn tego benchmarku, a rysunek 6.24 stosowany sposób kodowania kolorów zadań[6].

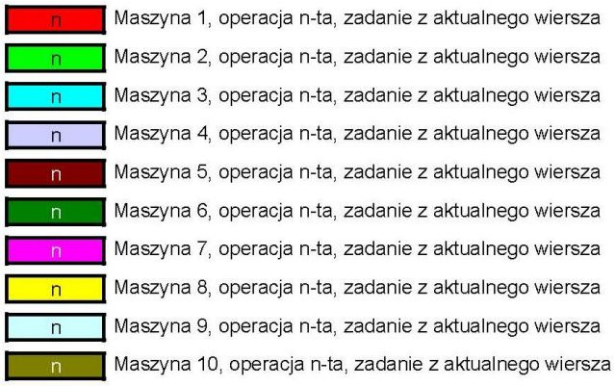
Sens rysunku 6.23 jest następujący: np. *klocek* czerwony dla maszyny 1 jest operacją 1 dla zadania 1 z rysunku 6.21 (również *klocek* czerwony), *klocek* czerwony dla maszyny 2 jest operacją 2 dla zadania 1 z rysunku 6.21 (*klocek* jasnozielony), *klocek* czerwony (bardzo wąski) dla maszyny 3 odpowiada operacji 3 dla zadania 1 z rysunku 6.21, przedstawionej również za pomocą bardzo wąskiego *klocka* jasno-niebieskiego, itd. itd.

Problemy harmonogramowania należą do najtrudniejszych problemów kombinatorycznych. Zarazem ich praktyczne znaczenia jest ogromne. Techniki rozwiązywania tych problemów przeszły sporą ewolucję, od technik klasycznych, w dużej mierze heurystycznych (patrz [Muth-63]), lecz ciągle jeszcze stosowanych (patrz [Baker-09]), do zyskujących coraz bardziej na znaczeniu technik programowania z ograniczeniami, patrz [Baptiste-95] i [Baptiste-01].

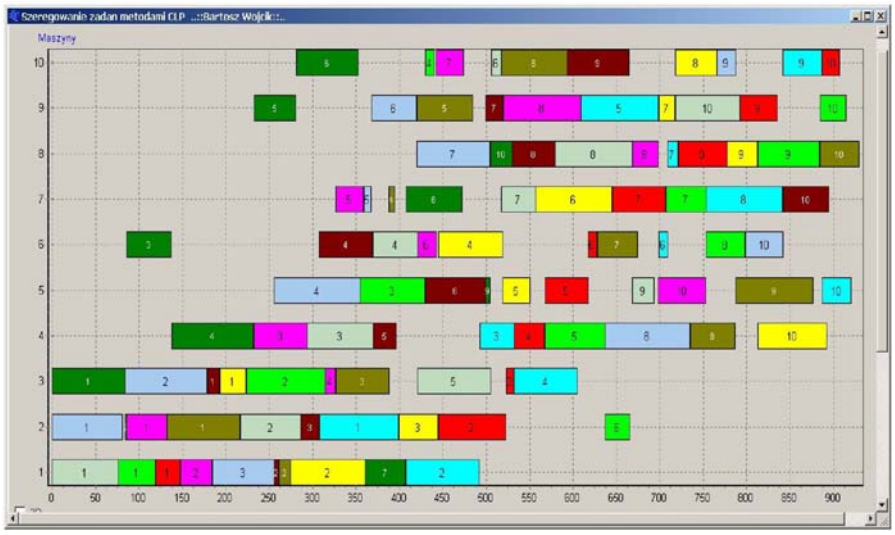
[6]Wykresy z rysunków 6.21 i 6.23 zostały wygenerowane za pomocą programu wykonanego w ramach pracy dyplomowej [Wójcik-05].



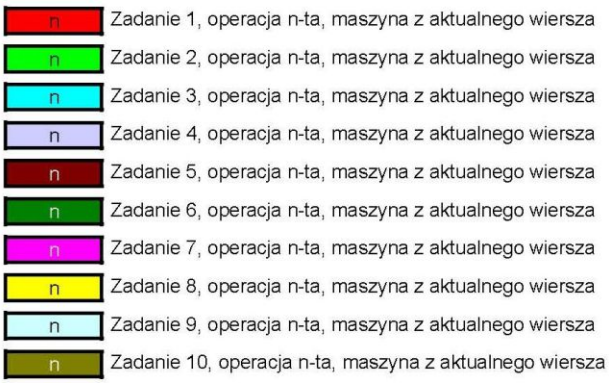
Rysunek 6.21: Wykres Gantta dla zadań benchmarku MT10



Rysunek 6.22: Kodowanie kolorów maszyn dla wykresu Gantta zadań



Rysunek 6.23: Wykres Gantta dla maszyn benchmarku MT10



Rysunek 6.24: Kodowanie kolorów zadań dla wykresu Gantta maszyn

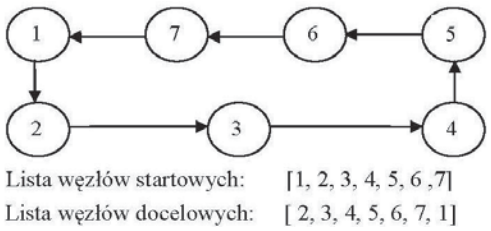
6.19 Problem komiwojażera

Problem komiwojażera (ang. *Traveling Salesman Problem* (*TSP*)) można sformułować następująco: komiwojażer mieszkający w jakimś mieście (np. mieście

`1`) musi udać się do każdego z miast `2,3,...,n` tylko raz, po czym wrócić do miasta `1`. Komiwojażer chciałby to zrobić najbardziej efektywnie, tzn. minimalizując całkowitą przebytą drogę. Niestety, nie istnieje ogólny (dla dowolnego `n`) i zarazem efektywny (tzn. działający w czasie będącym wielomianem zmiennej `n`) algorytm rozwiązania tego problemu. Problem ten słynie z groźnej eksplozji kombinatorycznej (albo jak to teoretycy wolą nazywać - jest problemem NP-trudnym). Gdyby chcieć go rozwiązywać metodą przeglądu zupełnego, to drugie miasto można wybrać na $(n-1)$ sposobów, trzecie - na $(n-2)$ sposobów, a więc dla miasta drugiego i trzeciego mamy $(n-1)\times(n-2)$ możliwości wyboru, przy czwartym mieście możliwości wyboru jest $(n-1)\times(n-2)\times(n-3)$ itd. A więc rozwiązywanie problemu komiwojażera metodą przeglądu zupełnego może wymagać (przy założeniu, że odległość od miasta `i` do miasta `j` jest różna od odległości od miasta `j` do miasta `i`) w najgorszym przypadku aż $(n-1)!$ obliczeń długości tras, co można zrobić dla niewięcej niż kilkunastu miast. Obecnie jednak znane są zarówno podejścia heurystyczne jak i dokładne, rozwiązujące problem komiwojażera na drodze obliczeń równoległych dla dziesiątek tysięcy miast.

6.19.1 Cykl Hamiltona

Podstawowym pojęciem problemu komiwojażera jest pojęcie cyklu Hamiltona. *Cyklem Hamiltona* nazywa się cykl przechodzący przez każdy węzeł dokładnie jeden raz. Cyklem jest każdy graf tworzący *drogę zamkniętą*. Ilustruje to rysunek 6.25.



Rysunek 6.25: Graf bedący cyklem Hamiltona dla węzłów 1,2,3,4,5,6,7.

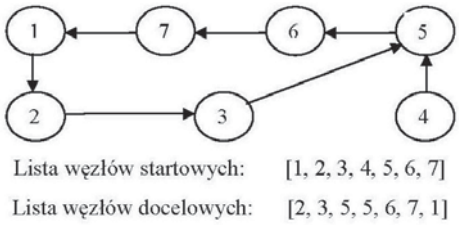
Cykle takie są doskonałymi modelami problemu komiwojażera, który chce odwiedzić wszystkie wybrane miasta tylko raz i powrócić do miasta, z którego wyjechał. Miasta są wierzchołkami grafu, a jego krawędzie to drogi między nimi.

Dla potrzeb *CLP* można opisać cykl Hamiltona dwiema listami. Są nimi:

1. *Lista wezłów startowych* będąca listą numerów wszystkich węzłów grafu; każdy węzeł występuje w niej tylko jeden raz, a dla przejrzystości zapisu dobrze jest, by lista była uporządkowana, tzn. numery wezłów w liście były kolejnymi liczbami całkowitymi.

2. *Lista wezłów docelowych* będąca listą numerów węzłów odwiedzanych z węzłów znajdujących się na tych samych pozycjach w liście węzłów startowych. Tak np. na rysunku 6.25, węzeł `3` jest węzłem docelowym dla węzła startowego `2`.

Dla lepszego zilustrowania tych pojęć przedstawiono na rysunku 6.26 graf nie będący cyklem Hamiltona: jego *lista wezłów docelowych* nie jest permutacją *listy węzłów startowych*.



Rysunek 6.26: Graf nie będący cyklem Hamiltona dla węzłów 1,2,3,4,5,6,7.

Program `6_18_hamilton.ecl`[7] można zastosować do weryfikacji obydwu grafów stosując predykat standardowy `circuit/1`:

```
/*1*/    :-lib(ic).

/*2*/  top:-
/*3*/        circuit([2, 3, 4, 5, 6, 7, 1]),
/*4*/        ~(circuit([2, 3, 5, 5, 6, 7, 1])).
 % ~+Cel jest efektywnym operatorem  negacji, który oczekuje aż +Cel zostanie uziemiony.

/*5*/  circuit(Lista_Wezlow_Docelowych):-
/*6*/        length(Lista_Wezlow_Docelowych,Liczba_Wezlow),
/*7*/        dim(Tabela_Wezlow_Docelowych,[Liczba_Wezlow]),
```

[7]Ten program i program definujący standardowy predykat circuit/1 generujący cykl Hamiltona w zorientowanym grafie są autorstwa dr inż. Łukasza Domagały.

```
/*8*/        Tabela_Wezlow_Docelowych=..[[]|Lista_Wezlow_Docelowych],
/*9*/        (
/*10*/       count(Nr_Wezla_Startowego,1,Liczba_Wezlow),
/*11*/       param(Tabela_Wezlow_Docelowych,Liczba_Wezlow)
/*12*/       do
/*13*/       arg(Nr_Wezla_Startowego,Tabela_Wezlow_Docelowych,Wezel_Docelowy),
/*14*/       Dlugosc_Cyklu is Liczba_Wezlow -2 ,
/*15*/          (
/*16*/          count(_,1,Dlugosc_Cyklu),
/*17*/          fromto(Wezel_Docelowy,Wezel_Docelowy_We,Wezel_Docelowy_Wy,_),
/*18*/          param(Nr_Wezla_Startowego,Tabela_Wezlow_Docelowych)
/*19*/          do
/*20*/          element_tablicy(Wezel_Docelowy_We, Tabela_Wezlow_Docelowych,
                          Wezel_Docelowy_Wy),
/*21*/          Wezel_Docelowy_Wy #\= Nr_Wezla_Startowego
/*22*/          )
/*23*/       ).

/*24*/  element_tablicy(Index,Array,Value):-
/*25*/       (
/*26*/       ground(Index)->
/*27*/       arg(Index,Array,Value)
/*28*/       ;
/*29*/       suspend(
/*30*/       arg(Index,Array,Value),
/*31*/       0,
/*32*/       [Index->inst],
/*33*/       _ThisSusp
/*34*/           )
/*35*/       ).
```

Program generuje komunikat `Yes` oznaczający, dla (domyślnej) listy węzłów startowych:

`[1, 2, 3, 4, 5, 6, 7]`, że lista węzłów docelowych:

`[2, 3, 4, 5, 6, 7, 1]` jest cyklem Hamiltona, a lista węzłów docelowych:

`[2, 3, 5, 5, 6, 7, 1]` nie jest cyklem Hamiltona.

6.19.2 Harmonogramowanie linii produkcyjnej

Problem komiwojażera jest modelem dla szeregu aplikacji nie mających nic wspólnego z komiwojażerowaniem. Spotyka się je w planowaniu i harmonogramowaniu różnych instalacji, w produkcji mikrochipów i nawet w sekwencjono-

waniu DNA. W tych aplikacjach pojęcie *miasta* (lub ogólnie - *węzła*) może np. odpowiadać różnym ustawieniom instalacji produkcyjnej, punktom lutowania lub odcinkom DNA, natomiast pojęcie *odległości* może odpowiadać czasom lub kosztom przestrojenia instalacji, czasom ruchu lutownicy pomiędzy punktami lutowania lub miarom podobieństwa odcinków DNA.

Rozpocznijmy małym problemem przestrajania instalacji petrochemicznej. Instalacja może produkować dowolne z pośród siedmiu paliw, jeżeli została odpowiednio nastrojona. Czas przestrojenia instalacji zależy od kolejności, w jakiej paliwa są produkowane. W pełnym cyklu produkcyjnym produkowane jest - w jakiejś kolejności - każde paliwo. Dla takiej sytuacji czasy nieprodukcyjne (czasy przestrajania) są przedstawione w tablicy 6.4[8].

Paliwo	1	2	3	4	5	6	7
Diesel 1	0	30	67	50	60	70	90
Regular 2	20	0	88	43	39	11	74
Premium 3	47	88	0	42	32	20	47
Ethanol_5% 4	38	43	62	0	41	59	57
Racing 5	46	39	32	41	0	52	29
Unleaded 6	40	11	20	59	52	0	69
Aviation 7	30	45	37	40	19	55	0

Tablica 6.4: Czasy przestrajania instalacji petrochemicznej przy zmianie produktu

Sens danych zamieszczonych w tej tablicy jest następujący: np. przestrojenie instalacji produkującej *Premium* na instalację produkujacą *Aviation* wymaga 47 jednostek czasu, natomiast przestrojenie instalacji produkującej *Aviation* na instalację produkującą *Premium* wymaga 37 jednostek czasu. Stąd wynika istnienia optymalnej kolejności przestrajania instalacji, minimalizującej całkowity czas przestrajania. Podobne problemy występują w innych instalacjach przemysłowych, np. w lakierniach karoserii samochodowych w fabrykach samochodów.

Problem przestrajania rozwiązuje program `6_19_TSP_small.ecl` korzystający z predykatu globalnego *circuit/1* zawartego w module `circuit.ecl`:

```
/*1*/ :-use_module(circuit).
/*2*/ :-lib(ic).
```

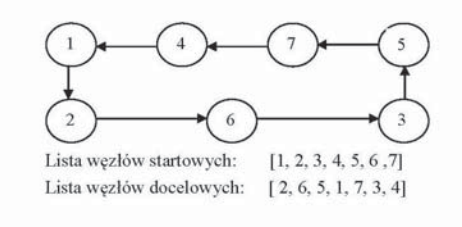
[8]Inspiracją dla tego przykładu był prostszy przykład z książki [Baker-09], gdzie. przedstawiono rozwiązanie stosujące klasyczne techniki BO.

```
/*3*/ :-lib(branch_and_bound).
 /*4*/    top:-
/*5*/        Lista_Wezlow_Docelowych = [X1, X2, X3, X4, X5, X6, X7],
          %  Xi  - numer  instalacji nastrajanej po ukończeniu produkcji
          %  z instalacją o numerze i
/*6*/        Lista_Wezlow_Docelowych :: 1..7,

/*7*/        element(X1, [ 0, 30, 67, 50, 60, 70, 90], C1),
/*8*/        element(X2, [20,  0, 88, 43, 39, 11, 74], C2),
/*9*/        element(X3, [47, 88,  0, 42, 32, 20, 47], C3),
/*10*/       element(X4, [38, 43, 62,  0, 41, 59, 57], C4),
/*11*/       element(X5, [46, 39, 32, 41,  0, 52, 29], C5),
/*12*/       element(X6, [40, 11, 20, 59, 52, 0,  69], C6),
/*13*/       element(X7, [30, 45, 37, 40, 19, 55,  0], C7),

/*14*/       circuit(Lista_Wezlow_Docelowych),

/*15*/       Suma_czasow_przestrojenia #= C1+C2+C3+C4+C5+C6+C7,
/*16*/       Szukaj=search(Lista_Wezlow_Docelowych, 0, most_constrained,
                    indomain_split, complete, []),
/*17*/       BBOptions=bb_options{strategy:dichotomic, timeout:_},
/*18*/       bb_min(Szukaj, Suma_czasow_przestrojenia, BBOptions),
/*19*/       writeln(" Startowa lista instalacji":[1, 2, 3, 4, 5, 6, 7]),
/*20*/       writeln("  Docelowa lista instalacji":[X1,X2,X3,X4,X5,X6,X7]),
/*21*/       writeln(" Minimalny calkowity czas przestrajania ":
                          Suma_czasow_przestrojenia).
```

Program generuje następujący komunikat:

```
Found a solution with cost 352
Found no solution with cost 0.0 .. 176.0
Found a solution with cost 263
Found a solution with cost 203
Found no solution with cost 176.0 .. 189.5
Found no solution with cost 189.5 .. 196.25
Found no solution with cost 196.25 .. 199.625
Found a solution with cost 200
Startowa lista instalacji: [1,2,3,4,5,6,7]
Docelowa lista instalacji: [2,6,5,1,7,3,4]
Minimalny calkowity czas przestrajania: 200
```

Rozwiązanie odpowiada cyklowi Hamiltona z rysunku 6.27.



Rysunek 6.27: Cykl Hamiltona dla optymalnej sekwencji przestrajania.

6.19.3 Marszrutyzacja dla komiwojażera

Rozpatrzmy bardziej złożony problem komiwojażera odwiedzającego stolice dzielnic Absurdolandii. Próba rozwiązania tego problemu za pomocą stosowanego dotąd `6_19_TSP_small.ecl` nie jest dobrym pomysłem: program ten dla tego problemu będzie bardzo mało efektywny, co wynika głównie ze złych właściwości propagacyjnych wielokrotnie stosowanego predykatu standardowego `element/3`. Bardziej wydajnym programem jest `6_20_TSP_large.ecl`, dla którego ponownie korzysta się z modułu `circuit.ecl`:

```
/*1*/ :-use_module(circuit).
/*2*/ :-lib(ic).
/*3*/ :-lib(branch_and_bound).
 /*4*/ :-lib(ic_global).

/*5*/    macierz_odleglosci(Macierz_odleglosci):-
/*6*/       Macierz_odleglosci=[](

                                %Docelowe stolice dzielnic:
             %   1   2   3   4   5   6   7   8   9   10  11  11  13  14  15  16
% Startowe
% stolice
% dzielnic:
/*7,  1*/     [](  0,384,484,214,234,267,524,656,446,371,459,561,585,683,634,751),
/*8,  2*/     [](384,  0,156,411,296,167,339,379,340,432,485,545,483,500,565,642),
/*9,  3*/     [](484,156,  0,453,323,217,213,223,281,442,452,479,394,370,500,516),
/*10, 4*/     [](214,411,453,  0,130,259,413,601,303,157,245,356,422,542,427,585),
/*11, 5*/     [](234,296,323,130,  0,129,310,491,212,178,261,335,354,465,403,517),
/*12, 6*/     [](267,167,217,259,129,  0,255,389,205,265,318,391,348,421,430,516),
/*13, 7*/     [](524,339,213,413,310,255,  0,188,134,344,319,297,181,161,295,303),
/*14, 8*/     [](656,379,223,601,491,389,188,  0,322,532,507,485,363,260,477,430),
/*15, 9*/     [](446,340,281,303,212,205,134,322,  0,204,181,196,143,242,220,306),
```

```
/*16, 10*/    [](371,432,442,157,178,265,344,532,204,  0, 86,199,300,428,268,433),
/*17, 11*/    [](459,485,452,245,261,318,319,507,181, 86,  0,113,220,382,182,347),
/*18, 12*/    [](561,545,479,356,335,391,297,485,196,199,113,  0,156,323, 75,244),
/*19, 13*/    [](585,483,394,422,354,348,181,363,143,300,220,156,  0,167,114,163),
/*20, 14*/    [](683,500,370,542,465,421,161,260,242,428,382,323,167,  0,269,170),
/*21, 15*/    [](634,565,500,427,403,430,295,477,220,268,182, 75,114,269,  0,165),
/*22, 16*/    [](751,642,516,585,517,516,303,430,306,433,347,244,163,170,165,  0)
/*23*/      ).

/*24*/   top:-
/*25*/      macierz_odleglosci(Macierz_odleglosci),
/*26*/      dim(Macierz_odleglosci,[Liczba_miast,Liczba_miast]),

     %  Każdemu miastu docelowemu odpowiada zmienna, której dziedziną
     %  są numery wszystkich miast:
/*27*/      length(Lista_Miast_Docelowych,Liczba_miast),
/*28*/      Lista_Miast_Docelowych#::1..Liczba_miast,

/*29*/      (foreach(Miasto_Docelowe,Lista_Miast_Docelowych),
     %  Konstruuj tabelę odległości i listę odległości dla par
     %  Miasto_Startowe - Miasto_Docelowe:
/*30*/      count(Miasto_Startowe,1,Liczba_miast),
     %  Konstruuj listę wszystkich odległości:
/*31*/      foreach(Odleglosc,Lista_odleglosci),
     %  Konstruuj listę wszystkich numerów miast startowych:
/*32*/      foreach(Miasto_Startowe,Lista_Miast_Startowych),
/*33*/      param(Macierz_odleglosci) do
     %  Miasto docelowe musi być różne od miasta startowego:
/*34*/      Miasto_Docelowe#\=Miasto_Startowe,
     %  Definiuj odległość pomiędzy miastem startowym a miastem docelowym:
/*35*/      arg(Miasto_Startowe,Macierz_odleglosci,Tablica_odleglosci),
/*36*/      Tablica_odleglosci=..[[]|Lista_odleglosci],
/*37*/      element(Miasto_Docelowe, Lista_odleglosci, Odleglosc)
/*38*/      ),

     %  Każde miasto docelowe należy odwiedzić tylko raz:
/*39*/      ic_global:    alldifferent(Lista_Miast_Docelowych),
     %  Ta implementacja alldifferent/1 ma silniejsze właściwosci propagacyjne
     %  aniżeli stosowana poprzednio ogólna implementacja alldifferent/1.

     %  Dla każdego miasta startowego musi istnieć miasto docelowe:
/*40*/      sorted(Lista_Miast_Docelowych, Lista_Miast_Startowych),

     %  Definiuj sumę odległości pomiędzy odpowiadającymi sobie miastami
     %  startowymi i docelowymi:
/*41*/      sumlist(Lista_odleglosci,Suma_odleglosci),

/*42*/      circuit(Lista_Miast_Docelowych),
```

```
/*43*/      Szukaj=search(Lista_Miast_Docelowych, 0, most_constrained,
                        indomain_split, complete, []),
/*44*/      BBOptions=bb_options{strategy:dichotomic, timeout:_},
/*45*/      bb_min(Szukaj, Suma_odleglosci, BBOptions),

/*46*/      write("Calkowita droga = "),writeln(Suma_odleglosci),
/*47*/      write("Stolice startowe = "), writeln([1,2,3,4,5,6,7,8,9,10,
                        11,12,13,14,15,16]),
/*48*/      write("Stolice docelowe = "),writeln(Lista_Miast_Docelowych).
```

Program rozwiązuje problem w czasie `1.75` sekund i generuje komunikat:

```
Found a solution with cost 3928
Found a solution with cost 3021
Found a solution with cost 2565
Found no solution with cost 2130.0 .. 2347.5
Found no solution with cost 2347.5 .. 2456.25
Found no solution with cost 2456.25 .. 2510.625
Found a solution with cost 2521
Found no solution with cost 2510.625 .. 2515.8125
Found no solution with cost 2515.8125 .. 2518.40625
Found no solution with cost 2518.40625 .. 2519.703125
Found no solution with cost 2519.703125 .. 2520.3515625
Calkowita droga = 2521

Stolice startowe:   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
Stolice docelowe:   [4, 6, 2,10, 1, 5, 8, 3, 7, 11, 12, 15,  9, 13, 16, 14]
Search time = 1.75
```

Optymalny cykl Hamiltona jest pokazany na rysunku 6.28.

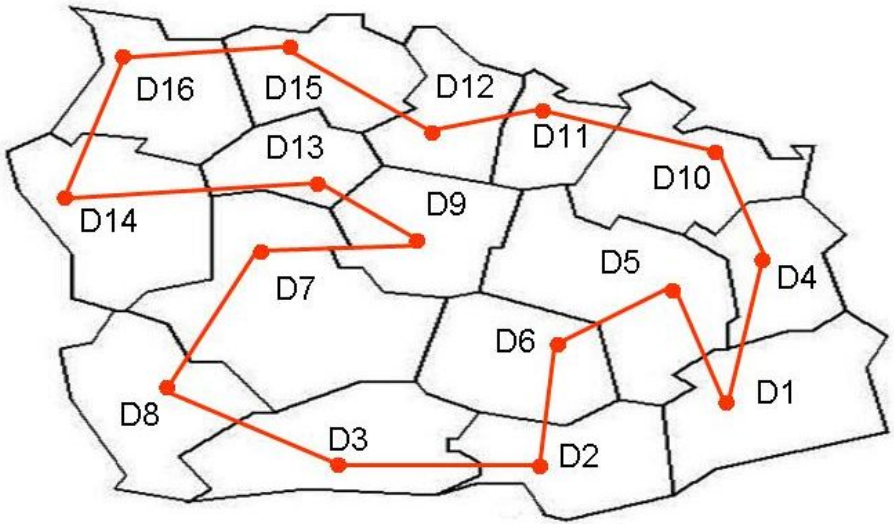
Właściwości propagacyjne programu można nadal poprawiać stosując predykat globalny `cycle/3`[9]:

```
cycle(+Lista_Miast_Docelowych,++Macierz_odleglosci,
            -Suma_odleglosci)
```

Lepszą propagację ilustruje przykład `6_19_TSP_with_cycle.ecl`, dla którego macierz odległości jest w module `macierz_odleglosci.clp`:

```
/*1*/ :-use_module(macierz_odleglosci).
/*2*/ :-lib(ic).
```

[9]Predykat ten został opracowany przez dr inż. Łukasza Domagałę.



Rysunek 6.28: Cykl Hamiltona rozwiązania problemu TSP dla stolic dzielnic Absurdolandii.

```
/*3*/ :-lib(branch_and_bound).
 /*4*/ :-lib(cycle).

/*5*/ top:-
/*6*/      macierz_odleglosci(Macierz_odleglosci),
/*7*/      dim(Macierz_odleglosci,[Liczba_miast,Liczba_miast]),
/*8*/      length(Lista_Miast_Docelowych,Liczba_miast),
/*9*/      Lista_Miast_Docelowych#::1..Liczba_miast,

/*10*/     cycle(Lista_Miast_Docelowych,Macierz_odleglosci,Suma_odleglosci),

/*11*/     cputime(Czas_startu),
/*11*/     Szukaj=search(Lista_Miast_Docelowych, 0, most_constrained,
                indomain_max, complete, []),
/*12*/     bb_min(Szukaj, Suma_odleglosci, bb_options{strategy:dichotomic}),
/*13*/     cputime(Czas_konca),
/*14*/     Czas_poszukiwania is Czas_konca - Czas_startu,

/*15*/     write("Calkowita droga = "),writeln(Suma_odleglosci),
/*16*/     write("Stolice startowe = "), writeln([1,2,3,4,5,6,7,8,9,10,
```

```
                    11,12,13,14,15,16]),
/*17*/     write("Stolice docelowe = "),writeln(Lista_Miast_Docelowych),
/*18*/     write("Czas_poszukiwania = "),writeln(Czas_poszukiwania).
```

Tym razem program daje rozwiązanie po czasie `0.906` sekund i generuje komunikat:

```
Found a solution with cost 4914
Found a solution with cost 3701
Found a solution with cost 3072
Found a solution with cost 2781
Found a solution with cost 2644
Found a solution with cost 2521
Calkowita droga = 2521
Stolice startowe = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
Stolice docelowe = [4, 6, 2,10, 1, 5, 8, 3, 7, 11, 12, 15,  9, 13, 16, 14]
Czas_poszukiwania = 0.906
```

6.20 Dodatki

Sposób definiowania modułów deklarowanych w przykładach `6_19_TSP_small.ecl` i `6_21_TSP_with_cycle.ecl` został przedstawiony poniżej.

6.20.1 "circuit.ecl" jako moduł

```
/*1*/:- module(circuit).
/*2*/:- export(circuit/1).
/*3*/:- lib(ic).
/*4*/  circuit(Lista_Wezlow_Docelowych):-
/*5*/          length(Lista_Wezlow_Docelowych,Liczba_Wezlow),
/*6*/          dim(Tabela_Wezlow_Docelowych,[Liczba_Wezlow]),
/*7*/          Tabela_Wezlow_Docelowych=..[[]|Lista_Wezlow_Docelowych],
/*8*/          (
/*9*/          count(Nr_Wezla_Startowego,1,Liczba_Wezlow),
/*10*/         param(Tabela_Wezlow_Docelowych,Liczba_Wezlow)
/*11*/         do
/*12*/         arg(Nr_Wezla_Startowego,Tabela_Wezlow_Docelowych,Wezel_Docelowy),
/*13*/         Dlugosc_Cyklu is Liczba_Wezlow -2 ,
/*14*/         (
/*15*/         count(_,1,Dlugosc_Cyklu),
```

```
/*16*/         fromto(Wezel_Docelowy,Wezel_Docelowy_We,Wezel_Docelowy_Wy,_),
/*17*/         param(Nr_Wezla_Startowego,Tabela_Wezlow_Docelowych)
/*18*/         do
/*19*/         arr_element(Wezel_Docelowy_We, Tabela_Wezlow_Docelowych,
                       Wezel_Docelowy_Wy),
/*20*/         Wezel_Docelowy_Wy #\= Nr_Wezla_Startowego
/*21*/         )
/*22*/         ).

/*23*/  arr_element(Indeks,Tabela,Liczba):-
/*24*/         (
/*25*/         ground(Indeks)->
/*26*/         arg(Indeks,Tabela,Liczba)
/*27*/         ;
/*28*/         suspend(
/*29*/         arg(Indeks,Tabela,Liczba),
/*30*/         0,
/*31*/         [Indeks->inst],
/*32*/         _To_Zawies
/*33*/         )
/*34*/         ).
```

6.20.2 ”macierzOdleglosci.ecl” jako moduł

```
:- module(macierzOdleglosci).
:- export(macierzOdleglosci/1).
:- lib(ic).

/*1*/ macierzOdleglosci(Macierz_odleglosci):-
/*2*/ Macierz_odleglosci=[](
     [](  0,384,484,214,234,267,524,656,446,371,459,561,585,683,634,751),
     [](384,  0,156,411,296,167,339,379,340,432,485,545,483,500,565,642),
     [](484,156,  0,453,323,217,213,223,281,442,452,479,394,370,500,516),
     [](214,411,453,  0,130,259,413,601,303,157,245,356,422,542,427,585),
     [](234,296,323,130,  0,129,310,491,212,178,261,335,354,465,403,517),
     [](267,167,217,259,129,  0,255,389,205,265,318,391,348,421,430,516),
     [](524,339,213,413,310,255,  0,188,134,344,319,297,181,161,295,303),
     [](656,379,223,601,491,389,188,  0,322,532,507,485,363,260,477,430),
     [](446,340,281,303,212,205,134,322,  0,204,181,196,143,242,220,306),
     [](371,432,442,157,178,265,344,532,204,  0, 86,199,300,428,268,433),
```

```
[](459,485,452,245,261,318,319,507,181, 86,  0,113,220,382,182,347),
[](561,545,479,356,335,391,297,485,196,199,113,  0,156,323, 75,244),
[](585,483,394,422,354,348,181,363,143,300,220,156,  0,167,114,163),
[](683,500,370,542,465,421,161,260,242,428,382,323,167,  0,269,170),
[](634,565,500,427,403,430,295,477,220,268,182, 75,114,269,  0,165),
[](751,642,516,585,517,516,303,430,306,433,347,244,163,170,165,  0)
).
```

6.21 Zadania

Proste harmonogramowanie

Na czterech identycznych maszynach należy wykonać siedem operacji o czasach trwania z tablicy 6.5[10]. Napisz program minimalizujący długość uszeregowania przy założeniu braku ograniczeń kolejnościowych.

Operacja	1	2	3	4	5	6	7
Czas trwania	3	3	3	1	1	1	4

Tablica 6.5: Czasy trwania operacji

Mniej proste harmonogramowanie

Trzy maszyny, jedna typu M1 i dwie typu M2, mają wykonać cztery zadania Za, Zb, Zc i Zd. Zadania są różne i wymagają wykonania jednej lub więcej operacji, w kolejności określonej przez numery zadań w tablicy 6.6.

Sens danych w tablicy jest następujący: np. operacja Ob3 może rozpocząć się dopiero po zakończeniu operacji Ob2. Napisz program minimalizujący długość uszeregowania.

Pięć operacji

Każda z pięciu operacji jest scharakteryzowana czasem dostępu do przetwarzanego surowca, czasem trwania operacji i czasem dostarczenia produktu przetwarzania, zgodnie z tablicą 6.7[11]. Czas dostępu jest czasem, od którego poczynając surowiec jest dostepny. Czas dostarczenia produktu przetwarzania jest czasem, do którego najpóźniej produkt musi być dostarczony odbiorcy, np. innej maszynie.

[10] Zob. [Baker-09].
[11] Zob. [Baker-09].

Zadania	Operacje	Maszyny	Czasy trwania
Za	Oa1	M1	2
Za	Oa2	M2	6
Zb	Ob1	M2	5
Zb	Ob2	M1	3
Zb	Ob3	M2	3
Zc	Oc	M2	4
Zd	Od1	M1	5
Zd	Od2	M2	2

Tablica 6.6: Dane dla mniej prostego harmonogramowania

Operacja	1	2	3	4	5
Czas dostępu	0	2	3	0	6
Czas trwania	2	1	2	3	2
Czas dostarczenia	5	2	6	3	1

Tablica 6.7: Dane dla pięciu operacji

Napisz program minimalizujący długość uszeregowania przy założeniu braku ograniczeń kolejnościowych.

Ciepłe grzanki z masłem [12]

Stary opiekacz grzanek ma dwoje drzwiczek na zawiasach. Może opiekać każdorazowo dwa kawałki chleba, każdy tylko z jednej strony. Czasy trwania czynności są następujące: 1)potrzeba 10 sekund dla opieczenia jednego kawałka chleba z jednej strony (pamiętaj, że równocześnie można opiekać dwa kawałki chleba), 2)potrzeba 3 sekund na włożenie kawałka chleba do opiekacza, 3)potrzeba 3 sekund na wyjęcia kawałka chleba z opiekacza, 4)potrzeba 3 sekund na obrócenie kawałka chleba w opiekacz bez jego wyjmowania. 5) potrzeba 12 sekund na posmarowanie jednej strony grzanki masłem. Ponadto czynności wkładania, obracania, wyjmowania i smarowania masłem wymagają obydwu rąk, a więc nie można ich wykonywać równocześnie. Każdy kawałek chleba jest smarowany masłem tylko z jednej strony, mianowicie tej, która już została opieczona. Na początku trzy kawałki chleba leżą obok opiekacza, a na końcu trzy kawałki grzanek (obustronnie opiekanych, jednostronnie posmarowanych masłem) powinny

[12] Przykład z [Michalewicz-07].

również leżeć obok opiekacza. Napisz program wyjaśniający, jak to zrobić w najkrótszym czasie.

Przeprawa przez most [13]

Czterech podróżników A, B, C i D musi przejść mostem nad głębokim wąwozem. Jest ciemna noc, a podróżnicy mają tylko jedną lampę naftową. Lampa jest niezbędna dla poruszania się po moście, który jest stary i pełen dziur oraz luźnych desek. Ponadto, jego konstrukcja jest osłabiona tak, że każdorazowo może po nim się poruszać tylko dwóch podróżników. Okazuje się, że każdy podróżnik przejdzie przez most w innym czasie. Pan A jest młody i silny i potrzebuje tylko minuty na przejście mostu. Pan D jest starszym panem, któremu ostatnio wymieniono staw biodrowy i dlatego potrzebuje aż dziesięciu minut na przejście mostu. Panowie B i C potrzebują odpowiednio dwóch i pięciu minut by przejść most. A ponieważ każdy podróżnik musi korzystać z naftowej lampy, to najwolniejszy podróżnik z każdej pary określa czas przejścia mostu przez parę. Napisz program wyznaczający harmonogram przejść minimalizujący całkowity czas pokonywania mostu przez wszystkich podróżników.

Wiercenie płytek

Producent obwodów drukowanych stosuje programowalne wiertarki dla wywiercania sześciu otworów w każdym obwodzie. Współrzędne `x` i `y` każdego otworu przedstawia tablica 6.8[14].

x	y	Otwór
1	2	1
3	1	2
5	3	3
7	2	4
8	3	5

Tablica 6.8: Współrzędne otworów obwodu drukowanego

Napisz program wyznaczający kolejność wiercenia otworów minimalizującą całkowitą drogę ruchu wiertarki.

[13] Przykład z [Michalewicz-07].
[14] Zob. [Winston-94].

Projekt

Tablica 6.9[15] przedstawia dane małego projektu, dla którego współdzielony resurs (liczba pracowników) nie może przekroczyć 5.

Operacja	Czas trwania	Poprzednicy	Wymagany resurs
A	6	-	2
B	8	-	3
C	4	-	3
D	4	A	4
E	4	A	2
F	12	B,E	3
G	14	B,E	1
H	6	B,C,E	4
I	8	D,F	2
J	16	D,F,G	1
K	2	D,F,G	1
L	12	H,K	3

Tablica 6.9: Dane projektu

Napisz program minimalizujący długość uszeregowania tego projektu.

Cztery zadania

Na pewnej maszynie należy wykonać cztery zadania. Czasy potrzebne dla wykonania tych zadań i terminy ich wymagalności przedstawia tablica 6.10[16]. *Terminem wymagalności* nazywa się czas, w którym oczekiwane jest ukończenie zadania.

Numer zadania	Czas trwania (dni)	Termin wymagalności
1	6	Koniec dnia 8
2	4	Koniec dnia 4
3	5	Koniec dnia 12
4	8	Koniec dnia 16

Tablica 6.10: Czasy trwania i termin wymagalności

[15] Przykład zaczerpniety z książki [Baker-09].
[16] Zob. [Winston-94].

Opóźnieniem zadania nazywa się liczbę dni po upływie terminu wymagalności, potrzebną dla ukończenia zadania. Opóźnienie uważa się za równe zero jeżeli zadanie zostało zakończone przed terminem wymagalności lub w terminie wymagalności. Napisz program wyznaczający harmonogram wykonywania zadań w taki sposób, by zminimalizować całkowite opóźnienie wszystkich czterech zadań.

Trzy zadania [17]

JobCo stosuje jedną maszynę dla trzech różnych zadań, których dane przedstawia tablica 6.11.

Numer zadania	Czas trwania (dni)	Termin wymagalności (dni)	Kara za opóźnienie (JP/dzień)
1	5	Koniec dnia 25	19
2	20	Koniec dnia 20	12
3	15	Koniec dnia 35	34

Tablica 6.11: Czasy trwania, termin wymagalności i kary za opóźnienie

Napisz program wyznaczający harmonogram wykonywania zadań w taki sposób, by zminimalizować całkowitą karę za opóźnienie.

Benchmark ABZ5

Sprawdź, czy dla benchmarku ABZ5 z rysunku 6.29 istnieje rozwiązanie dopuszczalne. Jeżeli tak, napisz program minimalizujący długość uszeregowania dla tego benchmarku.

Likwidacja szkód samochodwych

Jaś Kowalski jest likwidatorem szkód samochodowych. Mieszka w Rybniku przy ul. Lompy 8. Codziennie odwiedza kilka miast województwa śląskiego, gdzie ogląda i fotografuje szkody samochodowe, wycenia je i sporządza protokoły szkód. Najbliższego dnia roboczego ma ocenić szkody samochodowe pod następującymi adresami:

Gliwice, ul. Pszczyńska 114
Pyskowice, ul. Toszecka 85
Zabrze, ul. Wolności 115
Bytom, ul. Kolejowa 5
Sosnowiec, ul. Objazd 3

[17] Zob. [Taha-08].

	Op. 0		Op. 1		Op. 2		Op. 3		Op. 4		Op. 5		Op. 6		Op. 7		Op. 8		Op. 9	
	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T
Zad. 0	4	88	8	68	6	94	5	99	1	67	2	89	9	77	7	99	0	86	3	92
Zad. 1	5	72	3	50	6	69	4	75	2	94	8	66	0	92	1	82	7	94	9	63
Zad. 2	9	83	8	61	0	83	1	65	6	64	5	85	7	78	4	85	2	55	3	77
Zad. 3	7	94	2	68	1	61	4	99	3	54	6	75	5	66	0	76	9	63	8	67
Zad. 4	3	69	4	88	9	82	8	95	0	99	2	67	6	95	5	68	7	67	1	86
Zad. 5	1	99	4	81	5	64	6	66	8	80	2	80	7	69	9	62	3	79	0	88
Zad. 6	7	50	1	86	4	97	3	96	0	95	8	97	2	66	5	99	6	52	9	71
Zad. 7	4	98	6	73	3	82	2	51	1	71	5	94	7	85	0	62	8	95	9	79
Zad. 8	0	94	6	71	3	81	7	85	1	66	2	90	4	76	5	58	8	93	9	97
Zad. 9	3	50	0	59	1	82	8	67	7	56	9	96	6	58	4	81	5	59	2	96

Rysunek 6.29: Funkcje maszyn (M) i czasów operacji (T) benchmarku ABZ5

Tychy, ul. Rybna 20
Ruda Śląska, ul. Nowa 7
Żory, ul. Ogrodowa 6.

Napisz program wyznaczający najkrótszą trasę odwiedzin szkód, jeżeli po obejrzeniu wszystkich szkód Jaś Kowalski zechce wrócić do domu. Potrzebną macierz odległości wyznacz korzystając z Google Mapy. Macierz ta niekoniecznie musi być symetryczna. Narysuj najkrótszą trasę na mapie województwa śląskiego. Zakładając, że Jaś Kowalski wyjeżdża z domu o godz. 6.00, jeździ z średnia prędkością 80 km/godz., a średni czas oględzin szkody to 40 min., wyznacz godziny i minuty, o których może się pojawić pod każdym z wymienionych adresów.

Rozdział 7

CLP dla zmiennych ciągłych

7.1 Uwagi wstępne

Wszystkie rozpatrywane dotąd przykłady miały dyskretne zmienne decyzyjne o skończonych dziedzinach. Generowane w trakcie poszukiwań drzewa miały więc z natury swej skończoną głębokość, a przestrzenie poszukiwań miały skończoną liczbę punktów, które można było przeglądać uziemiając zmienne tak, by spełnione były ograniczenia.

Platforma ECL^iPS^e umożliwia również rozwiązywanie problemów spełniania ograniczeń i problemów optymalnego spełniania ograniczeń o zmiennych ciągłych, a więc mających ciągłe dziedziny, czyli zawierające nieskończenie wiele punktów; przykładem może być dziedzina $0 \leqslant X \leqslant 150$:

1. *Ciągłe Problemy Spełniania Ograniczeń*, oznaczane skrótem *CPSO*, można zdefiniować następująco:

   - dany jest skończony zbiór S zmiennych decyzyjnych $X_1, X_2, ..., X_n$, przyjmujących wartości z ciągłych *dziedzin* $D_1, D_2, ..., D_n$, gdzie $D_i = Max_i \diamond X_i \diamond Min_i$, a $\diamond \varepsilon (<, \leqslant, >, \geqslant)$;
   - dany jest zbiór ograniczeń pomiędzy zmiennymi decyzyjnymi. *Ograniczeniem i-tym* $C_i(X_{i_1}, X_{i_2}, ..., X_{i_k})$ pomiędzy k zmiennymi ze zbioru S nazywa się *relację* będącą takim podzbiorem iloczynu kartezjańskiego $D_{i_1} \times D_{i_2} \times, ..., \times D_{i_k}$, który określa wartości zmiennych

pasujące do siebie w sensie zdefiniowanym przez rozpatrywany problem. Relacja ta jest z reguły definiowana za pomocą równań lub nierówności;

- *rozwiązaniem* CPSO nazywa się każde takie przyporządkowanie wszystkim zmiennym przedziałów wartości z ich dziedzin, które spełniają wszystkie ograniczenia. Wynika to z niemożności stosowania dla zmiennych ciągłych metod stosowanych w rozdziałach 2,...,6: głębokość drzew poszukiwań byłaby wówczas nieskończona, a ich przeszukiwanie trwałoby również nieskończenie długo. Dlatego też propagacja ograniczeń dla dziedzin ciągłych polega na ich sukcesywnym zawężaniu, np. najpierw do $10.78 \leqslant X \leqslant 90.52$, potem do $50.124 \leqslant X \leqslant 70.513$, itd. Zawężanie takie nie doprowadza nigdy do "pojedynczego" punktu, lecz wyłącznie do odpowiednio małego przedziału. Dlatego też rozwiązania tych problemów są pierwotnie otrzymywane w postaci przedziałowej:

  ```
  X = Kres_dolny__Kres_górny,
  ```

  np.:

  ```
  X = 36345.099404108__36345.099448266
  ```

  zawierającej dwie wartości rzeczywiste rozdzielone dwoma podkreślnikami `__`.

  Bywają również zapisywane w postaci:

  ```
  X{Kres_dolny .. Kres_górny}
  ```

  np.:

  ```
  X{36345.099404108 .. 36345.099448266},
  ```

  zawierającej dwie wartości rzeczywiste rozdzielone dwoma kropkami `...` Aparatem umożliwiającym uzyskanie takich wyników jest *arytmetyka przedziałowa*[1].

[1] Arytmetyka przedziałowa - (ang. *interval arithmetic*) w odróżnieniu od klasycznej arytmetyki definiującej działania na liczbach - definiuje działania na przedziałach liczb. Wynikiem działań składających się na propagację ograniczeń nie będzie więc zbiór uzgodnionych wartości zmiennych decyzyjnych, lecz zbiór uzgodnionych przedziałów zmiennych decyzyjnych

Tak więc rozwiązaniem problemu *CPSO* jest zbiór przedziałów zmiennych decyzyjnych, będących podprzedziałami pierwotnych dziedzin tych zmiennych, spełniających wszystkie ograniczenia.

Podobnie jak dyskretne *PSO*, *CPSO* można klasyfikować jako problemy wyznaczania stanów dopuszczalnych lub problemy wyznaczania trajektorii dopuszczalnych. Dla *CPSO* nie ma potrzeby stosowania biblioteki *eplex*; biblioteka *ic* całkowicie wystarcza. Należy jednak baczyć, by symbole operacji arytmetycznych i relacyjnych dla zmiennych ciągłych były prefiksowane symbolem `$`.

2. *Ciągłe Problemy Optymalnego Spełniania Ograniczeń*, oznaczane skrótem *CPOSO*, można zdefiniować następująco:

   - dany jest skończony zbiór S zmiennych $X_1, X_2, ..., X_n$, przyjmujących wartości ze standardowej ciągłej dziedziny `0.0..1.0Inf`;
   - dany jest zbiór ograniczeń pomiędzy zmiennymi decyzyjnymi. *Ograniczeniem i-tym* $C_i(X_{i_1}, X_{i_2}, ..., X_{i_k})$ pomiędzy k zmiennymi ze zbioru S nazywa się *relację* będącą takim podzbiorem iloczynu kartezjańskiego $D_{i_1} \times D_{i_2} \times, ..., \times D_{i_k}$, który określa wartości zmiennych *pasujące* do siebie w sensie zdefiniowanym przez rozpatrywany problem. Relacja ta jest z reguły definiowana za pomocą równań lub nierówności;
   - dany jest *wskaźnik jakości*, będący liniową funkcja zmiennych decyzyjnych, o współczynnikach rzeczywistych lub całkowitych, dla którego należy wyznaczyć maksimum lub minimum przez odpowiedni wybór wartości zmiennych decyzyjnych.

   Podobnie jak dyskretne *POSO*, *CPOSO* można klasyfikować jako problemy wyznaczania stanów optymalnych lub problemy wyznaczania trajektorii optymalnych. Platforma ECL^iPS^e została wyposażona w inkrementalne przedziałowe solwery równań liniowych, programowania liniowego i programowania mieszanego. Stosowana w tym celu arytmetyka przedziałowa jest tak "gęsta", że przy zwykłej dla ECL^iPS^e liczbie pozycji po przecinku, przedziały redukują się do pojedynczego punktu. Solwery te można stosować korzystając z ECL^iPS^e-owej biblioteki *eplex*. Biblioteka ta umożliwia bezpośrednie tworzenie interfejsów do komercyjnych solwerów *XPRESS-MP* firmy *Dash Optimization* lub *CPLEX* firmy *ILOG*.

Aczkolwiek obydwie wymienione firmy dostarczają darmowe wersje akademickie, w omawianych przykładach będą stosowane wyłącznie solwery dostarczane z platformą ECL^iPS^e i uruchamiane za pośrednictwem biblioteki *eplex*.

7.2 Przekleństwo i błogosławieństwo procentu składanego

7.2.1 Podstawy

Rozważania na temat zastosowań ECL^iPS^e CPS dla ciągłych zmiennych decyzyjnych dobrze zacząć do przykładów spełniania ograniczeń. Klasycznymi i pouczającymi przykładami tego typu są przykłady wymagające obliczenia procentu składanego.

Zacznijmy od przypomnienia podstawowych pojęć. Załóżmy, że z kapitału początkowego K_o robimy lokatę oprocentowaną z roczną stopą procentową $0 < s < 1$, automatycznie wznawianą co roku. Po pierwszym roku mamy na lokacie $K_1 = K_o + sK_o = K_o(1 + s)$. Po drugim roku mamy na lokacie $K_2 = K_1 + sK_1 = K_1(1 + s) = K_o(1 + s)^2$. Po n latach mamy na lokacie kwotę $K_n = K_o(1 + s)^n$. Jeżeli natomiast pożyczylibyśmy w banku kwotę K_o z tym samym oprocentowaniem, to po n latach będziemy zadłużeni na tej samej zasadzie na kwotę $K_n = K_o(1 + s)^n$.

Co to znaczy? Jeżeli np. w pierwszym roku naszej ery nasi pra-pra-...pra przodkowie pożyczyli jedną jednostkę pieniężną na `1%` rocznie i zapomnieli dług spłacić, a bank przetrwał wszystkie zawieruchy dziejowe i skrupulatnie śledził następstwo pokoleń i aktualizował zadłużenie, nasi przodkowie zaś przejmowali kolejne spadki z pełnym dobrodziejstwem inwentarza, to my - jako spadkobiercy - jesteśmy po 2010 latach zadłużeni na 485245261.49 owych jednostek pieniężnych (słownie: czterysta osiemdziesiąt pięć milionów...)[2].

[2]Być może właśnie z powodu takiej "eksplozji" zadłużenia politycy nie przejmują się spłatą długu narodowego, spokojnie czekając aż przyjmie on astronomiczne rozmiary, uniemożliwiające jego spłatę. Być może dlatego banki uważnie "hodują" zadłużenie kredytobiorców, by wybrać taki moment windykacji, w którym dług będzie już duży, ale wciąż spłacalny.

7.2.2 Obliczanie procentu składanego w CLP

Mariott [Marriott-98] przedstawił bardzo użyteczną rekurencyjną definicję procentu składanego, z której licznych wariantów będziemy poniżej korzystać. Pierwszym przykładem jest program `7_1_procent_skladany.ecl` program:

```
/*1*/ :- lib(ic).
/*2*/ top :-
/*3*/      Lista = [PV,T,I],
/*4*/      Lista :: 0.0..1000000.0,
/*5*/      T  $=  24,    % Liczba okresów
/*6*/      I  $=  8/100, % Procent  za okres
/*7*/      PV  $=  1000,  % 'Present Value' - wartość obecna kapitału
/*8*/      procent_skladany(PV,T,I).

/*12*/ procent_skladany(PV,T,I):-
/*13*/      T  $>=  1,   % Ciągle pozostają jeszcze jakieś okresy procentowania
/*14*/      NT  $=  T-1, % ale z każdym okresem jest o okres mniej.
/*15*/      FV  $=  PV + (PV * I), % 'Future Value' - wartość przyszła jest z
                                                    % każdym okresem aktualizowana
/*16*/      procent_skladany(FV,NT,I).
/*17*/ procent_skladany(FV,T,_):- %
/*18*/      T  $=  0,
/*10*/       write("Przyszla wartosc kapitalu= "),write(FV), nl.
```

Istotą tego programu jest rekurencyjna definicja `procent_skladany/3` za pomocą tegoż predykatu `procent_skladany/3` z aktualizowaną liczbą okresów i aktualizowaną wartością kapitału. Warunkiem końcowym tej rekurencji jest osiągnięcie zerowej liczby okresów. Wynik uzyskuje sie w arytmetyce przedziałowej:

```
Przyszla wartosc kapitalu = 6341.18073133441__6341.18073724012
```

A więc dziedzina zmiennej `Przyszla wartosc kapitalu` została zredukowana do odpowiednio małego przedziału.

7.2.3 Na emeryturę jako milioner - 1

Rozpatrzmy następujący przykład: załóżmy, że mając lat 20 postanawiamy pójść na emeryturę w wieku 65 lat jako milioner. Ile w tym celu trzeba wpłacić

jednorazowo w wieku 20 lat na nasze (prywatne) konto emerytalne oprocentowane na 6% rocznie, by cel swój osiągnąć?
Rozwiązanie przedstawia program 7_2_milioner_1.ecl:

```
/*1*/  :- lib(ic).
/*2*/  top :-
/*3*/     LD = [K,T,I],
/*4*/     LD :: 0.0..100000,
/*5*/     T $= 45.0,   % Liczba lat  oszczędzania
/*6*/     I $= 6/100,  % Stopa procentowa za rok
/*7*/     emerytura(K,T,I),
/*8*/     write("Pierwsza i jedyna wplata = "), write(K), nl.

/*9*/  emerytura(K,T,I) :-
/*10*/    T  $>= 1.0,
/*11*/    NK $= K + (K * I), % Nowy stan konta
/*12*/    NT $= T-1,  % Roczne malenie lat spłaty
/*13*/    emerytura(NK, NT, I).

/*14*/ emerytura(K,T,_):-
/*15*/    T $= 0.0,
/*16*/    K $= 1000000.  % Stan konta po 45 latach
```

Program generuje komunikat:

```
Pierwsza i jedyna wplata = 72650.0743490985__72650.0743562801
```

A więc jedna jedyna wpłata 72650 JP w wieku 20 lat zapewni nam - przy procencie składanym 6% rocznie - emeryturę w wysokości miliona JP w wieku 65 lat.

Najważniejszą częścią powyższego programu jest ponownie elegancka definicja rekurencyjna w liniach /*9*/,...,/*16*/: z każdym obiegiem tej rekurencji maleje liczba okresów oszczędzania o 1 i odpowiednio wzrasta stan naszego konta, aż do wyzerowania liczby okresów spłaty (linia /*15*/), kiedy stan naszego konta powinien osiągnąć wartość 1000000, patrz linia /*15*/.

7.2.4 Na emeryturę jako milioner - 2

Załóżmy obecnie, że - aby przejść w wieku 65 lat na emeryturę jako milioner - nie dokonamy jednorazowej wpłaty w wieku lat 20, lecz będziemy systematycznie, co roku wpłacać przez 45 lat określoną jednakową ratę na nasze (prywatne)

konto emerytalne oprocentowane na 6% rocznie. Jak duża musi być ta rata? Rozwiązanie przedstawia program `7_3_milioner_2.ecl`, w którym wstępnie założyliśmy, że rata R nie będzie większa niż 4701:

```
/*1*/  :- lib(ic).
/*2*/  top :-
/*3*/     LD = [K,T,I,R],
/*4*/     LD :: 0.0..1000000,
/*5*/     K $= 0.0,   % Stan początkowy konta
/*6*/     T $= 45.0,  % Liczba okresów oszczędzania
/*7*/     I $= 6/100, % Stopa procentowa za rok
/*8*/     R $=< 4701, % Roczna rata oszczędności, wyznaczona na drodze
                      %  sukcesywnego zacieśnienia ograniczenia
/*9*/     emerytura(K,T,I,R),
/*10*/    write("Rata roczna = "), write(R), nl.

/*11*/ emerytura(K,T,I,R) :-
/*12*/    T  $>= 1.0,
/*13*/    NK $= K + (K * I) + R, % Roczny przyrost na koncie
/*14*/    NT $= T-1, % Roczne malenie lat spłaty
/*15*/    emerytura(NK, NT,I,R).

/*16*/ emerytura(K,T,_,_):-
/*17*/    T $= 0.0,
/*18*/    K $= 1000000.0.
```

Program generuje komunikat:

```
There are 44 delayed goals.
Yes (0.02s cpu, solution 1, maybe more)
Rata roczna = __12477{4696.74977810691 .. 4701.0},
```

z którego wynika, że wartość raty jest gdzieś w powyższym przedziale, pomiędzy (zaokrąglając) 4696.75 a 4701.

Wynik jest szokujący: wpłacając (na nasze prywatne konto emerytalne oprocentowane na 6% rocznie) trochę mniej niż 4701 JP rocznie (co odpowiada po zaokrągleniu 392 JP miesięcznie), możemy po 45 latach przejść na emeryturę mając na swym koncie cały milion jednostek pieniężnych, przy czym łącznie wypłaciliśmy bankowi 45 * 4701 = 211545 JP. Gdyby tak rządy przestały się o nas troszczyć!

Ale wróćmy do naszego programu: predykat `emerytura(K,T,I,R)` ma w porównaniu z poprzednią definicją z rozdziału 7.2.3 o jeden argument więcej. Jest

nim coroczna rata R, której wartość zgodnie z linią /*13*/ corocznie powiększa stan naszego konta.

Wątpliwości może również budzić linia /*8*/ , gdzie "pomogliśmy" znaleźć rozwiązanie na drodze kilkukrotnego zmniejszenia ograniczenia.

7.2.5 Ach te kredyty hipoteczne!

Wzięliśmy kredyt hipoteczny i będziemy go spłacać przez najbliższe 24 lata w wysokości 12000 JP/rok, przy rocznym oprocentowaniu kredytu 8%. Jaka właściwie była wielkość naszego kredytu? Wyjaśnia to program 7_4_hipoteka.ecl:

```
/*1*/  :- lib(ic).
/*2*/  top :-
/*3*/     Lista = [K,T,S,R],
/*4*/     Lista :: 0.0..1000000.0,
/*5*/     T $= 24.0,          % Liczba okresów spłaty
/*6*/     S $= 8/100,         % Stopa procentowa za rok
/*7*/     R $= 12000.0,       % Rata spłaty za okres
/*8*/     kredyt_hipoteczny(K,T,S,R),
/*9*/     Koszt $= T * R,
/*10*/    write("Wysokosc kredytu = "),write(K),nl,
/*11*/    write("Koszt kredytu = "),write(Koszt),nl.

/*12*/ kredyt_hipoteczny(K,T,S,R):-
/*13*/    T $>= 1.0,  % Ciągle pozostały jakieś okresy spłaty,
/*14*/    NT $= T-1,  % ale z roku na rok maleją o 1
/*15*/    NK $= K + (K * S) - R, % Roczny stan zadłużenia
/*16*/    kredyt_hipoteczny(NK,NT,S,R).
/*17*/ kredyt_hipoteczny(K,T,_,_):-  % uf! - nareszcie koniec udręki!
/*18*/    T $= 0,
/*19*/    K $= 0.
```

Program generuje komunikat:

```
Wysokosc kredytu = 126345.099404108__126345.099448266
Koszt kredytu = 288000.0__288000.0
```

A więc za kredyt hipoteczny w wysokości (w zaokrągleniu) 126346 JP zapłacimy w ciągu 24 lat 288000 JP. Nic lepiej nie obrazuje prawidłowości "Czas to pieniądz!"

Linie `/*12*/`,...,`/*19*/` przedstawiają tym razem rekurencyjne malenie naszego zadłużenia w wyniku corocznych spłat rat, aż do finalnego wyzerowania się naszego zadłużenia. Zgodnie z linią `/*15*/` wartość kredytu, który ciągle pozostaje nam do spłacenia, corocznie wzrasta o składową wynikającą z oprocentowania, a maleje o składową równą płaconej racie.

7.2.6 Wartość bieżąca netto: ile zyskujemy lub tracimy?

Podejmując decyzje biznesowe dobrze jest pamiętać o straconych możliwościach. Straconych - gdyż nie możemy z nich korzystać właśnie z powodu owych decyzji biznesowych. Aby jednak rzetelnie zbilansować to co zyskaliśmy (lub straciliśmy), dobrze jest jednak owe stracone możliwości wziąć pod uwagę. Do tego służy pojęcie *wartości bieżącej netto* (ang. *net present value*, w skrócie NPV), które ocenia przyszłe zyski (lub straty) w kategoriach ich wartości aktualnej (zwanej w dalszym ciągu bieżącą), z uwzględnieniem najbardziej oczywistych (a więc najbardziej pewnych) straconych możliwości. Podstawą obliczeń NPV jest fakt zmiany wartości pieniądza w czasie, wynikający z możliwości intratnego inwestowania. A więc n.p. m JP uzyskanych (lub straconych) za rok ma inną wartość bieżącą, równą:

$$\frac{m}{(1+r)}$$

i zwaną *wartością bieżącą netto* owych m JP uzyskanych (lub straconych) za rok. To znaczy, że $\frac{m}{(1+r)}$ JP zainwestowanych obecnie da nam na pewno m JP za rok, gdzie r jest *roczną stopą dyskonta*. Słowa "na pewno" oznaczają istnienie *produktu rynkowego*, n.p. obligacji rządowych, gwarantującego (w danym okresie czasu) uzyskanie rocznej stopy zwrotu równej r. Z prostego uogólnienia wynika, że $\frac{m}{(1+r)^k}$ JP zainwestowanych obecnie da (z pewnością) m JP k lat później. Pojęcie wartości bieżącej netto można uogólnić dla dowolnych przyszłych przepływów pieniężnych; dlatego bywa stosowane dla oceny celowości różnych inwestycji. Rozpatrzmy przykład następujących dwóch inwestycji[3]:

1. Inwestycja A wymaga wyłożenia 8 milionów JP w chwili obecnej, przyniesie zysk 26 milionów JP za rok i wymaga dodatkowo wyłożenia za dwa lata 18 milionów JP dla usunięcia szkód wyrządzonych środowisku w wyniku tej inwestycji. Przepływ pieniężny netto (w milionach JP) dla tej

[3] Jest to zmodyfikowana wersja przykładu z książki [Winston-94].

inwestycji jest równy:

$$-8 + 26 - 18 = 0,$$

co - na pierwszy rzut oka - nie wygląda zbyt zachęcająco. Załóżmy istnienie bonów skarbu państwa gwarantujących uzyskanie rocznej stopy zwrotu równej 0.25. Wówczas NPV inwestycji A będzie (w milionach JP) równe:

$$-8 + \frac{26}{(1+0.25)} - \frac{18}{(1+0.25)^2} = -8 + 20.8 - 11.52 = 1.2,$$

co nie jest złym wynikiem, a bierze się stąd, że owe 18 milionów JP, które będziemy musieli wyłożyć za dwa lata, to obecnie tylko $\frac{18}{(1+0.25)^2} = 11.52$ milionów JP, które - po zakupie odpowiedniej liczby bonów skarby państwa - dadzą nam za dwa lata potrzebne wówczas 18 milionów JP. A więc inwestycja A zwiększy wartość bieżącą naszej firmy o 1.2 miliona JP.

2. Inwestycja B wymaga wyłożenia 6 milionów JP w chwili obecnej, przyniesie zysk 8 milionów JP za rok i wymaga dodatkowo wyłożenia za dwa lata 1 miliona JP dla usunięcia szkód wyrządzonych środowisku w wyniku tej inwestycji. Przepływ pieniężny netto (w milionach JP) dla tej inwestycji jest dodatni i równy:

$$-6 + 8 - 1 = 1,$$

co - na pierwszy rzut oka - wygląda nieźle. Załóżmy ponownie istnienie bonów skarbu państwa gwarantujących uzyskanie rocznej stopy zwrotu równej 0.25. Wówczas NPV inwestycji B będzie (w milionach JP) ujemne i równe:

$$-6 + \frac{8}{(1+0.25)} - \frac{1}{(1+0.25)^2} = -6 + 6.4 - 0.65 = -0.25,$$

co bierze się stąd, że ów 1 milion JP, który będziemy musieli wyłożyć za dwa lata, to obecnie aż $\frac{1}{(1+0.25)^2} = 0.65$ miliona JP, które - po zakupie odpowiedniej liczby bonów skarby państwa - dadzą nam za dwa lata potrzebny wówczas 1 milion JP. Niestety, jest to więcej niż 0.4 miliona zysku za pierwszy rok inwestycji. A więc inwestycja B zmniejszy wartość bieżącą naszej firmy o 0.25 miliona JP.

Następujący przykład[4] ilustruje zastosowanie pojęcia NPV:

[4]Temat przykładu pochodzi z książki [Winston-94].

Star Oil Company analizuje pięć różnych opcji inwestycyjnych. Wydatki inwestycyjne i wartości NPV (w milionach JP) są dla tych opcji zestawione w tabeli 7.1:

Parametry finansowe	Inv. 1	Inv. 2	Inv. 3	Inv. 4	Inv. 5
Inwestycje obecne (miliony JP)	11	53	5	5	29
Inwestycje za rok (miliony JP)	3	6	5	1	14
NPV	13	10	16	14	39

Tablica 7.1: Parametry finansowe opcji inwestycyjnych

Star Oil dysponuje w chwili obecnej 40 milionami JP, które mogą zostać zainwestowane, i sądzi że za rok będzie dysponować 20 milion JP na inwestycje. Star Oil może zakupić dowolny ułamek każdej inwestycji z tabeli 7.1, przy odpowiednim dopasowaniu wartości inwestycji i wartości NPV. Np. kupując jedną piątą inwestycji `3` należy obecnie wyłożyć $\frac{1}{5}5 = 1$ milion JP, a za rok $\frac{1}{5}5 = 1$ milion JP . Jedna piąta inwestycji `3` skutkuje wartością NPV równą $\frac{1}{5}16 = 3.2$ miliona JP. Star Oil chce maksymalizować wartość NPV uzyskaną na drodze wyboru odpowiedniego udziału w inwestycjach `1`,...,`5`, zakładając, ze środki pieniężne nie wykorzystane obecnie nie będą dostępne za rok. Można tego dokonać za pomocą programu `7_5_NPV.ecl`:

```
/*1*/ :- lib(eplex).
 /*2*/ top :-
   %  Xn - ułamek kupowanej inwestycji  n-tej
   % Zmienne i ich dziedziny:
/*3*/ Zmienne = [X1,X2,X3,X4,X5],
/*4*/  Zmienne $:: 0.0..1.0Inf,
   % Ograniczenia:
/*5*/  X1 $=< 1,
/*6*/  X2 $=< 1,
/*7*/  X3 $=< 1,
/*8*/  X4 $=< 1,
/*9*/  X5 $=< 1,
   % Ograniczenie dla chwili obecnej:
/*10*/ 11*X1 + 53*X2  + 5*X3 + 5*X4 + 29*X5 $=< 40,
   % Ograniczenie dla chwili za rok:
/*11*/ 3*X1 + 6*X2  + 5*X3 + X4 + 34*X5 $=< 20,  \verb+
   % Konfigurowanie solwera eplex dla maksymalizacji wskaźnika jakości:
```

```
/*12*/  eplex_solver_setup(max(13*X1 + 16*X2 + 16*X3 +14*X4 + 39*X5)),
    % Rozwiązywanie problemu:
 /*13*/  eplex_solve(Zysk),
    % Przedstawienie wyników:
 /*14*/ (
 /*15*/ foreach(Nazwa,["X1","X2","X3","X4","X5"]),
 /*16*/  foreach(V, Zmienne)
 /*17*/  do
 /*18*/  eplex_var_get(V, typed_solution, V),
 /*19*/      write(Nazwa),write(" = "),write(V),nl
 /*20*/ ),
 /*19*/ write("Zysk = "),write(Zysk).
```

Rozwiązaniem jest:

```
X1 = 1.0
X2 = 0.200859950859951
X3 = 1.0
X4 = 1.0
X5 = 0.288083538083538
Zysk = 57.4490171990172
```

7.3 Hurtownie - dostawcy

Klasycznym problemem programowania liniowego jest problem minimalizacji liniowego wskaźnika jakości przy liniowych ograniczeniach. Przykładem może być następujący problem transportowy dla 3 hurtowni i 4 dostawców:
U czterech dostawców D1, D2, D3 i D4 zakonktraktowano pewną ilość produktu, które należy dostarczyć do trzech hurtowni H1, H2 i H3 o ograniczonej pojemności i różnej odległości od dostawców, zob. rysunek 7.1.
Znane są:
Wielkość kontraktu podpisanego z dostawcą j:

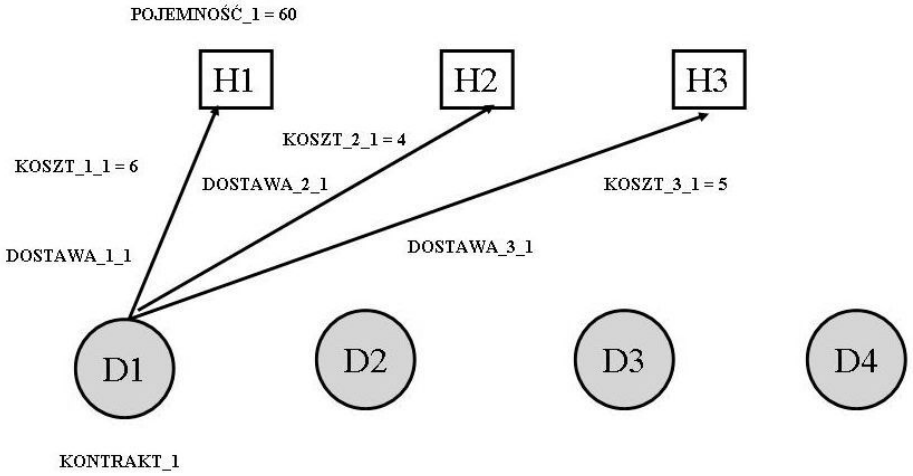
```
KONTRAKT_j,  j=1,2,3,4
```

Koszty dostarczenia jednostki wyrobu do hurtowni i od dostawcy j:

```
KOSZT_i_j,    i=1,2,3, j=1,2,3,4
```

Wolna pojemność każdej hurtowni i:

```
POJEMNOSC_i,    i=1,2,3
```



Rysunek 7.1: Hurtownie i dostawcy

Należy określić wielkość dostawy:

`DOSTAWA_i_j, i=1,2,3, j=1,2,3,4`

do każdej hurtowni i od każdego dostawcy j tak, by zminimalizować całkowity koszt transportu dostaw nie przekraczając pojemności hurtowni.
Rozwiązanie przedstawia program `7_6_hurtownie_dostawcy_1.ecl`:

```
/*1*/  :- lib(eplex).
 /*2*/  top :-
/*3*/     wyznacz(_,_).

/*4*/  wyznacz(Koszt,Zmienne):-
% Definicje zmiennych i ich zakresów:
/*5*/     Zmienne = [DOSTAWA_1_1,DOSTAWA_2_1,DOSTAWA_3_1,
                     DOSTAWA_1_2,DOSTAWA_2_2,DOSTAWA_3_2,
                     DOSTAWA_1_3,DOSTAWA_2_3,DOSTAWA_3_3,
                     DOSTAWA_1_4,DOSTAWA_2_4,DOSTAWA_3_4],
/*6*/     Zmienne $:: 0.0..1.0Inf,
/*7*/    % integers(Zmienne),

/*8*/     KOSZT_1_1 is 6.5,
/*9*/     KOSZT_1_2 is 2,
/*10*/    KOSZT_1_3 is 6.3,
/*11*/    KOSZT_1_4 is 7.3,
```

```
/*12*/    KOSZT_2_1 is 4,
/*13*/    KOSZT_2_2 is 9.7,
/*14*/    KOSZT_2_3 is 5.2,
/*15*/    KOSZT_2_4 is 3,
/*16*/    KOSZT_3_1 is 5.8,
/*17*/    KOSZT_3_2 is 2.4,
/*18*/    KOSZT_3_3 is 1.7,
/*19*/    KOSZT_3_4 is 9,

/*20*/    POJEMNOSC_1 is 60,
/*21*/    POJEMNOSC_2 is 55,
/*22*/    POJEMNOSC_3 is 51,

/*23*/    KONTRAKT_1 is 35.5,
/*24*/    KONTRAKT_2 is 37,
/*25*/    KONTRAKT_3 is 22.7,
/*26*/    KONTRAKT_4 is 32,

% Ograniczenie kontraktu każdego dostawcy :
/*27*/    DOSTAWA_1_1 + DOSTAWA_2_1 + DOSTAWA_3_1 $= KONTRAKT_1,
/*28*/    DOSTAWA_1_2 + DOSTAWA_2_2 + DOSTAWA_3_2 $=  KONTRAKT_2,
/*29*/    DOSTAWA_1_3 + DOSTAWA_2_3 + DOSTAWA_3_3 $=  KONTRAKT_3,
/*30*/    DOSTAWA_1_4 + DOSTAWA_2_4 + DOSTAWA_3_4 $= KONTRAKT_4,

% Ograniczenie pojemności każdej hurtowni :
/*31*/    DOSTAWA_1_1 + DOSTAWA_1_2 + DOSTAWA_1_3 + DOSTAWA_1_4 $=<  POJEMNOSC_1,
/*32*/    DOSTAWA_2_1 + DOSTAWA_2_2 + DOSTAWA_2_3 + DOSTAWA_2_4 $=<  POJEMNOSC_2,
/*33*/    DOSTAWA_3_1 + DOSTAWA_3_2 + DOSTAWA_3_3 + DOSTAWA_3_4 $=< POJEMNOSC_3,

% Konfiguracja solwera eplex'a  dla mnimalizowanego wskaźnika jakości:
/*34*/    eplex_solver_setup(min(
            KOSZT_1_1 * DOSTAWA_1_1 + KOSZT_2_1 * DOSTAWA_2_1 +
            KOSZT_3_1 * DOSTAWA_3_1 + KOSZT_1_2 * DOSTAWA_1_2 +
            KOSZT_2_2 * DOSTAWA_2_2 + KOSZT_3_2 * DOSTAWA_3_2 +
            KOSZT_1_3 * DOSTAWA_1_3 + KOSZT_2_3 * DOSTAWA_2_3 +
            KOSZT_3_3 * DOSTAWA_3_3 + KOSZT_1_4 * DOSTAWA_1_4 +
            KOSZT_2_4 * DOSTAWA_2_4 + KOSZT_3_4 * DOSTAWA_3_4
             )),

 % Rozwiązywanie problemu za pomocą solwera eplex'a:
/*35*/    eplex_solve(Koszt),

% Wydruk optymalnych wartości zmiennych
/*36*/    (foreach(Nazwa,[
            "DOSTAWA_1_1","DOSTAWA_2_1","DOSTAWA_3_1",
            "DOSTAWA_1_2","DOSTAWA_2_2","DOSTAWA_3_2",
            "DOSTAWA_1_3","DOSTAWA_2_3","DOSTAWA_3_3",
```

```
          "DOSTAWA_1_4","DOSTAWA_2_4","DOSTAWA_3_4"]),
/*37*/    foreach(V, Zmienne)
/*38*/    do
/*39*/    eplex_var_get(V, typed_solution, V),
/*40*/    write(Nazwa),write(" = "),write(V),nl
/*41*/    ),
/*42*/    write("Koszt"),write(" = "),write(Koszt),nl.
```

Program generuje komunikat:

```
DOSTAWA_1_1 = 0.0
DOSTAWA_2_1 = 23.0
DOSTAWA_3_1 = 12.5
DOSTAWA_1_2 = 37.0
DOSTAWA_2_2 = 0.0
DOSTAWA_3_2 = 0.0
DOSTAWA_1_3 = 0.0
DOSTAWA_2_3 = 0.0
DOSTAWA_3_3 = 22.7
DOSTAWA_1_4 = 0.0
DOSTAWA_2_4 = 32.0
DOSTAWA_3_4 = 0.0
Koszt = 373.09
```

Odkomentowanie linii `/*7*/` sprawi, że solwer będzie solwerem problemów programowania całkowitoliczbowego. Jeżeli dodatkowo zmienimy ograniczenia w liniach `/*27*/`„..., `/*30*/` z "`$=`" na "`$>=`" (tzn. jeżeli zaakceptujemy nieznaczne przekroczenie wielkości kontraktów), to otrzymamy program `7_7_hurtownie_dostawcy _2.ecl`:

```
/*1*/   :- lib(eplex).
 /*2*/   top :-
/*3*/      wyznacz(_,_).

/*4*/   wyznacz(Koszt,Zmienne):-
   % Definicje zmiennych i ich zakresów:
/*5*/      Zmienne = [DOSTAWA_1_1,DOSTAWA_2_1,DOSTAWA_3_1,
                      DOSTAWA_1_2,DOSTAWA_2_2,DOSTAWA_3_2,
                      DOSTAWA_1_3,DOSTAWA_2_3,DOSTAWA_3_3,
                      DOSTAWA_1_4,DOSTAWA_2_4,DOSTAWA_3_4],
/*6*/      Zmienne $:: 0.0..1.0Inf,
```

```
/*7*/      integers(Zmienne),

/*8*/   KOSZT_1_1 is 6.5,
/*9*/   KOSZT_1_2 is 2,
/*10*/  KOSZT_1_3 is 6.3,
/*11*/  KOSZT_1_4 is 7.3,
/*12*/  KOSZT_2_1 is 4,
/*13*/  KOSZT_2_2 is 9.7,
/*14*/  KOSZT_2_3 is 5.2,
/*15*/  KOSZT_2_4 is 3,
/*16*/  KOSZT_3_1 is 5.8,
/*17*/  KOSZT_3_2 is 2.4,
/*18*/  KOSZT_3_3 is 1.7,
/*19*/  KOSZT_3_4 is 9,

/*20*/  POJEMNOSC_1 is 60,
/*21*/  POJEMNOSC_2 is 55,
/*22*/  POJEMNOSC_3 is 51,
/*23*/  KONTRAKT_1 is 35.5,
/*24*/  KONTRAKT_2 is 37,
/*25*/  KONTRAKT_3 is 22.7,
/*26*/  KONTRAKT_4 is 32,

 % Ograniczenie kontraktu każdego dostawcy:
/*27*/  DOSTAWA_1_1 + DOSTAWA_2_1 + DOSTAWA_3_1 $>= KONTRAKT_1,
/*28*/  DOSTAWA_1_2 + DOSTAWA_2_2 + DOSTAWA_3_2 $>= KONTRAKT_2,
/*29*/  DOSTAWA_1_3 + DOSTAWA_2_3 + DOSTAWA_3_3 $>= KONTRAKT_3,
/*30*/  DOSTAWA_1_4 + DOSTAWA_2_4 + DOSTAWA_3_4 $>= KONTRAKT_4,

 % Ograniczenie pojemności każdej hurtowni :
/*31*/  DOSTAWA_1_1 + DOSTAWA_1_2 + DOSTAWA_1_3 + DOSTAWA_1_4 $=< POJEMNOSC_1,
/*32*/  DOSTAWA_2_1 + DOSTAWA_2_2 + DOSTAWA_2_3 + DOSTAWA_2_4 $=< POJEMNOSC_2,
/*33*/  DOSTAWA_3_1 + DOSTAWA_3_2 + DOSTAWA_3_3 + DOSTAWA_3_4 $=< POJEMNOSC_3,

 % Konfiguracja solwera eplex'a dla mnimalizowanego wskaźnika jakości:
/*34*/  eplex_solver_setup(min(
 KOSZT_1_1 * DOSTAWA_1_1 + KOSZT_2_1 * DOSTAWA_2_1 +
 KOSZT_3_1 * DOSTAWA_3_1 + KOSZT_1_2 * DOSTAWA_1_2 +
 KOSZT_2_2 * DOSTAWA_2_2 + KOSZT_3_2 * DOSTAWA_3_2 +
 KOSZT_1_3 * DOSTAWA_1_3 + KOSZT_2_3 * DOSTAWA_2_3 +
 KOSZT_3_3 * DOSTAWA_3_3 + KOSZT_1_4 * DOSTAWA_1_4 +
 KOSZT_2_4 * DOSTAWA_2_4 + KOSZT_3_4 * DOSTAWA_3_4
 )),

  % Rozwiązywanie problemu za pomocą solwera eplex'a:
/*35*/  eplex_solve(Koszt),

 % Wydruk optymalnych wartości zmiennych
```

```
/*36*/  (foreach(Nazwa,[
 'DOSTAWA_1_1','DOSTAWA_2_1','DOSTAWA_3_1',
 'DOSTAWA_1_2','DOSTAWA_2_2','DOSTAWA_3_2',
 'DOSTAWA_1_3','DOSTAWA_2_3','DOSTAWA_3_3',
 'DOSTAWA_1_4','DOSTAWA_2_4','DOSTAWA_3_4']),
/*37*/  foreach(V, Zmienne)
/*38*/  do
/*39*/  eplex_var_get(V, typed_solution, V),
/*40*/  write(Nazwa),write(' = '),write(V), nl
/*41*/  ),
/*42*/  write('Koszt'),write(' = '),write(Koszt), nl.
```

Rozwiązaniem jest:

```
DOSTAWA_1_1 = 0
DOSTAWA_2_1 = 23
DOSTAWA_3_1 = 13
DOSTAWA_1_2 = 37
DOSTAWA_2_2 = 0
DOSTAWA_3_2 = 0
DOSTAWA_1_3 = 0
DOSTAWA_2_3 = 0
DOSTAWA_3_3 = 23
DOSTAWA_1_4 = 0
DOSTAWA_2_4 = 32
DOSTAWA_3_4 = 0
Koszt = 376.5
```

Rozwiązanie to jest intuicyjnie oczywiste. Ponieważ dla otrzymania rozwiązania całkowitoliczbowego ograniczenia w liniach `/*27*/`, ..., `/*30*/` zostały "zrelaksowane", tzn. dopuściliśmy możliwości przekroczenia wielkości zakontraktowanych, minimalny koszt realizacji zamówień uległ powiększeniu.

7.4 Mieszamy oleje

Popatrzmy na rozwiązanie innego klasycznego przykład z dziedziny badań operacyjnych, rozwiązywanego metodami programowania liniowego:

Produkcja pewnego typu żywności wymaga rafinacji i mieszania niektórych z pośród pięciu olei: dwóch olei roślin pospolitych (P1 i P2) i trzech olei roślin tropikalnych (T1, T2 i T3). Mieszaninę należy tak dobrać, by miała potrzebną twardość i dawała maksymalny zysk.

Oleje roślin pospolitych wymagają innej linii produkcyjnej dla rafinacji aniżeli oleje roślin tropikalnych.

W ciągu miesiąca można rafinować nie więcej niż 200 ton olei roślin pospolitych i nie więcej niż 250 ton olei roślin tropikalnych. Straty olejów w produkcji są pomijalne.

Koszty zakupu i rafinacji tony oleju i współczynnik twardości olejów przedstawia tablica 7.2.

Oleje	P1	P2	T1	T2	T3
Koszt (JP)	110	120	130	110	115
Twardość	8.8	6.1	2.0	4.2	5.0

Tablica 7.2: Koszt i twardość olejów

Twardość produktu jest liniową kombinacja twardości składników i powinna być zawarta w przedziale `[3,...,6]`. Tonę produktu sprzedaje się za `150` JP. Określić liczbę ton olejów kupowanych w ciągu miesiąca i wielkość miesięcznej produkcji, maksymalizujące miesięczny zysk z produkcji. Definiuje się następujące zmienne:

`XP1, XP2, XT1, XT2, XT3` - tony zakupionych olejów w ciągu miesiąca,

`Y` – tony wyprodukowanego produktu w ciągu miesiąca.

Rozwiązanie przedstawia program `7_8_oleje_1.ecl`:

```
/*1*/  :- lib(eplex).
 /*2*/  top :-
/*3*/     oleje_1(_,_).

/*4*/     oleje_1(Zysk, Zmienne) :-
% Definicja zmiennych i ich zakresów:
/*5*/     Zmienne = [XP1,XP2,XT1,XT2,XT3,Y],
/*6*/    % integers(Zmienne),
/*7*/     Zmienne $:: 0.0..1.0Inf,

% Definicje ograniczeń:
/*8*/     XP1 + XP2 $=< 200,
/*9*/     XT1 + XT2 + XT3 $=< 250,
/*10*/    8.8*XP1 + 6.1*XP2 + 2*XT1 + 4.2*XT2 +5*XT3 $=< 6*Y,
```

```
/*11*/    8.8*XP1 + 6.1*XP2 + 2*XT1 + 4.2*XT2 +5*XT3 $>= 3*Y,
/*12*/    XP1 + XP2 + XT1 + XT2 + XT3 $= Y,

% Konfiguracja solwera eplex'a
% dla maksymalizowanego wskaźnika jakości:
/*13*/    eplex_solver_setup(max(
               150*Y - 110*XP1 - 120*XP2 -130*XT1 - 110*XT2 - 115*XT3)),

% Rozwiązywanie problemu za pomocą solwera eplex'a:
/*14*/    eplex_solve(Zysk),

% Wydruk optymalnych wartości zmiennych
/*15*/    (foreach(Nazwa,["XP1","XP2","XT1","XT2","XT3","Y"]),
/*16*/    foreach(V, Zmienne) do
/*17*/    eplex_var_get(V, typed_solution, V),
/*18*/    write(Nazwa),write(" = "),write(V),nl
/*19*/    ),
/*20*/    write("Zysk = "),write(Zysk).
```

Program generuje następujący komunikat:

```
XP1 = 159.259259259259
XP2 = 40.7407407407409
XT1 = 0.0
XT2 = 250.0
XT3 = 0.0
Y = 450.0
Zysk = 17592.5925925926
```

Jeżeli odkomentujemy linię `6`, otrzymujemy program `7_9_oleje_2.ecl`, który generuje rozwiązanie całkowitoliczbowe:

```
XP1 = 159
XP2 = 41
XT1 = 0
XT2 = 250
XT3 = 0
Y = 450
Zysk = 17590.0
```

7.5 Jak zarobić i się nie narobić?

Już dla poprzednich przykładów można było zauważyć, że ECL^iPS^e nie wymaga doprowadzenia problemu programowania liniowego do postaci kanonicznej. W poniższym przykładzie owa dowolność postaci problemu zostanie dalej posunięta. Przykład jest następujący:

Członek Izby Starszej Parlamentu Absurdolandii, *Bardzo Ważny Senator*, głęboko przekonany iż etanol w paliwach samochodowych z pewnością ocali Świat, przez długie lata czynił wszystko co mógł by sprostać legislacyjnym życzeniom i sugestiom znanego producenta etanolu - koncernu *Corny Fuels*. Doceniając jego starania, zaprzyjaźniony Dyrektor Generalny tego koncernu nakazał podległemu mu bankowi *Corny Fuels Bank* udzielić firmie Żony Bardzo Ważnego Senatora bardzo korzystnego kredytu: 100 milionów JP na 4 lata ze stałym oprocentowaniem 2% rocznie, z przeznaczeniem na jakąś mętną inwestycję. Przyjazny dyrektor *Corny Fuels Bank* zaproponował ponadto Żonie Senatora, by sama ustaliła raty splat kredytu, sugerując jedynie, by roczna rata spłaty nie była mniejsza niż 10 milionów JP. Żona Senatora bardzo słusznie zrezygnowała z mętnej inwestycji i cały kredyt ulokowała w innym, mniej przyjaznym banku, dającym za roczne lokaty oprocentowanie wyższe od inflacji o 2%. Inflacja jest prognozowana za pierwszy rok na 5% rocznie, za drugi rok na 4% rocznie, za trzeci rok na 3% a za czwarty rok na 2% rocznie.

Żona Senatora ma jednak poważny kłopot związany ze szczegółami umowy. Doradca finansowy Żony Senatora zaproponował dwie różne strategie inwestowania:
1. Maksymalizować zysk z lokaty przy spłacaniu kredytu zgodnie z życzeniami *Corny Fuels Bank*.
2. Minimalizować prognozowane rzeczywiste koszty spłaty kredytu przy spłacaniu kredytu zgodnie z życzeniami *Corny Fuels Bank*.

Doradca miał jednak trudności z wyjaśnieniem, jakie powinny być raty spłaty kredytu i jaki będzie zysk dla obydwu strategii. Może mu w tym pomóc program `7_10_senator.ecl`:

```
/*1*/  :- lib(eplex).

/*2*/  top:-
/*3*/     write("Napisz numer wersji (1 lub 2):"),nl,
/*4*/     read_token(Numer, integer),
/*5*/     wyznacz(Numer).
```

```
/*6*/  wyznacz(1):-
/*7*/     A $>= 10,    % rata po 1 roku
/*8*/     B $>= 10,    % rata po 2 roku
/*9*/     C $>= 10,    % rata po 3 roku
/*10*/    D $>= 10,    % rata po 4 roku

/*11*/    raty([A,B,C,D], 100),
/*12*/    Zysk_A $= 100*(1.07)-A,
/*13*/    Zysk_B $= Zysk_A*(1.06)-B,
/*14*/    Zysk_C $= Zysk_B*(1.05)-C,
/*15*/    Zysk_D $= Zysk_C*(1.04)-D,
/*16*/    Zysk_D $:: 0..250,

/*17*/    eplex_solver_setup(max(Zysk_D)),
/*18*/    eplex_solve(Zysk_D),
/*19*/    eplex_get(vars, Vars),
/*20*/    eplex_get(typed_solution, Vals),
/*21*/    Vars = Vals,

/*22*/    Koszt_splaty is
              A*0.95+B*0.95*0.96+C*0.95*0.96*0.97+
                        D*0.95*0.96*0.97*0.98,

/*23*/    write("Rata po 1 roku = "),write(A),write(" mln"),nl,
/*24*/    write("Rata po 2 roku = "),write(B),write(" mln"),nl,
/*25*/    write("Rata po 3 roku = "),write(C),write(" mln"),nl,
/*26*/    write("Rata po 4 roku = "),write(D),write(" mln"),nl,
/*27*/    write("Prognozowany rzeczywisty koszt splaty = "),
                  write(Koszt_splaty),write(" mln "),nl,
/*28*/    write("Maksymalny zysk po 4 latach = "),
                  write(Zysk_D),write(" mln "),nl,nl.

/*29*/  wyznacz(2):-
/*39*/    A $>= 10,    % rata po 1 roku
/*31*/    B $>= 10,    % rata po 2 roku
/*32*/    C $>= 10,    % rata po 3 roku
/*33*/    D $>= 10,    % rata po 4 roku
/*34*/    raty([A,B,C,D], 100),
/*35*/    Koszt_splaty $=
              A*0.95+B*0.95*0.96+C*0.95*0.96*0.97+
                        D*0.95*0.96*0.97*0.98,

/*36*/    eplex_solver_setup(min(Koszt_splaty)),
/*37*/    eplex_solve(Koszt_splaty),
/*38*/    eplex_get(vars, Vars),
/*39*/    eplex_get(typed_solution, Vals),
/*40*/    Vars = Vals,
/*41*/    Zysk_A is 100*(1.07)-A,
/*42*/    Zysk_B is Zysk_A*(1.06)-B,
```

```
/*43*/    Zysk_C is Zysk_B*1.05-C,
/*44*/    Zysk_D is Zysk_C*1.04-D,

/*45*/    write("Rata po 1 roku = "),write(A),write(" mln"),nl,
/*46*/    write("Rata po 2 roku = "),write(B),write(" mln"),nl,
/*47*/    write("Rata po 3 roku = "),write(C),write(" mln"),nl,
/*48*/    write("Rata po 4 roku = "),write(D),write(" mln"),nl,
/*49*/    write("Minimalny prognozowany rzeczywisty koszt splaty = "),
                  write(Koszt_splaty),write(" mln"),nl,
/*50*/    write("Zysk po 4 latach ="),write(Zysk_D),write(" mln"),nl.

/*51*/ raty([],Kredyt) :-
/*52*/    Kredyt $=0.
/*53*/ raty([Rata|Lista_rat],Kredyt) :-
/*54*/    Kwota_do_zaplacenia $=(1+2/100)*Kredyt-Rata,
/*55*/    raty(Lista_rat,Kwota_do_zaplacenia).
```

Program generuje komunikat:

```
Napisz numer wersji (1 lub 2): piszemy 1
 Rata po 1 roku = 10.0 mln
 Rata po 2 roku = 10.0 mln
 Rata po 3 roku = 10.0 mln
 Rata po 4 roku = 77.027136 mln
 Prognozowany rzeczywisty koszt splaty = 94.2448598792192 mln
 Maksymalny zysk po 4 latach: 13.932304 mln

Napisz numer wersji (1 lub 2): piszemy 2
 Rata po 1 roku = 10.0 mln
 Rata po 2 roku = 10.0 mln
 Rata po 3 roku = 10.0 mln
 Rata po 4 roku = 77.027136 mln
 Minimalny prognozowany rzeczywisty koszt splaty = 94.2448598792192 mln
 Zysk po 4 latach: 13.932304 mln
```

Okazuje się więc, że niezależnie od wyboru strategii, zyski będą takie same i koszty spłaty kredytu będą również takie same.

7.6 Inwestujemy

Odstępstwa od postaci kanonicznej programowania liniowego mogą być w ECL^iPS^e bardzo daleko posunięte. Ilustruje to następujący przykład:

Małe przedsiębiorstwo dysponuje pewna gotówką. Księgowy określił zapotrzebowanie na gotówkę na najbliższe 5 lat. Pozostałą gotówkę chce bezpiecznie zainwestować. Rozważa w tym celu trzy różne inwestycje:

- krótkoterminowe papiery wartościowe (1-roczne oprocentowane na 20 % rocznie);
- średnioterminowe papiery wartościowe (2-letnie obligacje oprocentowane na 47 % rocznie);
- długoterminowe papiery wartościowe (3-letnie obligacje oprocentowane na 78 % rocznie).

Księgowy planuje inwestycje na 5 lat biorąc pod uwagę zapotrzebowanie na gotówkę i następujące opcje inwestycyjne:

Opcja inwestycyjna 1: zaspokoić bieżące potrzeby i uzyskać maksimum zysku po 5 latach przy początkowej gotówce 100000 jednostek pieniężnych.

Opcja inwestycyjna 2: zaspokoić bieżące potrzeby i minimalizować początkową gotówkę.

Opcja inwestycyjna 3: zaspokoić bieżące potrzeby i mieć po 5 latach tą samą gotówkę, z którą rozpoczynało się inwestowanie na początku 1 roku.

Zapotrzebowanie na gotówkę z początkiem każdego roku jest następujące:

Rok	*Wielkość*	*Zmienna*
1	10000	Zap1
2	10000	Zap2
3	20000	Zap3
4	20000	Zap4
5	20000	Zap5

Zdefiniujmy następujące zmienne:

Kinx : Krótkoterminowe inwestycje na początku roku x
Sinx : Średnioterminowe inwestycje na początku roku x
Dinx : Długoterminowe inwestycje na początku roku x
Cinx : Całkowite inwestycje na początku roku x

Kzx : Zyski z krótkoterminowych inwestycji na początku roku x
Szx : Zyski z średnioterminowych inwestycji na początku roku x
Dzx : Zyski z długoterminowych inwestycji na początku roku x
Czx : Całkowite zyski na początku roku x

Zapx : zapotrzebowanie na gotówkę na początku roku x

Gnadx : gotówka nie wykorzystana na początku roku x

Jakie decyzje składają się na każdą z wymienionych opcji inwestycyjnych? Wyjaśnia to program `7_11_inwestycje.ecl`:

```
/*1*/  :- lib(eplex).
 /*2*/  top:-
/*3*/     writeln("Opcja inwestycyjna 1:"),
/*4*/     not( not(top(100000.0,_))),
/*5*/     writeln("Opcja inwestycyjna 2:"),
/*6*/     not( not(top(_,150000.0))),
/*7*/     writeln("Opcja inwestycyjna 3:"),
/*8*/     not( not(top(X,X))).

/*9*/  top(Gotowka_poczatkowa,Gotowka_koncowa):-
       % wskaźnik jakości jest narazie nieokreślony)
/*10*/    eplex_solver_setup(max(0)),
/*11*/    inwestycje(Gotowka_poczatkowa, Zmienne, Nazwy, Gotowka_koncowa),
      % wyznacz minimum Gotowka_poczatkowa and i zachowaj je:
/*12*/    eplex_probe(min(Gotowka_poczatkowa), Gotowka_poczatkowa),
      % wyznacz maksimum Gotowka_koncowa i zachowaj je::
/*13*/    eplex_probe(max(Gotowka_koncowa), Gotowka_koncowa),
/*14*/    eplex_get(typed_solution, Vs), eplex_get(vars, Vs),

/*15*/    writelist(Nazwy, Zmienne),
/*16*/    write("Gotowka poczatkowa:  "),writeln(Gotowka_poczatkowa),
/*17*/    write("Gotowka koncowa:     "),writeln(Gotowka_koncowa),nl.

/*18*/    inwestycje(Gotowka_poczatkowa, Zmienne, Nazwy, GnadC) :-
/*19*/    Zmienne =   [
                Kin1,Kin2,Kin3,Kin4,Kin5,
                Sin1,Sin2,Sin3,Sin4,Sin5,
                Din1,Din2,Din3,Din4,Din5,
                Cin1,Cin2,Cin3,Cin4,Cin5,
                Kz2,Kz3,Kz4,Kz5,Kz6,
                Sz3,Sz4,Sz5,Sz6,
                Dz4,Dz5,Dz6,
                Cz2,Cz3,Cz4,Cz5,Cz6,
                Gnad1,Gnad2,Gnad3,Gnad4,Gnad5],
```

```
/*20*/    Nazwy =   [
                "Kin1","Kin2","Kin3","Kin4","Kin5",
                "Sin1","Sin2","Sin3","Sin4","Sin5",
                "Din1","Din2","Din3","Din4","Din5",
                "Cin1","Cin2","Cin3","Cin4","Cin5",
                "Kz2","Kz3","Kz4","Kz5","Kz6",
                "Sz3","Sz4","Sz5","Sz6",
                "Dz4","Dz5","Dz6",
                "Cz2","Cz3","Cz4","Cz5","Cz6",
                "Gnad1","Gnad2","Gnad3","Gnad4","Gnad5"],

/*21*/    Zmienne $:: 0..inf,

/*22*/    Cin1 $= Sin1 + Kin1 + Din1,
/*23*/    Cin2 $= Sin2 + Kin2 + Din2,
/*24*/    Cin3 $= Sin3 + Kin3 + Din3,
/*25*/    Cin4 $= Sin4 + Kin4 + Din4,
/*26*/    Cin5 $= Sin5 + Kin5 + Din5,

/*27*/    Sz3 $= 1.47 * Sin1,
/*28*/    Sz4 $= 1.47 * Sin2,
/*29*/    Sz5 $= 1.47 * Sin3,
/*30*/    Sz6 $= 1.47 * Sin4,

/*31*/    Dz4 $= 1.78 * Din1,
/*32*/    Dz5 $= 1.78 * Din2,
/*33*/    Dz6 $= 1.78 * Din3,

/*34*/    Kz2 $= 1.2 * Kin1,
/*35*/    Kz3 $= 1.2 * Kin2,
/*36*/    Kz4 $= 1.2 * Kin3,
/*37*/    Kz5 $= 1.2 * Kin4,
/*38*/    Kz6 $= 1.2 * Kin5,

/*39*/    Cz2 $= Kz2,
/*40*/    Cz3 $= Kz3 + Sz3,
/*41*/    Cz4 $= Kz4 + Sz4 + Dz4,
/*42*/    Cz5 $= Kz5 + Sz5 + Dz5,
/*43*/    Cz6 $= Kz6 + Sz6 + Dz6,

/*44*/    Zap1 $>= 10000,
/*45*/    Zap2 $>= 10000,
/*46*/    Zap3 $>= 20000,
/*47*/    Zap4 $>= 20000,
/*48*/    Zap5 $>= 20000,

/*49*/    Gnad1 $= Gotowka_poczatkowa - Cin1 - Zap1,
/*50*/    Gnad2 $= Gnad1 + Cz2 - Cin2 - Zap2,
```

```
/*51*/    Gnad3 $= Gnad2 + Cz3 - Cin3 - Zap3,
/*52*/    Gnad4 $= Gnad3 + Cz4 - Cin4 - Zap4,
/*53*/    Gnad5 $= Gnad4 + Cz5 - Cin5 - Zap5,
/*54*/    Gotowka_koncowa $= Gnad5 + CzC.

/*55*/    writelist([], []).
/*56*/    writelist([FN|RN], [FV|RV]) :-
/*57*/    write(FN), write(" = "), writeln(FV),
/*58*/    writelist(RN, RV).
```

Zdumienie może budzić "podwójna negacja" w liniach `4`, `6` i `8`. Jest to stary trik prologowy, służący do tego, by kolejno wywołać ten sam predykat z różnymi wartościami argumentów: spełnienie predykatu `top(_,_)` sprawia że wewnętrzna negacja kończy się porażką powodującą kasowanie wszystkich wyników związanych ze wzmiankowanym predykatem `top(_,_)`, co z kolei sprawi, że zewnętrzna negacja jest spełniona, dzięki czemu można przejść do kolejnego predykatu `top(_,_)`.

Zasadniczy model problemu to zależności bilansowe z linii `/*22*/ - \verb`/*54*/, dokładnie odpowiadające opisowi problemu i bardzo odległe od postaci kanonicznej programowania liniowego.

Dla większej czytelności rozwiązania przedstawiono je w postaci tabelarycznej ułatwiającej porównanie wartości zmiennych dla różnych opcji. Jest ono różne od wydruku generowanego przez program i ma postać następującą:

Zmienna	*Opcja 1*	*Opcja 2*	*Opcja 3*
Kin1	8333.33333333333	8333.33333333333	8333.33333333333
Kin2	0.0	0.0	0.0
Kin3	0.0	0.0	0.0
Kin4	0.0	0.0	0.0
Kin5	0.0	0.0	0.0
Sin1	22860.8450182794	80187.1463253183	55293.192790018
Sin2	0.0	0.0	0.0
Sin3	13605.4421768707	13605.4421768696	13605.4421768707
Sin4	84674.3625341293	0.0	0.0
Sin5	0.0	0.0	0.0

Zmienna	*Opcja 1*	*Opcja 2*	*Opcja 3*
Din1	58805.8216483872	11235.9550561798	11235.9550561798
Din2	0.0	0.0	0.0
Din3	0.0	84269.6629213483	47675.5512244557
Din4	0.0	0.0	0.0
Din5	0.0	0.0	0.0
Cin1	90000.0	99756.4347148322	74862.4811795311
Cin2	0.0	0.0	0.0
Cin3	13605.4421768707	97875.105098218	61280.9934013264
Cin4	84674.3625341293	0.0	0.0
Cin5	0.0	0.0	0.0
Kz2	10000.0	10000.0	10000.0
Kz3	0.0	0.0	0.0
Kz4	0.0	0.0	0.0
Kz5	0.0	0.0	0.0
KzC	0.0	0.0	0.0
Sz3	33605.4421768707	117875.105098218	81280.9934013264
Sz4	0.0	0.0	0.0
Sz5	20000.0	19999.9999999984	20000.0
SzC	124471.31292517	0.0	0.0
Dz4	104674.362534129	20000.0	20000.0
Dz5	0.0	0.0	0.0
DzC	0.0	150000.0	84862.4811795311
Cz2	10000.0	10000.0	10000.0
Cz3	33605.4421768707	117875.105098218	81280.9934013264
Cz4	104674.362534129	20000.0	20000.0
Cz5	20000.0	19999.9999999984	20000.0
CzC	124471.31292517	150000.0	84862.4811795311
Gnad1	0.0	0.0	0.0
Gnad2	0.0	0.0	0.0
Gnad3	0.0	0.0	0.0
Gnad4	0.0	0.0	0.0
Gnad5	0.0	0.0	0.0
Got. początk.	100000.0	109756.434714832	84862.4811795311
Got. końcowa	124471.31292517	150000.0	84862.4811795311

7.7 Jeszcze jedno finansowe *Perpetuum Mobile*!

Modele matematyczne stosowane dla programowania liniowego są najczęściej różnego rodzaju bilansami, których natura jest różnorodna. Ilustruje to poniższy przykład, zainspirowany zadaniem przedstawionym w książce [Taha-08]:

Rozumosław Myślik, komputerowo-matematyczno-biznesowe cudowne dziecko, przemyśliwał czas jakiś nad tym, jak zarabiać on-line. Po długich rozważaniach zdecydował się na spekulacje walutowe bazujące na powszechnie dostępnych, ciągle aktualizowanych, internetowych kursach wymian, oraz na internetowych serwisach dla walutowych *spot trading*. Nieustannie zmieniające się kursy czyniły ponętną myśl, by coś kupić, coś innego sprzedać, znów coś tam kupić i coś jeszcze innego sprzedać, i zrobić to bardzo szybko w nadziei, że saldo tych transakcji będzie godziwym zyskiem spekulanta. Problem tylko w tym, co kupić, co sprzedać, no i wreszcie skąd wziąć kapitał początkowy. Jako osoba systematyczna i dobrze zorganizowana, zaczął od symulacji komputerowych dla wyników spekulacji pięcioma walutami: USD, EUR, GBP, JPY i PLN. Zebrane przezeń kursy wymiany są kursami średnimi globalnych rynków dla "kupuj" i "sprzedaj" dla określonej godziny, określonego dnia i miesiąca w określonym roku, zob. tablica 7.3.

	USD	EUR	GBP	JPY	PLN
USD	1	0.8353	0.692	91.89	3.4872
EUR	1.1972	1	0.8283	110.03	4.181
GBP	1.445	1.2073	1	132	5.0414
JPY	0.01088	0.009088	0.007575	1	3.7950
PLN	0.2867	0.2392	0.1983	0.2635	1

Tablica 7.3: Średnie kursy wymiany walut na dzień 10 marca 2010

Sens tablicy jest oczywisty, np. 1 EUR można sprzedać (kupić) za 1.1972 USD. Myślik uważa, że początkowy kapitał w wysokości `A` milionów (mln) USD) można powiększyć na rynku *spot trading* na drodze umiejętnej spekulacji (sprzedaży i kupna) odpowiednich ilości różnych walut. Problemem jest ile i czego kupować, ile i czego sprzedawać, by zmaksymalizować zysk z początkowego kapitału `A` mil USD.

Spekulacje są ograniczone dopuszczalną wielkością pojedynczej transakcji w następujący sposób: `USD <= 5`, `GBP <= 3.5`, `JPY <= 100`, `PLN <= 40` (w mln).

Dla potwierdzenia swych domniemań Myślik opracował program dla symulacji swych spekulacji o nazwie `7_12_spekulacje_walutowe.ecl`. W programie tym transakcje są zapisane za pomocą zmiennych typu:

`Waluta_i_na_Walute_j`

oznaczających ilość mln `Waluty_i` użytej na zakup waluty `Walutę_j`. Kursy wymiany z tablicy 7.3 są przedstawiane za pomocą zmiennych:

`Kurs_wymiany_Waluta_i_na_Waluta_j`.

Zmienna `A` oznacza początkową ilość USD (mln), zmienna `Z` - końcową ilość USD (mln).

Program ten korzysta z bilansów dla wszystkich walut będących przedmiotem spekulacji. Bilans te są (dla waluty i-tej) postaci:

```
Początkowa_ilość_waluty_i-tej +
     Waluty_i-ta_uzyskana_z_wymian_innych_walut =
       Końcowa_ilość_waluty_i-tej +
             Waluta_i-ta_wymieniona_na_inne_waluty.
```

W rozpatrywanym przykładzie przyjmuje się, że tylko dla USD jest niezerowa akumulacja początkowa i końcowa. Program `7_12_spekulacje_walutowe.ecl` jest postaci:

```
/*1*/  :- lib(eplex).

/*2*/  top:-
/*3*/      A = 5.0,
/*3a*/   %  A = 0.0,

   % konwersja  USD na inną walutę:
/*4*/     USD = [USD_na_EUR,USD_na_GBP,USD_na_JPY,USD_na_PLN],
/*5*/     USD :: 0.0..5.0,
/*6*/     Z :: 0.0..6.0,
/*6a*/ %    Z :: 0.0..120.0,
 %  Znaczenie zmiennych: USD_na_EUR - ilość mln USD użyta na zakup EUR
 %                       USD_na_GBP - ilość mln USD użyta na zakup GBP
 %                      itd., zgodnie z przyjętą definicją.

   %  konwersja  EUR na inną walutę:
/*7*/     EUR = [EUR_na_USD,EUR_na_GBP,EUR_na_JPY,EUR_na_PLN],
/*8*/     EUR :: 0.0..3.0,

   %  konwersja GBP na inną walutę:
/*9*/     GBP = [GBP_na_USD,GBP_na_EUR,GBP_na_JPY,GBP_na_PLN],
/*10*/    GBP :: 0.0..3.5,
```

```
   %  konwersja JPY na inną walutę:
/*11*/    JPY = [JPY_na_USD,JPY_na_EUR,JPY_na_GBP,JPY_na_PLN],
/*12*/    JPY :: 0.0..100.0,

   %  konwersja PLN  na inną walutę:
/*13*/    PLN = [PLN_na_USD,PLN_na_EUR,PLN_na_GBP,PLN_na_JPY],
/*14*/    PLN :: 0.0..40.0,

/*15*/    Kurs_wymiany_USD_na_EUR = 0.8353,
/*16*/    Kurs_wymiany_USD_na_GBP = 0.692,
/*17*/    Kurs_wymiany_USD_na_JPY = 91.89,
/*18*/    Kurs_wymiany_USD_na_PLN = 3.4872,
/*19*/    Kurs_wymiany_EUR_na_GBP = 0.8283 ,
/*20*/    Kurs_wymiany_EUR_na_JPY = 110.03,
/*21*/    Kurs_wymiany_EUR_na_PLN = 4.181,
/*22*/    Kurs_wymiany_GBP_na_JPY = 132,
/*23*/    Kurs_wymiany_GBP_na_PLN = 5.0414,
/*24*/    Kurs_wymiany_JPY_na_PLN = 3.7950,

/*25*/    Kurs_wymiany_EUR_na_USD is 1/(Kurs_wymiany_USD_na_EUR),
/*26*/    Kurs_wymiany_GBP_na_USD is 1/(Kurs_wymiany_USD_na_GBP),
/*27*/    Kurs_wymiany_JPY_na_USD is 1/(Kurs_wymiany_USD_na_JPY),
/*28*/    Kurs_wymiany_PLN_na_USD is 1/(Kurs_wymiany_USD_na_PLN),
/*29*/    Kurs_wymiany_GBP_na_EUR is 1/(Kurs_wymiany_EUR_na_GBP),
/*30*/    Kurs_wymiany_JPY_na_EUR is 1/(Kurs_wymiany_EUR_na_JPY),
/*31*/    Kurs_wymiany_PLN_na_EUR is 1/(Kurs_wymiany_EUR_na_PLN),
/*33*/    Kurs_wymiany_JPY_na_GBP is 1/(Kurs_wymiany_GBP_na_JPY),
/*33*/    Kurs_wymiany_PLN_na_GBP is 1/(Kurs_wymiany_GBP_na_PLN),
/*34*/    Kurs_wymiany_PLN_na_JPY is 1/(Kurs_wymiany_JPY_na_PLN),

    % bilans USD:
/*35*/    A + (Kurs_wymiany_EUR_na_USD)*EUR_na_USD +
                (Kurs_wymiany_GBP_na_USD)*GBP_na_USD +
                     (Kurs_wymiany_JPY_na_USD)*JPY_na_USD +
                          (Kurs_wymiany_PLN_na_USD)*PLN_na_USD $=
          Z + USD_na_EUR + USD_na_GBP + USD_na_JPY + USD_na_PLN.

    % bilans EUR:
/*36*/     (Kurs_wymiany_USD_na_EUR)*USD_na_EUR +
                (Kurs_wymiany_GBP_na_EUR)*GBP_na_EUR +
                     (Kurs_wymiany_JPY_na_EUR)*JPY_na_EUR +
                          (Kurs_wymiany_PLN_na_EUR)*PLN_na_EUR $=
           EUR_na_USD + EUR_na_GBP + EUR_na_JPY + EUR_na_PLN.

    % bilans GBP:
/*37*/     (Kurs_wymiany_USD_na_GBP)*USD_na_GBP +
                (Kurs_wymiany_EUR_na_GBP)*EUR_na_GBP +
                     (Kurs_wymiany_JPY_na_GBP)*JPY_na_GBP +
```

```
                              (Kurs_wymiany_PLN_na_GBP)*PLN_na_GBP $= ,
          GBP_na_USD + GBP_na_EUR + GBP_na_JPY + GBP_na_PLN.

     % bilans JPY:
/*38*/     (Kurs_wymiany_USD_na_JPY)*USD_na_JPY +
                 (Kurs_wymiany_EUR_na_JPY)*EUR_na_JPY +
                        (Kurs_wymiany_GBP_na_JPY)*GBP_na_JPY +
                               (Kurs_wymiany_PLN_na_JPY)*PLN_na_JPY $=
          JPY_na_USD + JPY_na_EUR + JPY_na_GBP + JPY_na_PLN.

     %bilans PLN :
/*39*/     (Kurs_wymiany_USD_na_PLN)*USD_na_PLN +
                  (Kurs_wymiany_EUR_na_PLN)*EUR_na_PLN +
                        (Kurs_wymiany_GBP_na_PLN)*GBP_na_PLN +
                                (Kurs_wymiany_JPY_na_PLN)*JPY_na_PLN $=
          PLN_na_USD + PLN_na_EUR + PLN_na_GBP + PLN_na_JPY.

/*40*/    eplex_solver_setup(max(Z)),
 /*41*/    eplex_solve(Z),
/*42*/    eplex_get(vars, Vars),
/*43*/    eplex_get(typed_solution, Vals),
/*44*/    Vars = Vals,

/*45*/    writeln("Poczatkowa ilosc USD (mln) ": A),
/*46*/    writeln("Koncowa ilosc USD (mln) ": Z),
/*47*/    write_positive("USD_na_EUR", USD_na_GBP),
/*48*/    write_positive("USD_na_GBP", USD_na_GBP),
/*49*/    write_positive("USD_na_JPY", USD_na_JPY),
/*50*/    write_positive("USD_na_PLN", USD_na_PLN),
/*51*/    write_positive("EUR_na_USD", EUR_na_USD),
/*52*/    write_positive("EUR_na_GBP", EUR_na_GBP),
/*53*/    write_positive("EUR_na_JPY", EUR_na_JPY),
/*54*/    write_positive("EUR_na_PLN", EUR_na_PLN),
/*55*/    write_positive("GBP_na_USD", GBP_na_USD),
/*56*/    write_positive("GBP_na_EUR", GBP_na_EUR),
/*57*/    write_positive("GBP_na_JPY", GBP_na_JPY),
/*58*/    write_positive("GBP_na_PLN", GBP_na_PLN),
/*59*/    write_positive("JPY_na_USD", JPY_na_USD),
/*60*/    write_positive("JPY_na_EUR", JPY_na_EUR),
/*61*/    write_positive("JPY_na_GBP", JPY_na_GBP),
/*62*/    write_positive("JPY_na_PLN", JPY_na_PLN),
/*63*/    write_positive("PLN_na_USD", PLN_na_USD),
/*64*/    write_positive("PLN_na_EUR", PLN_na_EUR),
/*65*/    write_positive("PLN_na_GBP", PLN_na_GBP),
/*66*/    write_positive("PLN_na_JPY", PLN_na_JPY),

/*67*/    writeln("Kurs_wymiany_USD_na_EUR":Kurs_wymiany_USD_na_EUR),
/*68*/    writeln("Kurs_wymiany_USD_na_GBP":Kurs_wymiany_USD_na_GBP),
/*69*/    writeln("Kurs_wymiany_USD_na_JPY":Kurs_wymiany_USD_na_JPY),
```

```
/*70*/    writeln("Kurs_wymiany_USD_na_PLN":Kurs_wymiany_USD_na_PLN),
/*71*/    writeln("Kurs_wymiany_EUR_na_USD":Kurs_wymiany_EUR_na_USD),
/*72*/    writeln("Kurs_wymiany_EUR_na_GBP":Kurs_wymiany_EUR_na_GBP),
/*73*/    writeln("Kurs_wymiany_EUR_na_JPY":Kurs_wymiany_EUR_na_JPY),
/*74*/    writeln("Kurs_wymiany_EUR_na_PLN":Kurs_wymiany_EUR_na_PLN),
/*75*/    writeln("Kurs_wymiany_GBP_na_USD":Kurs_wymiany_GBP_na_USD),
/*76*/    writeln("Kurs_wymiany_GBP_na_EUR":Kurs_wymiany_GBP_na_EUR),
/*77*/    writeln("Kurs_wymiany_GBP_na_JPY":Kurs_wymiany_GBP_na_JPY),
/*78*/    writeln("Kurs_wymiany_GBP_na_PLN":Kurs_wymiany_GBP_na_PLN),
/*79*/    writeln("Kurs_wymiany_JPY_na_USD":Kurs_wymiany_JPY_na_USD),
/*80*/    writeln("Kurs_wymiany_JPY_na_EUR":Kurs_wymiany_JPY_na_EUR),
/*81*/    writeln("Kurs_wymiany_JPY_na_GBP":Kurs_wymiany_JPY_na_GBP),
/*82*/    writeln("Kurs_wymiany_JPY_na_PLN":Kurs_wymiany_JPY_na_PLN),
/*83*/    writeln("Kurs_wymiany_PLN_na_USD":Kurs_wymiany_PLN_na_USD),
/*84*/    writeln("Kurs_wymiany_PLN_na_EUR":Kurs_wymiany_PLN_na_EUR),
/*85*/    writeln("Kurs_wymiany_PLN_na_GBP":Kurs_wymiany_PLN_na_GBP),
/*86*/    writeln("Kurs_wymiany_PLN_na_JPY":Kurs_wymiany_PLN_na_JPY).

/*87*/    write_positive(A, B):-
/*88*/    (B > 0 -> writeln(A:B); true).
```

Program generuje komunikat:

```
Poczatkowa ilosc USD (mln)  : 5.0
Koncowa ilosc USD (mln)  : 6.0
EUR_na_JPY : 0.00843576360267486
JPY_na_PLN : 0.928187069202315
PLN_na_USD : 3.4872
PLN_na_EUR : 0.0352699276227836

Kurs_wymiany_USD_na_EUR : 0.8353
Kurs_wymiany_USD_na_GBP : 0.692
Kurs_wymiany_USD_na_JPY : 91.89
Kurs_wymiany_USD_na_PLN : 3.4872
Kurs_wymiany_EUR_na_USD : 1.19717466778403
Kurs_wymiany_EUR_na_GBP : 0.8283
Kurs_wymiany_EUR_na_JPY : 110.03
Kurs_wymiany_EUR_na_PLN : 4.181
Kurs_wymiany_GBP_na_USD : 1.44508670520231
Kurs_wymiany_GBP_na_EUR : 1.20729204394543
Kurs_wymiany_GBP_na_JPY : 132
Kurs_wymiany_GBP_na_PLN : 5.0414
Kurs_wymiany_JPY_na_USD : 0.0108825769942322
Kurs_wymiany_JPY_na_EUR : 0.00908843042806507
```

```
Kurs_wymiany_JPY_na_GBP : 0.00757575757575758
Kurs_wymiany_JPY_na_PLN : 3.795
Kurs_wymiany_PLN_na_USD : 0.286763019041064
Kurs_wymiany_PLN_na_EUR : 0.239177230327673
Kurs_wymiany_PLN_na_GBP : 0.198357599079621
Kurs_wymiany_PLN_na_JPY : 0.263504611330698
```

W komunikacie są wymieniane tylko waluty podlegające wymianie, a mianowicie: `EUR_na_JPY`, `JPY_na_PLN`, `PLN_na_USD` i `PLN_na_EUR`. Waluta `GBP` nie uczestniczy w wymianie. Z symulacji wynika, że startując z początkową ilością 5 mln USD, uzyskuje się jednym aktem wielokrotnego kupna i sprzedaży końcową ilość 6 mln USD. Wynik był tak zaskakujący, ze Myślik postanowił dokładnie sprawdzić wszystkie bilanse:

Bilans USD, patrz linia `/*35*/`:

```
5+0+0+0+0.286763019041064*3.4872 = 6+0+0+0+0
```

co daje:

```
6 = 6.
```

Bilans EUR, patrz linia `/*36*/`:

```
0+0+0+0.239177230327673*0.0352699276227836 =
            0+0+0.00843576360267486+0
```

co daje:

```
0.008435763602674869 = 0.008435763602674869.
```

Bilans GBP jest trywialny:

```
0+0+0+0 = 0+0+0+0.
```

Bilans JPY, patrz linia `/*38*/`:

```
0+110.03*0.00843576360267486+0+0 = 0+0+0+0.928187069202315
```

co daje:

```
0.9281870692023148458 = 0.928187069202315.
```

Bilans PLN, patrz linia `/*39*/`:

```
0+0+0+3.795*0.928187069202315 = 3.4872+0.0352699276227836+0+0
```

co daje:

```
3.5224699 = 3.5224699.
```

A więc wszystko jest O.K.! Jeżeli rozwiązanie zostanie przekazane dealerowi walutowemu jako jedno zlecenie, nie ma potrzeby czekania na uzbieranie pewnej kwoty w jakiejś walucie przed przystąpieniem do kupna innej kwoty w innej walucie. Oczywiście pozostaje problem: skąd wziąć 5 USD (mln) aby wyspeku-

lować 6 USD (mln). Rozumosław Myślik, w zgodzie ze swą naturą, kontynuował eksperymenty symulacyjne, tym razem zakładając, że `A=0`, tzn. dla przypadku braku jakichkolwiek początkowych USD. Wynik był (z pominięciem kursów wymiany) następujący:

```
Poczatkowa ilosc USD (mln)  : 0.0
Koncowa ilosc USD (mln)  : 6.0
EUR_na_JPY : 0.0506145816160492
JPY_na_PLN : 5.56912241521389
PLN_na_USD : 20.9232
PLN_na_EUR : 0.211619565736702
```

gdzie GBP nie "uczestniczy" w wymianie,

Rozumosław Myślik wprost nie mógł uwierzyć własnym oczom: otrzymał 6 mln USD, jak się to mówi, ni stąd, ni zowąd (ang. *out of thin air*), bez jakiegokolwiek wkładu początkowego! Nic dziwnego, że ponownie sprawdził bilanse:

Bilans USD, patrz linia `/*35*/`:
```
  0+0+0+0+20.9232*3.4872 = 6+0+0+0+0
```
co daje:
```
 6 = 6.
```

Bilans EUR, patrz linia `/*36*/`:
```
  0+0+0+ 0.239177230327673*0.211619565736702 =
           0+0+0.0506145816160492+0
```
co daje:
```
  0.0506145816160492 = 0.0506145816160492.
```

Bilans GBP jest trywialny:
```
  0+0+0+0 = 0+0+0+0.
```

Bilans JPY, patrz linia `/*38*/`:
```
   0+110.03*0.0506145816160492+0+0 = 0+0+0+5.56912241521389
```
co daje:
```
  5.56912241521389 = 5.56912241521389.
```

Bilans PLN, patrz linia `/*39*/`:
```
 0+0+0+3.795*5.56912241521389 = 20.9232+0.211619565736702+0+0
```
co daje:
```
  21.13481956573 = 21.1348195657.
```

Ponownie wszystko jest O.K.: nowe finansowe *Perpetuum Mobile*[5] zostało odkryte! Niespokojny duch Myślika nie mógł jednak przejść do porządku dziennego nad faktem, że znów uzyskał tyle samo co poprzednio - 6 mln USD. Ponieważ ta liczba jest równa dosyć arbitralnie przyjętemu kresowi górnemu dziedziny końcowej ilości USD (mln) (patrz linia 6 programu), postanowił ten kres zmienić na 120 mln USD. A oto co otrzymał:

```
Poczatkowa ilosc USD (mln) : 0.0
Koncowa ilosc USD (mln) : 21.208105932626
EUR_na_USD : 3.0
EUR_na_JPY : 1.08782427718034
GBP_na_USD : 3.5
JPY_na_USD : 100.0
JPY_na_PLN : 19.6933052181531
PLN_na_USD : 40.0
PLN_na_EUR : 17.091193302891
PLN_na_GBP : 17.6449,
```

gdzie GBP już "uczestniczy" w wymianie, a wydruk kursów wymiany został ponownie pominięty. Tym razem Myślik wyczarował aż ponad 21 mln USD. I jedynym jego problemem jest znalezienie maklera walutowego, który przyjmie takie zlecenie od kogoś nie mającego przysłowiowego *złamanego* US centa przy duszy[6].

7.8 Zadania

Budowa fabryki

Wybitny Przedsiębiorca po latach prób i przemyśleń opracował rewelacyjny projekt i technologię produkcji mechaniczno-elektronicznego wihajstra (zwanego przez siebie *XYZ Gizmo*), który uważał za wypełniający istniejącą sporą lukę rynkową. Pierwszą rzeczą na drodze do niechybnego zalania rynku przez *XYZ Gizmo*sy było uzyskanie niezbędnego finansowania. Po 4 miesiącach starań i wydatkach 0.1 mln MU uzyskał kredyt banko-

[5] Hipotetyczna maszyna produkująca więcej energii aniżeli pobiera, bez względu na czas działania. Fizycy sa zgodni, że Perpetuum Mobile nie da się wykonać: jego istnienie byłoby negacją podstawowych praw fizyki.

[6] Matematyka stosowana przez Myślika jest bezbłędna, ale jego dane nie: Myślik zastosował niepoprawne kursy JPY_naPL i PL_naJPY.

wy w wysokości 6 mln JP. Kredyt ów był trzyletni i bardzo korzystnie oprocentowany w wysokości 7% rocznie. Niespłacenie kredytu w terminie pociągnie za sobą konieczność spłacenia pozostałości z oprocentowaniem 12% rocznie. Po upływie 4 miesięcy udało mu się zakupić potrzebny pod fabrykę teren za 1 mln JP. Natychmiast po kupnie terenu Wybitny Przedsiębiorca musiał płacić podatek od nieruchomości w wysokości 0.01 mln JP miesięcznie. Następnie należało opracować analizę wpływu inwestycji na środowisko, co może trwać do 1.5 roku. Niestety, inwestycja została oprotestowana przez Silną Grupę Zielonych, która stwierdziła obecność na zakupionym terenie szeregu kolonii pewnego typu mikrorobaczków, zagrożonych wyginięciem w skali globalnej. Skutkowało to trwającym 2 lata postępowaniem sądowym, zakończonym ugodą zobowiązującą Wybitnego Przedsiębiorcę do zapłacenia Silnej Grupie Zielonych 0.2 mln JP za wysiedlenie kolonii mikrorobaczków ze swej posiadłości. Wybitny Przedsiębiorca został również przez sąd zobowiązany do zainstalowania w trawie na swej posiadłości sieci mikrofonów i kamer monitorujących ewentualną obecność nie przesiedlonych mikrorobaczków i ewentualne naruszenie ich środowiska przez działalność produkcyjną, co kosztowało 0.05 mln JP w skali roku.Tylko wtedy Wybitny Przedsiębiorca mógł legalnie zbudować stację transformatorową i stację pomp wodnych oraz stację odbioru ścieków. Wymagało to trzech miesięcy i kosztowało 0.2 mln JP. Równolegle Wybitny Przedsiębiorca rozpoczął prace na tworzeniem ścieżek, dróg, rur odwadniających i pasów zieleni. Ponieważ jego posiadłość była blisko ważnej drogi, musiał zapłacić za modyfikacje jej odcinka. Zajęło to 5 miesięcy i kosztowało 0.2 mln JP. Następnie zlecił architektowi opracowanie dokumentacji budynków fabryki, co trwało 5 miesięcy i kosztowało 0.2 mln JP. Niestety, dokumentacja ta nie została zatwierdzona przez odpowiedni organ administracji z powodu protestu wyznawców modnego kultu Boo-Woo, domagających się uwzględnienia w dokumentacji Kaplicy Modlitw i Rozmyślań dla swoich wyznawców (jeżeli zostaliby zatrudnieni w fabryce), i z powodu protestów Związku Karmiących Matek, domagających się uwzględnienia w projektach Izby Wypoczynku i Opieki dla karmiących matek, które z pewnością zostaną zatrudnione w fabryce. Ostatecznie, po 3 miesiącach i licznych korektach, przy dodatkowym koszcie 0.5 mln JP, dokumentacja została zatwierdzona. Dopiero wtedy, po 3 latach od uzyskania kredytu, z którego nie spłacono dotąd pojedynczej JP, Wybitny Przedsiębiorca znalazł generalnego wykonawcę dla swej fabryki, który przyjął zlecenie jej zbudowania w ciągu 7 miesięcy za kwotę 3.4 mln JP. Równolegle Wybitny Przedsiębiorca rozpoczął zabiegi o dostawę energii

elektrycznej, dostawę wody i odbiór ścieków, którym to zabiegom towarzyszyło szereg inspekcji różnych instytucji. Inspekcje te kosztowały 0.3 mln JP i musiały być powtarzane co 3 miesięcy. Skoro fabryka została zbudowana, Wybitny Przedsiębiorca zaczął wyposażać ją w niezbędną aparaturę produkcyjną i sprzęt biurowy, co zajęło 3 miesiące i kosztowało 1.2 mln JP. Równolegle Wybitny Przedsiębiorca rozpoczął kompletowanie i szkolenie załogi, co ze względu na nadpodaż niezatrudnialnych bezrobotnych oraz protesty lokalnego stowarzyszenia LGBT oskarżającego go o dyskryminację przy zatrudnianiu, było przedsięwzięciem trudnym i czasochłonnym. Trwało to 6 miesięcy i kosztowało 0.1 mln JP. Wybitny Przedsiębiorca mógł więc wreszcie przystąpić do produkcji swych *XYZ Gizmo*sów, wprowadzić je na rynek i zacząć spłacać swoje długi. Los jednak chciał, że zupełnie przypadkiem znalazł się w hipermarkecie Technik-Markt, gdzie zobaczył półki wypełnione różnymi typami wihajstrów typu *XYZ Gizmo*, wyprodukowanymi przez Korporację Dalekowschodniego Imperium i oferowane po śmiesznie niskich cenach. Na ów widok Wybitny Przedsiębiorca doznał śmiertelnego ataku serca.

Napisz program wyjaśniający ile Wybitny Przedsiębiorca zużył czasu i pieniędzy na wybudowanie fabryki XYZ Gizmos do momentu swego nieszczęśliwego zgonu. Załóż, że żadne płatności z tytułu zaciągniętego kredytu nie zostały wykonane i wszystkie wydatki Wybitny Przedsiębiorca powyżej zaciągniętego kredytu 6 mln JP pochodziły z dodatkowych kredytów oprocentowanych na `12%` rocznie.

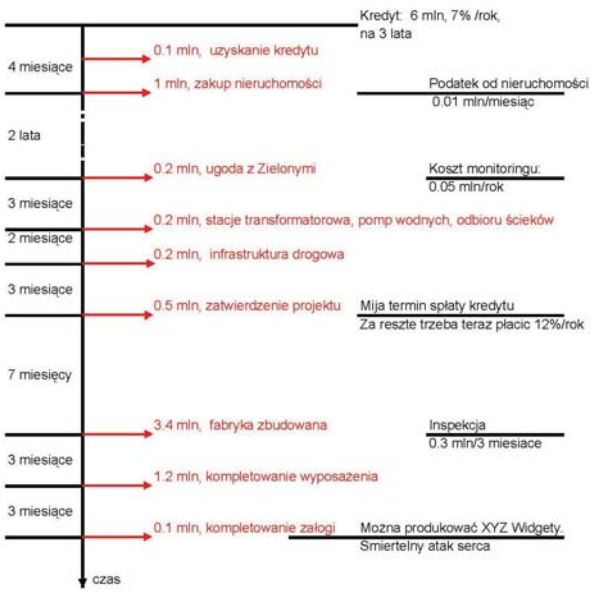
Dla lepszego zobrazowania struktury czasowej wydatków przedstawiono ją na rysunku 7.2.

Prywatne inwestycje

Prywatny inwestor chciałby zainwestować 15000 JP na najbliższy rok w dwóch inwestycjach: inwestycja `I1` zapewnia przychód `5%`, a inwestycja `I2` przychód `9%`. Doradca inwestycyjny sugeruje, by co najmniej `30%` zainwestować w `I1` i co najwyżej `55%` w `I2`. Ponadto, inwestycja `I1` powinna co najmniej być równa połowie inwestycji `I2`. Napisz program wyznaczający wielkość inwestycji maksymalizującą roczny przychód.

Kampania pijarowa

Znana partia *Wszystko dla Wszystkich* przystąpiła do kolejnego bałamucenia elektoratu za pomocą kampanii pijarowej w radiu i telewizji. Ma na to budżet w wysokości 15000 JP miesięcznie. Każda minuta bałamucenia



Rysunek 7.2: Struktura czasowa wydatków

radiowego kosztuje 15 JP, a każda minuta bałamucenia telewizyjnego - 300 JP. Centrala partyjna ceni bałamucenie radiowe dwa razy wyżej od bałamucenia telewizyjnego. Badania wykazały jednak, że:
- bałamucenie radiowe jest nieopłacalne powyżej 400 godzin miesięcznie;
- bałamucenie telewizyjne jest 25-krotnie bardziej efektywne od bałamucenia radiowego.
Napisz program wyjaśniający jak ulokować miesięczne fundusze na kampanię pijarową by zmaksymalizować jej efektywność.

Montowanie telefonów

Linia montażowa z czterema stacjami roboczymi służy do montowania dwóch telefonów, `Handy_1` i `Handy_2`. Dane linii przedstawia tablica 7.4.

Co-zmianowa konserwacja pochłania pewien procent całkowitego czasu pracy na zmianie (480 minut). Napisz program wyznaczający taką liczbę

Numer stacji	Czas montażu w minutach na telefon dla `Handy_1`	Czas montażu w minutach na telefon dla `Handy_2`	Zmianowa konserwacja w `%` od 480 minut
l	6	4	10
2	4	6	12
3	5	5	14
4	7	8	16

Tablica 7.4: Dane linii montażowej

telefonów `Handy_1` i `Handy_2` produkowanych na zmianie, która minimalizuje całkowity czas postoju linii montażowej w czasie zmiany.

Domy i apartamenty

Projekt osiedla Lotus Point Condo będzie obejmował zarówno domy jak i apartamenty. Rozmiary terenu wystarczają do zbudowania co najwyżej 10.000 jednostek mieszkaniowych. Osiedle musi posiadać obiekt rekreacyjny, którym może być jedno z dwojga: albo obiekt basenowo-tenisowy, albo marina dla żaglówek. Jeżeli powstanie marina, liczba domów powinna być co najmniej trzykrotnie większa od liczby apartamentów. Budowa mariny pochłonie 1.2 mln JP, a budowa obiektu basenowo-tenisowego - 2.8 mln JP. Deweloperzy uważają, że każdy apartament będzie można sprzedać za 48000 JP, a każdy dom za 46000 JP. Budowa domu lub apartamentu to koszt 40000 JP. Napisz program maksymalizujący zysk z budowy osiedla.

Komputery osobiste

Orange Co jest właścicielem czterech fabryk komputerów osobistych. Firma może sprzedać rocznie co najwyżej 20.000 komputerów w cenie 3500 JP każdy. Dane każdej fabryki przedstawia tablica 7.5.

Napisz program maksymalizujący roczne zyski z produkcji komputerów.

Budowa mostu

Należy opracować plan finansowania budowy mostu nad Dużą Zatoką. Tablica 7.6 przedstawia oceny rocznych kosztów dla 6-letniego planu budowy.

Miasto Zatokowe planuje zdobyć potrzebne fundusze na drodze emisji obligacji o okresie ważności nie dłuższym niż 6 lat. Obligacje można kupować

Numer fabryki	Zdolność produkcyjna	Koszty stałe fabryki (JP)	Koszt produkcji komputera (JP)
1	10000	9 mln	1000
2	8000	5 mln	1700
3	9000	3 mln	2300
4	6000	1 mln	2900

Tablica 7.5: Dane o produkcji komputerów

Rok	Koszt (mln JP)
1	20
2	17
3	23
4	24
5	25
6	21

Tablica 7.6: Roczne koszty konstrukcji mostu

każdego pierwszego stycznia z terminem wymagalności 31 grudnia tego roku, na który opiewa obligacja, zob tablica 7.7 [7].

Oprocentowanie obligacji jest wypłacane - wraz z kapitałem - w miesiącu jej wymagalności. Środki finansowe nie wykorzystane na finansowanie budowy mostu można zainwestować w Dużym Banku z roczną stopą procentową `6.8%`.

Napisz program określający wielkość emitowanych obligacji w sposób minimalizujący płatną wierzytelność po koniec roku 6-tego przy corocznym finansowaniu budowy mostu.

Autobusy

Dwie Zajezdnie Autobusowe (ZA1 i ZA2) obsługują cztery Stacje Autobusowe (SA1, SA2, SA3 i SA4). Tablica 7.8 przedstawia odległości pomiędzy zajezdniami i stacjami, liczbę dostępnych autobusów i zapotrzebowanie na autobusy. Napisz program przyporządkowujący autobusy z zajezdni sta-

[7] Przykład zaczerpnięty z ftp.math.tu-berlin.de/pub/Lehre/LinOpt/WS09/linoptWS09-08.pdf

Wymagalność obligacji, lata	Całkowita stopa procentowa, %
1	7
2	15
3	23
4	32
5	41
6	50

Tablica 7.7: Wymagalność i stopy procentowe obligacji

cjom autobusowym tak, by zminimalizować całkowitą trasę przejechaną pomiędzy zajezdniami i stacjami przy równoczesnym zaspokojeniu zapotrzebowania.

Zajezdnie	Stacje				Dostępne autobusy
	SA1	SA2	SA3	SA4	
ZA1	15	12	10	17	100
Za2	5	18	24	7	150
Zapotrzebowanie	40	65	45	60	

Tablica 7.8: Dane dla alokacji autobusów

Zarządzanie farmą

Rolnik może uprawiać na 100 hektarach pszenicę lub żyto, każda uprawa da inny zysk z hektara i wymaga różnych nakładów czasu pracy rolnika[8]. Rolnik dysponuje ograniczoną liczbą dni, które może poświęcić na pracę przy wymienionych uprawach. Napisz program wyznaczający maksymalny zysk ze 100 hektarów i 40 dni pracy rolnika, jeżeli zysk z ara pszenicy jest równy 2.5 JP, natomiast z ara żyta 3.5 JP, a stosunek czasu potrzebnego dla uprawy pszenicy do czasu potrzebnego dla uprawy żyta to 1:2.

Strategia pożyczek

Famous Bank przygotowuje strategię pożyczek na łączną kwotę nie przekraczającą 12 mln JP. Tablica 7.9[9] przedstawia dane odnośnie typów pożyczek udzielanych przez bank.

[8]Przykład pochodzi z http://www.ifcomputer.com/IFProlog/

[9]Przykład pochodzi z książki [Taha-08].

Rodzaj pożyczki	Oprocentowanie	Udział złych długów
Osobista	0.140	0.1
Samochodowa	0.130	0.07
Mieszkaniowa	0.120	0.03
Rolna	0.125	0.05
Komercyjna	0.100	0.02

Tablica 7.9: Dane dla strategii pożyczek

Zakłada się, że złe długi są nieodzyskiwalne i nie generują żadnych przychodów. Konkurencja z innymi bankami sprawia, że *Famous Bank* musi przeznaczyć co najmniej `40%` łącznej kwoty pożyczek na pożyczki rolne i komercyjne. Aby wspomóc lokalne budownictwo mieszkaniowe, pożyczki mieszkaniowe powinny stanowić co najmniej `50%` pożyczek osobistych, samochodowych i pożyczek mieszkaniowych. Bank przestrzega zasady nieprzekraczania przez złe długi `4%` wszystkich pożyczek. Napisz program maksymalizujący zysk netto banku, tzn. różnicę pomiędzy przychodami z oprocentowania pożyczek a stratami na złych długach.

Leczymy chorobę bezobjawową

Badacze-farmakolodzy słynnej korporacji *BigPharma* opracowali rewelacyjny środek uzależniający (*RSU*), nie generujący jakichkolwiek przyjemnych doznań w wyniku zażycia, lecz - po kilku zażyciach - wywołujący ogromny dyskomfort, któremu może zapobiec jedynie dalsze zażywanie tego środka. Rada Nadzorcza *BigPharmy* długo analizowała możliwości komercjalizacji *RSU*, akceptując ostatecznie projekt zwany *Leczeniem Choroby Bezobjawowej*: chorobę bezobjawową (zwaną z angielska *NSD - No Symptoms Disease*) zdefiniowano jako *absolutnie śmiertelne zagrożenia życia* Osób Normalnych*, którym pozornie nic nie dolega*, którzy cieszą się życiem i swoją pracą, prowadzą rozsądny tryb życia z umiarkowanym udziałem niekontuzyjnych sportów, nie palą papierosów, nie zażywają alkoholu ani narkotyków, odżywiają się racjonalnie stosując dietę niskowęglowodanową i wysokotłuszczową, unikają bezsensownych wydatków i stresorodnych zajęć, są zwolennikami (zakazanych przez prawo) zabiegów wodoleczniczych wg. księdza Kneippa. Poważnym problemem było skłonienie *Osób Normalnych* do "leczenia się" nowym specyfikiem. Pomoc okazały tu Media Głównego Nurtu, które szczodrze sponsorowane przez *BigPharmę*, wszczęły graniczącą z histerią kampanię medialną nagłaśniającą śmier-

telność *NSD*, publikując alarmujące dane o śmiertelnych zejściach osób cierpiących na *NSD* bez podawania ich wieku i krypto-reklamując skuteczność leczenia za pomocą *RSU*. Akcja zyskała ciche lecz wymierne poparcie *Departamentu Długowieczności* w *Światowym Instytucie Zdrowia, Szczęścia i Pomyślności*, którego urzędnicy od lat ubolewali nad aspołecznością osób cierpiących na *NSD*, gdyż ich tryb życia nie generuje tak potrzebnych dochodów podatkowych, generuje natomiast długowieczność stwarzającą realne zagrożenie dla rozmaitych funduszy ubezpieczeniowych i emerytalnych. Chcąc wykorzystać rentę monopolową, *BigPharma* szybko przystąpiła do opracowania technologii produkcji *RSU* w postaci tabletek, czopków, syropów, zastrzyków i plastrów, tak by każdemu zdiagnozowanemu na *NSD* umożliwić korzystanie z *RSU* w jego ulubionej formie. Opracowanie tych technologii napotkało jednak na pewne ograniczenia:

- *podstawowym surowcem* (*PS*) dla produkcji *RSU* okazał się być ekstrakt z pewnej rośliny, rosnącej tylko w *Słynnej Puszczy Tropikalnej*, którego dostawy nie mogą być większe niż 1000 kg miesięcznie.
- produkcja tabletek i czopków odbywa się na tej samej linii produkcyjnej, obłożonej również inną produkcją, i w związku z tym mogącej wyprodukować miesięcznie nie więcej niż 1000 standardowych opakowań tabletek i czopków łacznie;
- produkcja plastrów wymaga specjalnego materiału tekstylnego, który można nabyć w ilości 10 m^2 miesięcznie;
- produkcja zastrzyków i syropów wymaga tego samego rozpuszczalnika, który można nabyć w ilości 100 litrów miesięcznie.

Napisz program wyjaśniający, jakie liczby standardowych opakowań tabletek, czopków, syropów, zastrzyków i plastrów powinna produkować miesięcznie *BigPharma*, by uzyskać maksimum zysku, jeżeli wiadomo, że:

- dla wyprodukowania standardowego opakowania tabletek potrzeba 0.01 kg PS, dla wyprodukowania standardowego opakowania czopków - 0.015 kg PS, dla wyprodukowania standardowego opakowania syropu potrzeba 0.01 kg PS, dla wyprodukowania standardowego opakowania zastrzyków potrzeba 0.02 kg PS, a dla wyprodukowania standardowego opakowania plastrów - 0.012 kg PS;
- dla wyprodukowania standardowego opakowania plastrów potrzeba 10 cm^2 specjalnego materiału tekstylnego;

- dla wyprodukowania standardowego opakowania syropu potrzeba 0.001 litra rozpuszczalnika, dla wyprodukowania standardowego opakowania zastrzyków potrzeba 0.01 litra rozpuszczalnika,

natomiast zyski ze sprzedaży różnych produktów kształtują się następująco:

- standardowe opakowanie tabletek daje zysk równy 1.5 JP;
- standardowe opakowanie czopków daje zysk równy 1.8 JP;
- standardowe opakowanie syropu daje zysk równy 2.0 JP;
- standardowe opakowanie zastrzyków daje zysk równy 3.5 JP;
- standardowe opakowanie plastrów daje zysk równy 2.5 JP.

Posłowie

"...W naszym kraju - powiedziała wciąż jeszcze zdyszana Alicja - zwykle jest się innym miejscu, jeżeli biegało się tak szybko i długo, jak my biegłyśmy.
- Musi to być powolny kraj! powiedziała Królowa. - Bo tu, jak widzisz, trzeba biec tak szybko, jak się potrafi, żeby zostać w tym samym miejscu. Jeżeli chcesz znaleźć się w innym miejscu, trzeba biec co najmniej dwa razy szybciej!"
Lewis Carroll, *"O tym, co Alicja odkryła po drugiej stronie lustra"*

Pomimo zakresu materiału objętego książką i szybkości prezentacji tego materiału, poznaliśmy jedynie najbardziej podstawowe elementy ECL^iPS^e. Platforma ECL^iPS^e oferuje użytkownikowi znacznie, znacznie więcej, niż dało się przedstawić w tym elementarnym wprowadzeniu, skoncentrowanym na przedstawieniu najbardziej podstawowych idei CLP i najbardziej podstawowych zasad programowania na platformie ECL^iPS^e. Czytelnik mający w tej mierze jakieś wątpliwości jest proszony o przyjrzenie się zestawieniu standardowych predykatów wymienionych w *Alphabetical Predicate Index*, zob. rysunek 5, lub zestawieniu bardziej zaawansowanych przykładów, zob. [Kjellerstrand-13]. Tak na przykład przedstawiono bardzo pobieżnie sposoby sterowania poszukiwaniami, zostały pominięte milczeniem zaawansowane zagadnienia, np. grafika, interfejsowanie CLP z językami proceduralnymi, wyznaczanie rozwiązań suboptymalnych metodami heurystycznymi (metody genetyczne, przeszukiwania *tabu*, symulowane wyżarzanie). Zainteresowani czytelnicy zechcą szukać potrzebnej wiedzy na ciągle uaktualnianym portalu ECL^iPS^e oraz na forum dyskusyjnym ECL^iPS^e, i wreszcie w ciągle zwiększającej się liczbie przykładów, które można znaleźć w Internecie.

Celem książki było wyłącznie nauczenie podstaw *CLP* stosując ECL^iPS^e. Autor zawsze uważał, że nauczanie czegokolwiek nie jest "napełnianiem naczynia", lecz raczej "rozpaleniem ognia". Czytelnikom wypada osądzić, w jakim stopniu udało się ów ogień rozpalić. Autorowi wypada podkreślić, że pisanie tej książki było źródłem satysfakcji i radości.

Ważne pojęcia

Absurdolandia - całkowicie fikcyjne państwo, w którym większość spraw jest postawiona na głowie, będące miejscem, w którym rozgrywa się akcja szeregu zadań i przykładów tej książki.

AI - (ang. *Artificial Intelligence*), zob. *sztuczna inteligencja*.

Akumulator - pierwotnie pusta lista, do której dodawane są kolejne *głowy*.

Algorytm - metoda rozwiązywania problemu gwarantująca sukces.

Anonimowa zmienna - zob. *Zmienna - anonimowa*.

AoA - Activities on Arc, rodzaj sieci działań, w której akcje są przedstawiane ukierunkowanymi łukami.

Argument - zmienna w predykacie.

Arność - argumentowość, liczba zmiennych *krotki*, liczba argumentów *termu*.

Arność predykatu - liczba argumentów *predykatu*.

Arność relacji - liczba zbiorów w odpowiednim *iloczynie kartezjańskim*.

Arytmetyka przedziałowa - zasady operacji arytmetycznych na przedziałach liczb.

Asertować - przechować spełniony predykat w dynamicznej bazie danych.

Atom - nie-numeryczna (tzn. symboliczna lub logiczna) stała w *Prolog*u i *CLP* o zerowej *arności*, w postaci ciagu znaków rozpoczynających się małą literą lub sekwencją dowolnych znaków zawartą pomiędzy podwójnymi lub pojedynczymi cudzysłowami.

Badania operacyjne - interdyscyplinarna nauka i technologia stosująca matematyczne modelowanie, analizę statystyczną i metody optymalizacji dla uzyskania optymalnych lub quasi-optymalnych rozwiązań złożonych problemów decyzyjnych.

Baza wiedzy - plik tekstowy zawierający (w odpowiedniej syntaktycznie postaci) pełną *wiedzę* potrzebną *systemowi wnioskującemu* dla rozwiązania problemu decyzyjnego.

Benchmark - trudny problem obliczeniowy służący do testowania efektywności różnych sposobów jego rozwiązywania.

Błądzenie - wielokrotny powrót do stanów niedopuszczalnych.

Boolowska - zob. *zmienna - boolowska*.

Cel - celem programu prologowego lub clp-owego jest takie uziemienie zmiennych programu, aby *zapytanie* stało się *prawdą*.

Ciało reguły - koniunkcja warunków *reguły*.

Ciągła zmienna - zob. *Zmienna - ciągła*.

CLP - constraint logic programming, akronim programowania w logice z ograniczeniami.

clp-owy - charakterystyka programu zbudowanego zgodnie z zasadami *CLP*.

CPOSO - ciągły problem optymalnego spełnienia ograniczeń.

CPSO - ciągły problem spełnienia ograniczeń.

Cykl Hamiltona - zamknięta ścieżka, przechodząca przez każdy wierzchołek grafu tylko jeden raz.

Czas kwantyfikowany - czas występujący w ograniczeniach w postaci przedziałów o stałej długości, np. godziny i dni'

Czas niekwantyfikowany - czas występujący w ograniczeniach kolejnościowych w postaci czasów trwania czynności.

Czynność - synonim *operacji*.

Deklaratywne programowanie - paradygmat programowania wymagający wyłącznie odpowiedniego opisu rozwiązywanego problemu, nie zaś opisu sposobu jego rozwiązywania, tzn. przedstawienia odpowiedniego *algorytmu*.

Długość uszeregowania - makespan, różnica pomiędzy czasem zakończenia a czasem rozpoczęcia serii współbieżnych zadań.

Drzewo poszukiwań - graf-drzewo przedstawiający konkretny sposób poszukiwań dla określonej przestrzeni stanu.

Dyskretna zmienna - zob. *Zmienna - dyskretna*.

Dyzjunkcja - zdanie złożone postaci *q ; p*, będące prawdą tylko wtedy, gdy zdania `p` *lub* `q` są prawdą.

Dyzrówność - (disequation) relacja prawdziwa wtedy i tylko wtedy, gdy dwa *termy* nie są identyczne: `?Term1 \== ?Term2`.

Dziedzina - zbiór *wartości*, które można przyporządkować *zmiennej*.

Etykietowanie - synonim *uziemienia zmiennej*.

Fakt - predykat lub atom uważany za *prawdę*.

fail - predykat standardowy bezargumentowy, który jest zawsze niespełniony. Stosowany w *Prolog*u i *CLP* dla wymuszania nawrotów w celu znalezienia alternatywnych rozwiązań.

Funkcja - szczególny przypadek relacji dla zbioru n zmiennych o określonych dziedzinach. Funkcja przyporządkowuje unikalną wartość jednej zmiennej (zmiennej wyjściowej) z jej dziedziny każdej (n-1)-krotce pozostałych zmiennych (zmiennych wejściowych) z ich dziedzin.

Funktor - synonym *predykatu*, nie stosowany w tej książce.

Gałęzie i ograniczenia - (ang. branch-and-bound), ogólna metoda rozwiązywania problemów optymalizacji kombinatorycznej na drodze przeszukiwań w głąb z nawrotemi.

Gantt'a wykres - graficzna prezentacja alokacji zasobów w czasie dla szeregu współbieżnie realizowanych zadań.

Głowa reguły - wniosek reguły.

Głowa listy - pierwszy element listy.

GPS - (ang. *General Problem Solver*), zob. *uniwersalny rozwiązywacz problemów*.

Harmonogramowanie - określenie przebiegu szeregu zadań w czasie z wyznaczeniem dla każdej z nich czasu początku i czasu końca, z uwzględnieniem *ograniczeń kolejnościowych*, *ograniczeń nierównoczesności* pomiędzy zadaniami oraz *ograniczeń zasobów* potrzebnych do wykonywania zadań.

Herbranda dziedzina - dziedzina *termów*. Stała jest termem Herbranda, zmienna jest termem Herbranda, predykat o arności n, którego argumenty są termami Herbranda jest termem Herbranda, lista której elementy są termami Herbranda jest termem Herbranda.

Heurystyka - metoda rozwiązywania problemu nie gwarantująca sukcesu.

Heurystyka wyboru wartości - heurystyka określająca kolejność wyboru *wartości* zmiennej z jej *dziedziny* w celu uziemienia tej zmiennej w trakcie *poszukiwań*.

Heurystyka wyboru zmiennej - heurystyka określająca kolejność wyboru *zmiennych* w trakcie *poszukiwań*.

i - sposób czytania funktora koniunkcji, zapisywanego w *Prologu* i *CLP* w postaci przecinka (,).

IC - solwer dla działań na liczbach całkowitych i rzeczywistych.

Iloczyn kartezjański - zob. *kartezjański iloczyn*.

Imperatywne programowanie - paradygmat programowania wymagający tworzenia algorytmów potrzebnych dla rozwiazywania problemów.

Infiksowa notacja - nazwa predykatu jest pisana pomiedzy argumentami.

Iteracja - wielokrotne stosowanie predykatu dla kolejnych danych w pętli.

Job-shop - zob. *linia zadań*.

JP - Jednostka Pieniężna, fikcyjna jednostka monetarna stosowana dla przykładów w książce.

Kartezjański iloczyn *n* zbiorów - zbiór wszystkich możliwych n-*krotek*, których pierwszy element należy do pierwszego zbioru,...itd.,...,`n`-ty element nalezy do `n`-tego zbioru.

Klauzula - zdanie zakończone kropką, będące podstawowym elementem programów prologowych i clp-owych. Klauzula może być *faktem* lub *regułą*.

Kombinatoryczna eksplozja - bardzo szybki wzrost liczby elementów *przestrzeni stanu* problemu ze wzrostem liczby zmiennych.

Konfigurowanie - wyznaczenie - z zadanego zbioru - takiego podzbioru, którego elementy spełniają zadane ograniczenia.

Koniunkcja - zdanie złożone postaci *q, p*, będące prawdą tylko wtedy, gdy zdania `p` *i* `q` są prawdą.

Krotka - uporządkowany zbiór elementów.

Krytyczna ścieżka - najkrótsza sekwencja działań w projekcie, poczynając od działania początkowego a kończąc na działaniu końcowym.

Linia zadań - ang. *Job-shop* - specyficzny problem harmonogramowania, dla którego szereg *zadań* (jobs) składających się z określonych *operacji* (tasks) wymagających określonej kolejności wykonywania, należy wykonać *współbieżnie* za pomocą określonego zbioru *maszyn* (processors).

Lista - *krotka* rozpoczynająca się lewostronnym kwadratowym nawiasem `([)` i kończąca się prawostronnym kwadratowym nawiasem `(])`.

Logika - nauka o tym, co z czego wynika.

LP - akronim "Linear Programming" lub "Linear Problem", patrz *Programowanie liniowe*.

lub - sposób czytania funktora dyzjunkcji, zapisywanego w *Prolog*u i *CLP* w postaci średnika (*;*).

Maszyna - (ang. *processor*) dowolny byt służący do wykonania jakiejś operacji.

Matematyczne programowanie - techniki numeryczne dla rozwiązywania problemów *programowania liniowego*, problemów *programowania całkowitoliczbowego* i problemów *programowania mieszanego*.

Metoda gałęzi i ograniczeń - (ang. *branch-and-bound*), obszerny zbiór metod poszukiwań z nawrotami, których celem jest optymalizacja pewnego wskaźnika jakości przy ograniczeniach kombinatorycznych.

Metoda wnioskowania - metoda wykrywania informacji ukrytej w danych.

Metoda wnioskowania - kompletna - metoda gwarantująca wyznaczenie rozwiązanie dla PSO, jeżeli takie rozwiązanie istnieje.

Metoda wnioskowania - niekompletna - metoda nie gwarantująca wyznaczenie rozwiązanie dla PSO, nawet jeżeli takie rozwiązanie istnieje.

MIP - akronim "Mixed Integer Programming" lub "Mixed Integer Problem", patrz *programowanie mieszane*.

Mod zmiennej - charakter zmiennej będącej argumentem standardowego predykatu (wejście, wyjście, wejście ukonkretnione, wejście uziemione).

Modelowanie - tłumaczenie werbalnego opisu problemu na program prologowy lub clp-owy.

Nawrót - proces *odziemienia* ostatnio *uziemionej zmiennej* z równoczesnym przejściem do najbliższego *predykatu powrotu* w celu uzieminia zmiennej swobodnej tego predykatu.

Nawrót - standardowy - *nawrót* inicjowany w wyniku pojawienia się pustej dziedziny dla którejkolwiek zmiennej.

Nawrót ze sprawdzeniem dwóch kroków w przód - *nawrót* inicjowany w wyniku predykcji pojawienia się pustej dziedziny dla którejkolwiek zmiennej w drugim kolejnym kroku poszukiwań.

Nawrót ze sprawdzeniem kroku w przód - *nawrót* inicjowany w wyniku predykcji pojawienia się pustej dziedziny dla którejkolwiek zmiennej w następnym kroku poszukiwań.

Nazwa/arność - referencja predykatu standardowego na drodze podania jego nazwy i arności.

Nazwa zmiennej - jakikolwiek ciąg liter alfabetu łacińskiego rozpoczynający się literą dużą lub podkreślnikiem.

Nazwa predykatu - jakikolwiek ciąg liter alfabetu łacińskiego rozpoczynający się literą małą.

Niespełniony - mający wartość logiczna *nieprawda*.

Nieprawda - element dziedziny *zmiennych boolowskich*, wskazujący, że zmienna ta nie jest prawdą.

Niezgodność - pojawienie się pustej dziedziny dla jednej lub kilku zmiennych w trakcie poszukiwań i propagacji ograniczeń.

Numer - stała całkowita (np. 9 lub 123) albo stała zmienno-przecinkowa w notacji inżynierskiej (np. 3.14 lub 2.79), wyłącznie z kropką dziesiętną.

Odkomentowanie - usunięcie znaku **%** czyniącego linię komentarzem.

Odziemienie zmiennej - uczynienie zmiennej uziemionej ponownie zmienną wolną z równoczesnym powrotem jej wartości do jej domeny.

Ograniczenia przynależności - związki łączące elementy różnych zbiorów.

Ograniczenie - *relacja* dla zbioru *zmiennych*, określajaca kombinacje wartości z dziedzin tych zmiennych, do których zmienne te mogą być *uziemione*.

Ograniczenie - aktywne - ograniczenie inicjujące propagację i poszukiwania.

Ograniczenie - ciągłe - ograniczenie obejmujące wyłącznie zmienne ciągłe.

Ograniczenia czasowe - ograniczenie określające czas potrzebny dla wykonania zadania.

Ograniczenie - dyzjunktywne - ograniczenie wymagające wykonania dwóch różnych *operacji* za pomocą tego samego *zasobu*.

Ograniczenie - kolejnościowe - ograniczenie definiujące kolejność wykonywania zadań.

Ograniczenie - kombinatoryczne - ograniczenie obejmujące wyłącznie zmienne dyskretne.

Ograniczenie nierównoczesnosci - ograniczenie zakazujące wykonywania pewnych czynnosci w tym samym czasie.

Ograniczenie - niespełnione - ograniczenie, którego zmienne są uziemione w sposób nie spełniający to ograniczenie.

Ograniczenie - pasywne - ograniczenie stosowane jako test w przypadku gdy wszystkie jego zmienne są uziemione.

Ograniczenie - propagacja - zob. *Propagacja ograniczenia*.

Ograniczenie - przynależnosci - ograniczenie określające przynależność do jakiegoś zbioru.

Ograniczenie - sąsiedztwa - ograniczenie określajace rodzaj sąsiadowania elementów jakiegoś zbioru.

Ograniczenie - spełnione - ograniczenie, którego wszystkie zmienne są uziemione w sposób spełniający to ograniczenie.

Operacja - (task) elementarna nieprzerywalna czynność, będąca składową *zadania*.

Optymalna trajektoria stanów - dopuszczalna trajektora stanów optymalizująca jakiś wskaźnik jakosci.

Optymalizacja - najlepszy sposób korzystania z ograniczonych zasobów (kapitał, zdolności produkcyjne, czas).

OS - optymalny stan, zob. *stan - optymalny*.

OTS - optymalna trajektoria stanów, zob. *optymalna trajektoria stanów*.

Permutacja zbioru n-elementowego - dowolna n-*krotka* zawierająca w/w elementy w określonym porządku.

Plecakowy problem - dokonanie takiego wyboru przedmiotów, by ich sumaryczna wartość była jak największa i jednocześnie mieściły się w plecaku.

Porażka - uziemienie predykatu nie jest prawdą.

Poszukiwanie i propagacja - naprzemiennie wykonywanie *poszukiwań* i *propagacja ograniczeń*.

Poszukiwanie - przegląd zupełny - kolejne generowanie wszystkich stanów przestrzeni stanu i ich testowanie.

Poszukiwanie w głąb ze standardowymi nawrotami - następujący ciąg kroków: 1)uziemienie wybranej *zmiennej decyzyjnej*, 2)testowanie *ograniczeń* z tą zmienną, 3)niespełnienie jakiegokolwiek ograniczenia inicjuje *nawrót*, 4)spełnienie wszystkich ograniczeń skutkuje wyborem kolejnej zmiennej decyzyjnej i jej uziemieniem. Poszukiwanie jest kompletną *metodą wnioskowania*.

Poszukiwanie ze sprawdzaniem kroku w przód - nawrót w *CLP* inicjowany w sytuacji, gdy ostatnie uziemienie zmiennej generuje dla jakiejś innej zmiennej pustą dziedzinę.

Poszukiwanie ze sprawdzaniem dwóch kroków w przód - nawrót w *CLP* inicjowany w sytuacji, gdy ostatnie uziemienie zmiennej umożliwia predykcję pojawienia się pustej dziedziny dla jakiejś innej zmiennej po dwóch kolejnych uziemieniach.

PSO - problem spełnienia ograniczeń.

POSO - problem optymalnego spełnienia ograniczeń.

Prawda - element dziedziny *zmiennych boolowskich*, wskazujący, że zmienna ta jest prawdą.

Predykat - *relacja* o określonej nazwie pomiędzy uporządkowanymi *zmiennymi* zwanymi *argumentami*, zdefiniowanymi albo za pomocą innych predykatów, albo na drodze zdefiniowania domen argumentów. Predykaty są podstawowym sposobem deklarowania ograniczeń w *Prolog*u i *CLP*.

Predykat - elementarny - predykat standardowy definiujący podstawowe relacje z bibliotek `ic` lub `branch_and_bound`, którego *argumenty* mogą być zawarte w jednej *liście*.

Predykat - globalny - predykat standardowy definiujący złożone relacje z bibliotek `ic_global`, `ic_cumulative`, `ic_edge_finder`, `ic_edge_finder3`, którego *argumenty* mogą być zawarte w szeregu *listach*.

Predykat - niespełniony - uziemiony predykat będący nieprawdą.

Predykat - prywatny - predykat zdefinowany przez twórcę programu prologowego lub clpowego, o nazwie różnej od nazw p*redykatów standardowych*.

Predykat - spełniony - *predykat uziemiony* będący prawdą.

Predykat - standardowy - predykat zdefiniowany i zaprojektowany przez twórców *Prolog*u lub języków *CLP*, udostępniony użytkownikom platformy ECL^iPS^e.

Predykat - uziemiony - predykat o wszystkich *zmiennych uziemionych*.

Predykat - wbudowany - zob. *predykat standardowy*.

Predykat - zagnieżdżony - predykat będący argumentem innych predykatów.

Prefiksowa notacja - nazwa predykatu jest pisana przed ujętymi w nawias argumentami predykatu.

Problem komiwojażera - problem optymalizacji kombinatorycznej polegający na znalezieniu najkrótszego cyklu Hamiltona łączącego wierzchołki grafu.

Proceduralne programowanie - zob. *Imperatywne programowanie*.

Procent składany - odsetki za dany okres są doliczane do wkładu i procentują wraz z nim w okresie następnym.

Programowanie całkowitoliczbowe - zbiór metod numerycznych dla optymalizacji liniowego *wskaźnika jakości* przy liniowych ograniczeniach całkowitoliczbowości i nieujemności zmiennych decyzyjnych.

Programowanie kwadratowe - zbiór metod numerycznych dla optymalizacji kwadratowego *wskaźnika jakości* przy liniowych ograniczeniach całkowitoliczbowości i nieujemności zmiennych decyzyjnych.

Programowanie liniowe - zbiór metod numerycznych dla optymalizacji ciągłego liniowego *wskaźnika jakości* przy liniowych ciągłych ograniczeniach równościowych i/lub nierównościowych dla ciągłych nieujemnych zmiennych decyzyjnych.

Programowanie mieszane - zbiór metod numerycznych dla optymalizacji liniowego *wskaźnika jakości* przy ograniczeniach równościowych i/lub nierównościowych dla ciągłych i całkowitoliczbowych nieujemnych zmiennych decyzyjnych.

Programowanie z wyodrębnioną bazą wiedzy - programowanie korzystające z wiedzy dziedzinowej zawartej w wydzielonej części programu (w bazie wiedzy).

Prolog - *Pro*gramming in *log*ic, akronim programowania w logice.

Propagacja autonomiczna - propagacja ograniczenia, która może dooprowadzic do rozwiazania pronblemu bez stosowania poszukiwań.

Propagacja ograniczenia - przeniesienie skutków uziemienia zmiennej w aktywnym ograniczeniu do innych elementów programu.

Propagacja ograniczenia w CLP - proces modyfikacji przez aktywne ograniczenie dziedzin tych wszystkich zmiennych, które w tym ograniczeniu występują. Istotą propagacji jest usunięcie z w/w dziedzin tych wszystkich wartości, które naruszają ograniczenie. Propagacja ograniczeń w CLP może być autonomiczna, lecz nie jest kompletną *metodą wnioskowania*.

Propagacja ograniczenia w Prologu - proces[powodujący, że uziemienie zmiennej w wyniku *unifikacji* dla predykatu jakiejś reguły jest powtórzone dla wszystkich instancji tej zmiennej w ciele tej reguły i dla wszystkich instancji tej zmiennej w innych predykatach. Propagacja ograniczenia w Prologu nie może być autonomiczna.

Propagacja wartości zmiennej - uziemienie dokonane dla *zmiennej predykatu* w jakiejś *regule* jest powtórzone dla wszystkich egzemplarzy tej zmiennej w *ciele* reguły.

Przegląd zupełny - zob. *Poszukiwania - przegląd zupełny*.

Przepaść semantyczna - różnica pomiędzy werbalnym sformułowania problemu optymalizacji a programem komputerowym rozwiązującym ten problem.

Przestrzeń stanu - pełna - wszystkie przyporządkowania wartości z *dziedzin* wszystkim *zmiennym decyzyjnym*.

Przestrzeń stanu - częściowa - niektóre przyporządkowania wartości z *dziedzin* niektórym *zmiennym decyzyjnym*.

Przestrzeń poszukiwań - zob. *przestrzeń stanu*.

Przyporządkowanie - wyznaczenie dla każdego elementu jednego zbioru jakiegoś elementu innego zbioru w sposób spełniający zadane ograniczenia.

Punkt wyboru - wierzchołek drzewa poszukiwań odpowiadający predykatowi mającemu co najmniej jedną zmienną nieuziemioną do jakiejś wartości ze swojej dziedziny, służący do powrotu w przypadku porażki w trakcie poszukiwań.

Q.E.D - akronym łacińskiej frazy *"Quod errat demonstrandum"*, oznaczającej *"czego należało dowieść"*; stosowany na końcu dowodów i argumentów.

QP - akronim "Quadratic programming", patrz *programowanie kwadratowe*.

Rekurencja - definiowanie *predykatu* jako *głowy reguły*, w *ciele* której występuje ten sam *predykat*.

Rekurencja ogonowa - rekurencja, dla której ostatnim warunkiem w *ciele reguły* definiującej ten predykat jest tenże predykat.

Relacja - dowolny uporządkowany podzbiór *iloczynu kartezjańskiego* pewnych zbiorów. Relacja dziedziczy uporządkowanie po iloczynie kartezjańskim.

Reuziemienie zmiennej - przyporządkowanie zmiennej wolnej co najmniej drugiej wartości z jej dziedziny.

Reguła - klauzula warunkowa o postaci: *Jeżeli Warunki są prawdą, to Wniosek jest prawdą*, gdzie *Wniosek* jest *nieuziemienioną zmienną logiczną* lub *nieuziemionym predykatem* zwanym *głową* reguły, zaś *Warunki* są koniunkcją *uziemionych* lub nieuziemionych zmiennych i predykatów, zwaną *ciałem* reguły.

Resurs - wszystko (maszyny, ludzie, energia, surowce) czego potrzeba dla wykonania *operacji*.

Rozkładowanie - (ang. *timetabling*) rozmieszczenie zadań (ze zbioru zadań) w różnych przedziałach czasu (ze zbioru przedziałów czasu).

Rozstrzygalność - właściwość problemu polegająca na możliwości określenia poprawności jego rozwiązania.

SD - zob. stan dopuszczalny.

SO - zob. stan optymalny.

Semantyka - wiedza o znaczeniu symboli i zdań w języku.

Solwer - rozwiązywacz, program do rozwiązywania problemów optymalizacji lub układów równań liniowych/nieliniowych, zintegrowany z systemem *CLP*.

Spełnienie - uziemienie predykatu jest prawdą.

Spełniony - mający wartość logiczną *prawda*.

Stała - atom, liczba całkowita, liczba rzeczywista, lista lub tabela atomów, tabela liczb całkowitych lub tabela liczb rzeczywistych.

Stan - całkowity - dowolne uziemienie wszystkich *zmiennych decyzyjnych*.

Stan - częściowy - dowolne uziemienie niektórych *zmiennych decyzyjnych*.

Stan - dopuszczalna trajektoria - zob. *Trajektoria stanów dopuszczalna*.

Stan - dopuszczalny - każde uziemienie zmiennych decyzyjnych spełniające wszystkie ograniczenia.

Stan - niedopuszczalny - każde uziemienie zmiennych decyzyjnych naruszające jakieś ograniczenia.

Stan - optymalna trajektoria - dopuszczalna trajektoria stanów optymalizująca zadany wskaźnik jakości.

Stan - optymalny - taki stan dopuszczalny, dla którego jakiś wskaźnik jakości osiąga wartość optymalną.

Stwierdzenie - (ang. *statement*), *uziemiony predykat*, *spełniony* lub *niespełniony*.

String - każdy ciąg dowolnych znaków zawartych w podwójnych cudzysłowach górnych.

Sukces - synonim *spełnienia*, uziemiony predykat jest prawdą.

Syntaks - wiedza o dopuszczalnych strukturach, w jakich mogą występować symbole języka.

Syntaktyczna równoważność - identyczność struktury dwóch termów, identyczność zmiennej i termu, oznaczona symbolem '='.

System wnioskujący - część kompilatora języka *Prolog* lub kompilatora *CLP*, służąca do wyciagania wnioskoów z bazy wiedzy programu.

Szeregowanie - wyznaczenie kolejności zadań spełniających *ograniczenia kolejnościowe*, *ograniczenia czasowe* i *ograniczenia nierównoczesności*.

Sztuczna inteligencja - dziedzina informatyki zmierzająca do emulowania inteligencji ludzkiej.

Ścieżka krytyczna - najkrótszy ciąg zadań (dla projektu wielozadaniowego) poczynając od zadania początkowego a kończąc na zadaniu końcowym.

Tablica - sposób zapisu wektorów i macierzy.

Tautologia - stwierdzenie prawdziwe na mocy znaczenia słów w nim występujących, np. "*Jeżeli mam forsę, to mam szmal*".

Techniki zgodnościowe - algorytmy, które modyfikują początkowe *dziedziny* zbioru zmiennych całkowitoliczbowych tak, by spełniały ograniczenia wiążące te zmienne. Stosowane dla *propagacji ograniczeń* w *CLP*. Techniki zgodnościowe nie są kompletną *metodą wnioskowania*.

Term - podstawowa struktura danych *Prologu* i *CLP*: ciąg znaków pozbawionych semantycznego znaczenia, mogący być *atomem*, *zmienną*, *liczbą*, *predykatem*, *listą*.

Termy syntaktycznie równoważne - termy tego samego typu o tej samej budowie; zmienna wolna i dowolny inny term.

Testowanie ograniczeń - sprawdzenie, czy ograniczenie o wszystkich zmiennych uziemionych jest, czy też nie jest spełnione.

top. - główne *zapytanie* programów prologowych i clp-owych w tej książce.

Tryb wiersza polecenia - tryb wykonywania programów *CLP* z wykorzystaniem DOS-o podobnego okna polecenia.

Trajektoria stanów - ciąg kolejno występujących stanów rozpoczynajacy się stanem początkowym i kończący się stanem końcowym.

Trajektoria stanów dopuszczalna - trajektoria stanów obejmująca tylko *stany dopuszczalne*.

Trajektoria stanów optymalna - taka *trajektoria stanów dopuszczalna*, która optymalizuje jakis *wskaźnik jakości*.

TSD - trajektoria stanów dopuszczalna.

TSO - trajektoria stanów optymaalna.

Ukonkretnienie zmiennej - unifikacja zmiennej z predykatem, listą, atomem lub liczbą.

Unifikacja - uziemianie zmiennych doprowadzające do równości dwóch termów.

Uniwersalny rozwiązywacz problemów - hipotetyczny program do rozwiązywania wszelkich problemów, bazujący tylko na logice.

Urzeczowienie - (reifikacja) powiązanie *ograniczenia* ze zmienną *logiczną*, uziemianą do `1` jeżeli ograniczenie jest spenione, lub do `0` w przypadku przeciwnym.

Uwolnienie zmiennej - uczynienie zmiennej uziemionej zmienną wolną.

Uziemienie predykatu - przyporządkowanie argumentom predykatu *wartości* z ich *dziedzin*.

Uziemienie zmiennej - unifikacja zmiennej wolnej z *wartością* z jej *dziedziny*. Zob. również *etykietowanie*.

Uziemiona zmienna - zob *zmienna uziemiona*.

Warunek reguły - *zmienna logiczna* lub nieuziemiony *predykat* współdecydujący o prawdziwości *wniosku reguły*, element *ciała* reguły.

Wartość - *stała* prologowa lub clp-owa.

Wartość bieżąca netto - (ang. *NPV*, *Net Present Value*) ocena przyszłych zysków (strat) w kAtegoriach ich wartości aktualnych.

Wartość logiczna - stałe *prawda* i *nieprawda*.

Wejście predykatu - zmienna określona na zewnątrz predykatu, w którym występuje.

Wejście programu - zmienna określona przez użytkownika programu prologowego lub clp-owego, w którym występuje.

Wiedza - takie rozumienie jakiejś dziedziny, które umożliwia podejmowanie racjonalnych decyzji w jej obrębie.

Wniosek - reguła - synonim głowy reguły.

Wskaźnik jakości - funkcja *zmiennych decyzyjnych*, przedstawiająca koszt, zysk lub stratę, optymalizowana (minimalizowana lub maksymalizowana) w trakcie rozwiązywania *PSO* lub *POSO*.

Współbieżność - równoczesność wykonywania kilku ciągów *operacji* (kilku *zadań*).

Wyjście predykatu - zmienna określona przez predykat, w którym występuje.

Wyjście programu - zmienna określona przez program prologowy lub clp-owy, w którym występuje.

Wyliczenie - wyznaczenie wszystkich elementów zbioru.

Zadanie - (ang. *job*) ciąg *operacji* (ang. *tasks*), które należy wykonać w zadanej kolejności za pomocą określonych *maszyn* (ang *processor*).

Zakomentowanie - wprowadzenie przed linię programu znaku **%** czyniącego ją komentarzem.

Założenie zamkniętego świata - założenie, że *głowa reguły* niespełnionej jest nieprawdą.

Zgodnościowe techniki - zob. *techniki zgodnościowe*.

Zapytanie - *głowa reguły*, którą należy spełnić. Zapytania służą do aktywacji programów prologowych i clp-owych.

Zasób - zob. *resurs*.

Zmienna - cokolwiek co ma *nazwę* i *dziedzinę*.

Zmienna - anonimowa - zmienna, której nie należy uziemiać.

Zmienna - boolowska - *zmienna*, w której domenie są tylko dwie wartości, *prawda* i *nieprawda* (fałsz).

Zmienna - ciągła - zmienna o dziedzinie ciągłej.

Zmienna - decyzyjna - zmienna niezbędna dla modelowania ograniczeń w PSO, POSO, CPSO i CPOSO.

Zmienna - dyskretna - zmienna o skończonej dziedzinie.

Zmienna - logiczna - zob. *zmienna boolowska*.

Zmienna - niedecyzyjna - zmienna niepotrzebna dla modelowania ograniczeń w PSO, POSO, CPSO i CPOSO, ale np. potrzebna dla zdefiniowania *akumulatora* lub dla wyświetlania komunikatu o wynikach wnioskowania.

Zmienna - nienumeryczna - zmienna logiczna lub symboliczna.

Zmienna - odziemiona - zmienna, która przestaje być *uziemioną* stając się ponownie *wolna*.

Zmienna - wolna - zmienna, której nie jest przyporządkowana żadna wartość z jej dziedziny

Zmienna - symboliczna - zmienna przedstawiająca atrybut nienumeryczny i nielogiczny, np. kolor, zapach,

Zmienna - reuziemiona - zmienna, której (aktualnie) została przyporządkowana co najmniej druga wartość z jej dziedziny.

Zmienna - ukonkretniona - zmienna, której w programie został przyporządkowany predykat (uziemiony lub nieuziemiony), lista (uziemiona lub nieuziemiona), atom lub liczba.

Zmienna - uziemiona - zmienna, której została przyporządkowana wartość z jej dziedziny.

Zmienna - wolna - zmienna, z którą nie jest związana *stała*.

Zmienna związana z atomem - zmienna, której wartość określa ów atom.

Zmienna związana z liczbą - zmienna, której wartość określa owa liczba.

Zmienna związana z listą - zmienna, której wartość określa owa lista.

Zmienna związana z predykatem - zmienna, której wartość określa ów predykat.

Słownik angielsko-polski

Accumulator - akumulator.
AI - SI, sztuczna inteligencja.
Algorithm - algorytm.
Anonymouse variable - zmienna anonimowa.
AoA - Akcje na łukach.
Arity - arność.
Arity - of predicate - arność predykatu.
Arity - of relation - arność relacji.
Array - tablica.
Assignment - przyporządkowanie.
Assert - asertować.
Atom - atom.
Backtracking - nawrót.
Backtracking - Forward Checking - nawrót ze sprawdzeniem kroku w przód.
Backtracking - Looking Ahead - ze sprawdzeniem dwóch kroków w przód.
Backtracking - Standard - nawrót standardowy.
Belongness constraint - ograniczenie przynależności.
Body - ciało reguły.
Boolean - boolowski.
Built-in - wbudowany.
Branch-and-bound - metoda gałęzi i ograniczeń.
Cartesian product - iloczyn kartezjański.
CCOP - CPOSO, ciągły problem optymalnego spełnienia ograniczeń.
CCSP - CPSO, ciągły problem spełnienia ograniczeń.
Choice point - punkt powrotu.

Clause - klauzula, zdanie.
Closed World Assumption - założenie zamkniętego świata.
Combinatorial explosion - eksplozja kombinatoryczna.
Command mode - tryb wiersza polecenia.
Compound interest - procent składany.
Configuration - konfiguracja.
Conjunction - koniunkcja.
Constant - stała.
Consistency techniques - techniki zgodnościowe.
Constraint - ograniczenie.
Constraint - active - ograniczenie aktywne.
Constraint - consistent - ograniczenie zgodne.
Constraint - elementary - ograniczenie elementarne.
Constraint - global - ograniczenie globalne.
Constraint - passive - ograniczenie pasywne.
Constraint propagation - propagacja ograniczenia.
Continuous variable - zmienna ciągła.
COP - POSO, problem optymalnego spełnienia ograniczeń.
CPS - Constraint Programming System, pakiet programowania w logice z ograniczeniami.
Crew scheduling - rozkładowanie dla załóg.
Critical path - ścieżka krytyczna.
CSP - PSO, problem spełnienia ograniczeń.
Cumulative constraint - ograniczenie na wspólny resurs.
Decision variable - zmienna decyzyjna.
Declarative programming - programowanie deklaratywne.
Decomment - odkomentowanie.
Degrounding a variable - odziemienie zmiennej.
Direct enumeration - przegląd zupełny.
Discrete variable - zmienna dyskretna.
Disequation - dyzrówność.
Disjunction - dyzjunkcja.
Disjunctive constraint - ograniczenie dyzjunktywne.
Domain - dziedzina.
Exhaustive search - przegląd zupełny.
Fact - fakt.

fail - ponieść porażkę.
Failure - porażka.
Forward checking - poszukiwanie ze sprawdzaniem kroku w przód.
Function - funkcja.
FS - SD, stan dopuszczalny.
FST - TSD, trajektoria stanów dopuszczalna.
Functor - funktor.
Gantt chart - wykres Gantta.
General Problem Solver - uniwersalny rozwiązywacz problemów.
Goal - cel.
Grounding of variable - uziemienie zmiennej.
Ground predicate - uziemiony predykat.
Ground variable - uziemiona zmienna.
Head - głowa reguły, głowa listy.
Heuristic - heurystyka.
Imperative programming - programowanie imperatywne.
Inconsistency - niezgodność.
Inference methods - metody wnioskowania.
Inference methods - complete - metody wnioskowania kompletne.
Inference methods - uncomplete - metody wnioskowania niekompletne.
Inference system - system wnioskujący.
Infix notation - notacja infiksowa.
Input of predicate - wejście predykatu.
Input of program - wyjście programu.
Instantiated - ukonkretnienie zmiennej.
Integer Programming - programowanie całkowito-liczbowe.
Interval arithmetic - arytmetyka przedziałowa.
Iteration - iteracja.
Job - zadanie.
Job-shop - linia zadań.
Knowledge - wiedza.
Knowledge base - baza wiedzy.
Knowledge based programming - programowanie z wyodrębnioną bazą wiedzy.
Labeling - etykietowania, uziemianie.
List - lista.

Logic - logika.
Logical values - wartość zmiennej logicznej.
Logical variable - zmienna logiczna.
LP - programowanie liniowe.
Makespan - długość uszeregowania.
Mathematical programming problems - problemy programowania matematycznego.
Mode of variable - mode zmiennej.
Modeling - modelowania.
MP - programowanie mieszane.
MU - JP, Jednostka Pieniężna.
Name/arity - nazwa predykatu/arność predykatu.
Name of variable - nazwa zmiennej.
Name of predicate - nazwa predykatu.
Neighbourhood constraint - ograniczenie sąsiedztwa.
Nested predicate - zagnieżdżony predykat.
Net Present Value (NPV) - wartość bieżąca netto.
Non-numerical - logiczny lub symboliczny.
Number - numer, liczba.
Objective function - wskaźnik jakości.
Operation Research - badania operacyjne.
Optimization - optymalizacja.
OS - SO, stan optymalny.
OST - TSO, trajektoria stanów - trajektoria stanów optymalna.
Output of predicate - wyjście predykatu.
Output of program - wyjście programu.
Permutation - permutacja.
Precedence constraint - ograniczenie kolejnościowe.
Predicate - predykat.
Predicate - built-in - predykat standardowy.
Predicate - ground - predykat uziemiony.
Predicate - nested - predykat zagnieżdżony.
Predicate - private - predykat prywatny.
Predicate - satisfied - predykat spełniony.
Predicate - unsatisfied - predykat niespełniony.
Prefix notation - notacja prefiksowa.

Procedural programming - programowanie proceduralne.
Processor - maszyna.
Propagation - propagacja.
Q.E.D. - Quod errat demonstrandum, czego należało dowieść.
Query - zapytanie.
Reification - urzeczowienie.
Recursion - rekurencja.
Recursion - tail - rekurencja ogonowa.
Regrounding a variable - ponowne uziemienie zmiennej.
Relation - relacja.
Resource - resurs, coś niezbędnego dla wykonania operacji.
Resource constraint - ograniczenie resursu.
Rule - reguła.
Satisfied - spełniony, spełniona.
Scheduling - harmonogramowanie.
Search - poszukiwanie.
Search and propagation - poszukiwanie i propagacja.
Search space - przestrzeń poszukiwań.
Sequencing - szeregowanie.
Semantics - semantyka.
Solver - rozwiązywacz.
Spreading a variable value - propagacja wartości zmiennej.
Syntax - syntaks.
State - complete - stan całkowity.
State - contracted - stan częściowy.
State - feasible - stan dopuszczalny.
State - optimal - stan optymalny.
State - optimal trajectory - optymalna trajektoria stanów.
State space - przestrzeń stanów pełna.
State space - contracted - przestrzeń stanów częściowa.
State trajectory, feasible - dopuszczalna trajektoria stanów.
State trajectory, optimal - optymalna trajektoria stanów.
State - unfeasible - stan niedopuszczalny.
String - string.
Structure - struktura, rekord

Struct - struktura, rekord

Success - sukces.

Task - operacja.

Tautology - tautologia.

Term - term.

Timetabling - rozkładowanie.

top. - główne zapytanie programu.

Trashing - błądzenie.

TSP - problem komiwojażera.

Tuple - krotka.

Unification - unifikacja.

Unsatisfied - niespełniony, niespełniona.

Value - wartość.

Value choice heuristic - heurystyka wyboru wartości.

Variable - zmienna.

Variable - anonymous - zmienna anonimowa.

Variable - Boolean - zmienna boolowska.

Variable - combinatorial - zmienna dyskretna.

Variable - continuous - zmienna ciągła.

Variable - decision - zmienna decyzyjna.

Variable - degrounded - zmienna odziemiona.

Variable - degrounding - ziemieni zmiennej.

Variable - discrete - zmienna dyskretna.

Variable - free - zmien na wolna.

Variable - ground - zmienna uziemiona.

Variable - grounding - uziemienie zmiennej.

Variable - regrounded - zmienna uziemiona ponownie.

Variable - regrounding - ponowne uziemienie zmiennej.

Variable choice heuristic - heurystyka wyboru zmiennej.

Bibliografia

[Aggoun-93] Aggoun, A. i N. Baldiceanu, *Extending CHIP in Order to Solve Complex Scheduling and Placement Problems*, Mathl. Comput. Modelling, Vol. 17, No. 7, str. 57-73, 1993.

[Apt-03] Apt, K. R., *Principles of Constraint Programming*, Cambridge University Press, Cambridge 2003.

[Apt-07] Apt, K. R. i M. G. Wallace, *Constraint Logic Programming using* ECL^iPS^e, Cambridge University Press, Cambridge 2007.

[Ahriz-13] Ahriz, H., *Z models for constraint and optimisation problems* http://www.comp.rgu.ac.uk/staff/ha/ZCSP/, 2013.

[Baker-09] Baker, K. R. i D. Trietsvch, *Principles of Sequencing and Scheduling*, John Wiley and Sons, Hoboken, 2009.

[Baldiceanu-94] Baldiceanu, N. and Contejean E., *Introducing Global Constraints in CHIP* Math.Comput.Modelling, Vol. 20, No. 12, pp. 97-123, 1994.

[Baldiceanu-10] Baldiceanu, N., *Global Constraints Catalog* http://www.emn.fr/x-info/sdemasse/gccat/, 2010.

[Baptiste-95] Baptiste, Ph., C. Le Pape i W. Nuijten, *Constraint-Based Optimization and Approximation for Job-Shop Scheduling*, Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95, Montreal, Canada, 1995.

[Baptiste-01] Baptiste, Ph., C. Le Pape i W. Nuijten, *Constraint-Based Scheduling*, Kluwer Academic Publishers, Boston, 2001.

[Bartak-10] Barták, R., *On-Line Guide to Prolog Programming*, http://kti.mff.cuni.cz/ bartak/prolog/, 2010.

[Bartak-10a] Barták, R., *On-Line Guide to Constraint Programming*, http://kti.mff.cuni.cz/ bartak/constraints/, 2010.

[Bellman-61] Bellman, R., *Adaptive Control Processes*, Princeton University Press, Princeton 1961.

[Bizam-75] Bizám, G. i J. Herczeg, *Gry i logika*, Wydawnictwa Naukowo-Techniczne, Warszawa, 1975.

[Brachman-04] Brachman, R. J. i H. J. Levesque, *Knowledge Representation and Reasoning*, Elsevier - Morgan Kaufmann, Amsterdam, 2004.

[Bratko-01] Bratko, I., *Prolog programming for Artificial Intelligence*, Addison Wesley, Harlow, 3rd ed., 2001.

[Carlier-89] Carlier, J. i E. Pinson, *An algorithm for solving the job-shop problem*, Management Science, 35(2), 164-176, 1989.

[Clark-84] Clark K.L. i F.G. McCabe. *Micro-PROLOG: programming in logic*.
Prentice Hall, 1984.
Przekład polski "Micro-Prolog", WNT, Warszawa, 1988.

[CHIP-07] COSYTEC Complex Systems Technologies. *CHIP V5*.
http://www.cosytec.com/.

[Dechter-03] Dechter, R., *Constraint Processing*, Morgan Kaufmann, San Francisco, 2003.

[Domagala-11] Domagała, Ł., *Application of CLP to instruction modulo scheduling for VLIW processors*, J. Skalmierski CS, Gliwice, 2011.

[Dough-10] Dough, E., *Dough Edmund's Puzzle Page*,
http://www.eclipseclp.org/eclipse/examples, 2010.

[Duncan-90] Duncan, T., *Scheduling Problems and Constraint Logic Programmingf: A Simple Example and its Solution*, AIAI-TR-120,AIAI University of Edinburgh, 1990.

[ECLiPSe-12] CISCO, *The ECLiPSe Constraint Programming System*,
http://www.eclipseclp.org/, 2012.

[ECLiPSe Documentation-12] *Dokumentacja dostępna z opcji Help okna głównego ECLiPSe.*
http://www.eclipseclp.org/, 2012.

[Edmund-98] Edmund, D., *Bob's Shish Kebabs*,
Dell Logic Puzzles, June, 1998, p. 24.

[Edmund-10] Edmund, D., *Dough Edmund's Puzzle Page*,
http://www.eclipseclp.org/eclipse/examples, 2010.

[French-82] French, S., *Sequencing and Scheduling: An Introduction to the Mathematics of Job-Shop*, Ellis Horwood, 1982.

[Friedman-Hill-03] Friedman-Hill, E., *Jess in Action. Rule-Based Systems in Java*, Manning Publ. Co., Greewich, 2003.

[From-94] From, A., *Programmation par Contraintes*, Addison-Wesley, Paris, 1994.

[Gołuchowski-07] Gołuchowski, J., *Technologie informatyczne w zarządzaniu wiedzą w organizacji*, Wydawnictwo Akademii Ekonomicznej, Katowice, wyd. 2, 2007.

[Graham-83] Graham, R. L., *Kombinatoryczna teoria szeregowania*, W Steen L. A. (red.) *Matematyka współczesna. Dwanaście esejów*, WNT, Warszawa, 1983, str. 200-229.

[Hansen-03] Hansen, J., *Constraint Programming versus Mathematical Programming*, Orbit, Vol. 3, Feb. 2003.

[Hooker-00] Hooker, J.N., *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, John Wiley and Sons, New York, 2000.

[Hooker-07] Hooker, J.N., *Integrated Methods for Optimization*, Springer, Pittsburgh, 2007.

[Hunt-13] Hunt, W., *The Stable Marriage Problem*, http://www.csee.wvu.edu/ ksmani/courses/fa01/random/lecnotes/lecture5.pdf, 2013.

[Jaffar-94] Jaffar, J. i M. J. Maher, *Constraint Logic Programming: A Survey*, Journal of Logic Programming, 1994, 19-20, str. 503-581.

[Kjellerstrand-13] Kjellerstrand, H., *Constraint Programming Blog*, http://www.hakank.org/ECLiPSe/, 2013.

[Kjellerstrand-13] Kjellerstrand, H., *Constraint Programming Blog*, http://www.hakank.org/ECLiPSe/, 2013.

[Kowalski-89] Kowalski R., *Logika w rozwiązywaniu zadań*, WNT, Warszawa, 1989.

[Legierski-06] Legierski, W., *Automated Timetabling via Constraint Programming*, J. Skalmierski CS, Gliwice, 2006.

[Lines-95] Lines, M.E., *Liczby wokół nas*, Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 1995.

[Luger-98] Luger, G. F. i Stubblefield, W.A., *Artificial Intelligence. Structures and Strategies for Complex Problem Solving*, Addison Wesley Longman, Inc, Harlow, 3rd ed., 1998.

[Mackworth-92] Mackworth, A.K., *Constraint Satisfaction*, W: Shapiro S.C. (red.) *Encyclopedia of Artificial Intelligence* John Wiley and Sons, New York, 1992.

[Marriott-98] Marriott, K. i P. J. Stuckey, *Programming with Constraints: An Introduction*, The MIT Press, Cambridge Mass., 1998.

[Michalewicz-07] Michalewicz, Z. and M. Michalewicz, *Puzzle Based Learning*, Proceedings of the 2007 AaeE Conference, Melbourne, 2007.

[Michalewicz-08] Michalewicz, Z. and M. Michalewicz, *Puzzle-Based Learning: Introduction to Critical Thinking, Mathematics, and Problem Solving*, Hybrid Publishers, Melbourne, 2008.

[Milano-04] Milano, M.,(ed.) *Constrained and Integere Programming. Towards a Unified Methodology*, Kluwer Academic Publishers, Boston, 2004.

[Morgan-08] Morgan, T., *Business Rules and Information Systems.Aligning IT with Business Goals*, Addison-Wesley, Boston, 4th Printing, 2008.

[Mozart/Oz-10] *Mozart/Oz multi-paradigm language*, http://www.mozart-oz.org/home/doc/fdt/node37.html, 2010.

[Muth-63] Muth, J.F. i G. L. Thompson, *Industrial Scheduling*, Prentice-Hall,Inc.,Englewood Cliffs, 1963.

[Niederliński-11a] Niederliński, A., *rmes Rule-and Model-Based Expert Systems*, J. Skalmierski CS, Gliwice, 2011.

[Niederliński-11] Niederliński, A., *A Quick and Gentle Guide to Constraint Logic Programming via* ECL^iPS^e, J. Skalmierski CS, Gliwice, 2011.

[Niederliński-99] Niederliński, A., *Constraint Logic Programming - from Prolog to CHIP*, Proceedings of the Workshop on Constraint Programming for Decision and Control,Editor J. Figwer, Institute of Automation, Silesian University of Technology, Gliwice, 1999, str. 27-34.

[Niederliński-06] Niederliński, A., *Regułowo-modelowe systemy ekspertowe rmse*, WPKJS, Gliwice, 2006.

[Pieprzyca-09] Pieprzyca, W., *Techniki agentowe w środowiskach ECLiPSe i Java wraz z zastosowaniami*, Rozprawa doktorska, Wydział AEiI, Politechnika Śląska, 2009.

[Poole-98] Poole, D., A. Mackworth i R. Goebel, *Computational Intelligence. A Logical Approach*, Oxford University Press, New York, 1998.

[Puget-08] Puget, J-F., *Constraint Programming: A Great AI Success*, W Prade, H. (red), Proceedings of the 13th European Conference on Atrtificial Intelligence ECAI98, pp. 698-705. John Wiley and Sons, 1998.

[Ross-03] Ross, R. G., *Principles of the Business Rule Approach*, Addison-Wesley, Boston, 2003.

[Rossi-06] Rossi, F., P. van Beek i T. Walsh, *Handbook of Constraint Programming*, Elsevier, Amsterdam, 2006.

[Russel-03] Russel, S. i O. Norvig, *Artificial Intelligence. A Modern Approach*, Prentice Hall, II-nd edition, Upper Saddle Roiver, 2003.

[Schimpf-10] Schimpf, J., *Sudoku*, http://www.eclipse-clp.org/eclipse/examples, 2010.

[Schimpf-10a] Schimpf, J., *Liars*, http://www.eclipseclp.org/eclipse/examples, 2010.

[Simonis-10] Simonis, H., *ECLiPSE ELearning Website*, http://www.eclipse-clp.org/eclipse/examples, 2010.

[Steen-83] Steen, L. A. (red.), *Matematyka współczesna. Dwanaście esejów*, WNT, Warszawa, 1983.

[Szczygieł-03] Szczygieł, T., *Angle Packing Problems: Constraint Logic Programming versus Classical Approaches*, JSCS, Gliwice, 2003.

[Szklarczyk-09] Szklarczyk, R., *Zastosowanie programowania w logice z ograniczeniami do problemu marszrutyzacji pojazdów*, Rozprawa doktorska, Wydział AEiI, Politechnika Śląska, 2010.

[Taha-08] Taha, H. A.,Operations Research: An Introduction, *Operations Research: An Introduction*, 8-me wydanie, Prentice Hall, Upper Saddle River, 2008.

[Tatoń-09] Tatoń, T., *Wybrane problemy aukcji i przetargów kombinatorycznych*, Rozprawa doktorska, Wydział AEiI, Politechnika Śląska, 2009.

[Trzaskalik-08] Trzaskalik, T., *Wprowadzenie do badań operacyjnych z komputerem + CD*, PWE, Warszawa, 2008.

[Tsang-95] Tsang, E., *Foundation of Constraint Satisfaction*, Academic Press, London 1995.

[Turban-11] Turban, E., R. Sharda i D. Delen, *Decision Support and Business Intelligence Systems*, 9-te wydanie, Pearson, Boston, 2011.

[van Hentenryck-89] van Hentenryck, P., *Constraint Satisfaction in Logic Pogramming*, The MIT Press, Cambridge Mass., 1989.

[van Hentenryck-99] van Hentenryck, P., *The OPL Optimization Programming Language*, The MIT Press, Cambridge Mass., 1999.

[von Halle-02] von Halle, B., *Business Rules Applied*, John Wiley and Sons, New York, 2002.

[Wagner-75] Wagner, H. M., *POrinciples of operations Resaerch*, Prentice Hall, Englewood Cliffs, 1975.

[Wikipedia-13] *Stable marriage problem*, http://en.wikipedia.org/wiki/`Stable_marriage_problem`, 2013.

[Williams-99] Williams, H. P., *Model Building in Mathematical Programming*, John Wiley, Chichester, 4th ed., 1999.

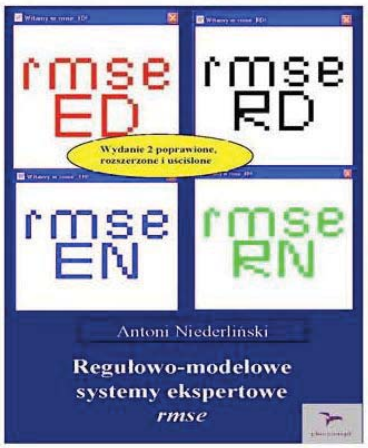
[Winston-94] Winston, W.L., *Operations Research*, Duxbury Press, Belmont, 3rd ed., 1994.

[Wirth-00] Wirth, N., *Algorytmy + struktury danych = programy*, WNT Warszawa, 2000.

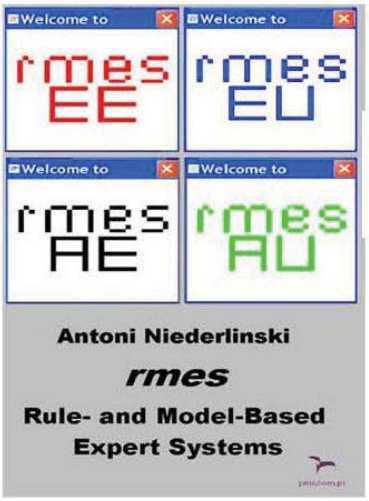
[Wójcik-05] Wójcik, B., *Szeregowanie zadań metodami CLP*, Praca dyplomowa magisterska, Instytut Automatyki, Politechnika Śląska, Gliwice, 2005.

Skorowidz

Jeżeli jeszcze nie kupiłeś, kup koniecznie!



Do nabycia na witrynie:
http://www.rmse.pl



Do nabycia na witrynie:
http://www.rmes.pl

:

Książka jest wprowadzeniem do ważnej techniki modelowania i rozwiązywania problemów decyzyjnych, korzystającej z darmowej platformy ECL^{i}PS^e, dostępnej w ramach *Cisco-style Mozilla Public License*. Technika ta jest szeroko stosowane m. in. dla zadań rozkładowania i harmonogramowania.

9 788360 716953

W książce skorzystano z serii przykładów o wzrastającym stopniu trudności dla łagodnego wprowadzenia w świat pojęć i metod programowania w logice z ograniczeniami, ograniczając treści teoretyczne do niezbędnego minimum. Przykłady mają postać programów dla platformy ECL^{i}PS^e.

Poczynając od prostych kombinatorycznych łamigłówek, poprzez bardziej złożone kombinatoryczne łamigłówki, czytelnik jest wprowadzany w problematykę kombinatorycznej optymalizacji ze szczególnym eksponowaniem harmonogramowania (kulminującego znanym *benchmarkiem* MT10), na optymalizacji ciągłej kończąc.

Prof. zw. dr hab. inż.
Antoni Niederliński
antoni.niederlinski@ae.katowice.pl

PDF tej książki jest nieodpłatnie dostępny na witrynie http://www.pwlzo.pl

Pracuje w Katedrze Inżynierii Wiedzy na Wydziale Informatyki i Komunikacji Uniwersytetu Ekonomicznego w Katowicach. Do 2009 r. pracował w Instytucie Automatyki Politechniki Śląskiej w Gliwicach. Napisał następujące książki: „Układy wielowymiarowe automatyki" (WNT, 1974), „Systemy cyfrowe automatyki przemysłowej" (2-tomy, WNT, 1977, w 1984 r. wydanie czeskie), „Systemy i sterowanie" (PWN, 1984), „Systemy komputerowe automatyki przemysłowej" (2-tomy, WNT, 1984, 1985), „Mikroprocesory-mikrokomputery-mikrosystemy" (4 wydania, WSP, 1978-1987), „Regułowe systemy ekspertowe" (PKJS, 2000), „Regułowo-modelowe systemy ekspertowe *rmse*" (PKJS, 2006), „*rmes* – Rule- and Model Based Expert Systems" (PKJS, 1 wyd. 2008, 2 wyd. 2011), „Programowanie w logice z ograniczeniami" (PKJS, 1 wyd. 2010),"A Quick and Gentle Guide to Constraint Logic Programming *via* ECL^{i}PS^e", (PKJS, 2011). Przedmiotem jego zainteresowań naukowych i pracy dydaktycznej są systemy ekspertowe i zastosowania programowania w logice z ograniczeniami.