Nagyhatékonyságú logikai programozás

(Constraint Logic Programming)

Jegyzetek a BME informatikus hallgatói számára

Kézirat

Szeredi Péter
IQSOFT Rt.
szeredi@iqsoft.hu

A jegyzetet az előadásvázlatok alapján elkészítette:
Rozmán Tamás

Budapest 1997

Tartalom

1 Constraint Logic Programming (CLP)

A CLP (Constraint Logic Programming) séma a logikai programozás egy új irányzata, tekinthető egy kiterjesztésének is. Alapvető tulajdonsága, hogy a program tartalmazhat változók értékeire való megszorításokat, angolul constrainteket. Az alábbi táblázatban összefoglaljuk a legelterjedtebb logikai programozási nyelv, a Prolog alapelemit és ezek CLP megfelelőit.

Prolog	CLP
hívás	hívás vagy constraint
egyesítés	constraint solving
válasz - behelyettesítés	válasz - constraint

1.1 A CLP nyelv elemei

Minden CLP nyelvben szerepelnek a következő elemek:

- egy D tartomány (domain), például
  - egészek (N), valósak (R), racionálisak(Q), racionális intervallumok
  - véges alaphalmazok (finite domain, fd)
  - Boole értékek (B)
  - listák, füzérek (string-ek)
- D-ben definiált függvényeknek egy F halmaza, például
  - $+$, $-$, $*$, $/$
  - $\vee$, $\wedge$
- D-ben definiált relációknak (constrainteknek) egy R halmaza, például
  - $=$, $\neq$, $<$, $>$, $\geq$, $\leq$
  - $\in$ reláció
- egy constraint-megoldó algoritmus, constraint-solver $< D, F, R >$ -re, azaz a D tartományban az $R + F$ -beli jelekből felépített constraintekre, például
  - Gauss elimináció
  - szimplex módszer
  - constraint satisfaction (CSP) algoritmusok

A függvényeket és a relációkat CLP-ben - a Prolog-tól eltérően - nem szintaktikusan, hanem szemantikusan kezeljük, a Prologban a program dolga, hogy jelentést tulajdonítson egy függvény (struktúra) - kifejezésnek (pl. az $is/2$ predikátum). A Prolog csak a szintaktikus egyesítést (a = /2 műveletet) ismeri, ami például az $X + 2 = Y + 3$ kifejezést sikertelenül próbálja feldolgozni, így a hívás meghiúsul. Ha ezt a kifejezést egy alaphalmaz, domain felett nézzük, pl. a valós számok fölött (CLP(R)), akkor a matematikailag helyes megoldást kapjuk, azt hogy $X = Y + 1$. Ahhoz, hogy ezt az eredményt kaphassuk, szükség van a constraint solver-re, ami a változókból, értékekből, függvényekből és a relációkból álló constrainteket ki tudja értékelni, és ellenőrizni tudja a konzisztenciájukat. Ha egy új constraint hozzávételével a tár inkonzisztenssé válna, azt a constraint solver észreveszi, és a programnak az aktuális végrehajtási ága meghiúsul. Tulajdonképpen a Prolog-ot is felfoghatjuk egy olyan CLP nyelvnek, aminek a D halmaza a Herbrand-fák halmaza (a Prolog fastruktúrája), nincsen függvénye és csak az egyesítés (= /2) relációt ismeri.

1.2 Példák CLP nyelvekre

- CLP(fd) - egészek részhalmazán
  - $D = \{\ -m, -m+1, \ldots, -1, 0, 1, \ldots, n-1, n\ \}$
  - $F = \{\ +\ ,\ -\ ,\ *\ ,\ /\ ,\ mod\ \}$
  - $R = \{\ =\ ,\ \neq\ ,\ \leq\ , \in\ \}$
  - módszerek: constraint satisfaction technikák (konzisztencia-módszerek, constraint-propagálás)
- CLP(B) - CLP(fd) speciális esete, de erősebb bizonyítási módszerek vannak rá
  - $R = \{\ 0\ ,\ 1\ \}$
  - $F = \{\ \wedge\ , \vee\ ,\ not\ ,\ xor\ \}$
  - $R = \{\ =\ \}$
  - módszerek : ítéletkalkulusbeli tételbizonyítási módszerek
- CLP(Q) - racionálisokon
  - $D = Q$
  - $F = \{\ +\ ,\ -\ ,\ *\ , /\ \}$
  - $R = \{\ =\ , \neq\ ,\ \leq\ ,\ \ldots\ \}$
  - módszerek : Lineáris programozás: Gauss elimináció, szimplex módszer
- CLP(R) - ugyanaz, mint CLP(Q), de valósak lebegőpontos ábrázolású közelítéseivel.

1.3 CLP szintaxis és deklaratív szemantika

Szintaxis:

program:	klózok halmaza
klóz:	$P : -G_1, \ldots, G_n$, ahol G_i vagy cél vagy constraint
deklaratív olvasata:	P igaz, ha $G_1, \ldots, G_n$ mind igaz
kérdés:	$? - G_1, \ldots, G_n$
válasz egy kérdésre:	constrainteknek egy olyan konjunkciója, amelyből a kérdés következik

1.4 CLP procedurális szemantika

A CLP séma szemantikája leírható, mint egy kezdeti célból történő levezetés, ami felhasználja a program klózait.

A levezetés egy végrehajtási állapota: $< G, s >$, ahol

- $G =$ a megoldandó célok és constraintek konjunkciója
- $s =$ constraint-tár: az eddig felhalmozott constraintek konjunkciója. A constraint tár mindig kielégíthető !

Sok esetben a constraint tár a nyelvben megengedett constrainteknek csak bizonyos fajtáit tárolhatja, ezek az úgynevezett egyszerű constraintek (jelölés: C általános, c egyszerű constraint). Például a CLP(fd) egyszerű constraintje az $\in$ reláció.

Egy végrehajtási alaplépés:

1. egy részcél, P determinisztikus kiválasztása
2. egy klóz, $P' : -G_1, \ldots, G_n$ nemdeterminisztikus kiválasztása

3. ellenőrzés, hogy $P = P'$ konzisztens-e a constraint-tárral

A végrehajtás véget ér, ha üres a célsorozat.

A végrehajtás eredménye a constraint tár, vagy annak a kérdésbeli változókra való vetítése.

Általános következtetési lépések:

- Rezolúció:
$$< P \,\&\, G, s > \Longrightarrow < P = P' \,\&\, G_1 \,\&\, \ldots \,\&\, G_n \,\&\, G, s >$$
ha a programban van egy $P' : -G_1, \ldots, G_n$ klóz
- constraint-megoldás:
$$< c \,\&\, G, s > \Longrightarrow < G, s \wedge c >$$
ha c konzisztens s-sel
- constraint-erősítés:
$$< C \,\&\, G, s > \Longrightarrow < C' \,\&\, G, s \wedge c >$$
ha c konzisztens s-sel és ha s-ből következik, hogy C ekvivalens $C' \wedge c$ -sel.

A constraint erősítésre akkor lehet szükség, ha más típusú constraintek vannak a programban és a constraint-tárban. Például CLP(fd) esetén a tárban csak $\in$ típusú constraintek szerepelhetnek - így pl. egy $X > Y$ relációt nem mindig lehet közvetlenül bevinni a tárba. Ilyenkor alkalmazzák ezt a technikát.

A constraint-erősítésnek vannak speciális esetei:

1. $C = C'$, azaz a célsorozatban megmarad a constraint. Ekkor a c-re vonatkozó feltétel az, hogy $s \Rightarrow (C \Rightarrow c)$, azaz az s tárból és a C constraintből megpróbálunk egy tárba rakható egyszerű constraintet kikövetkeztetni.
2. $C' = true$, azaz teljesen eltűnik a constraint. Ez akkor lehetséges, ha a tárból (vagy annak egyszerű constraintekkel való bővítéséből) következik C. Ez a constraint megoldással analóg.

Lássunk az erősítésre egy példát:

1. A constraint tár tartalma kezdetben
$$X \in \{1..10\}, Y \in \{4..11\}$$
2. hozzávesszük az $C_1 = \{X > Y\}$ constraint-et:
$$X \in \{5..10\}, Y \in \{4..9\}, C_1' = \{X > Y\}$$
Látható, hogy a constraint-tár tartalma nem garantálja az $X > Y$ teljesülését, így ezt a feltételt a továbbiakban is figyelembe kell venni (C_1').
3. felvéve a $C_2 = \{X + Y = 10\}$ constraint-et:
$$X \in \{5..6\}, Y \in \{4..5\}, C_2' = \{X + Y = 10\}$$
4. feléled a C_1' constraint, de nem tud mit csinálni, és nem is következik a tárból: $C_1'' = C_1'$
5. felvéve a $C_3 = \{X \neq 5\}$ constraint-et:
$$X \in \{6\}, Y \in \{4, 5\}, C_3' = \{\}$$
6. feléled a C_1'' és a C_2' constraint, és teljesül a tárból:
$$X \in \{6\}, Y \in \{4, 5\}, C_1''' = \{\}$$
7. feléled a C_2' constraint:
$$X \in \{6\}, Y \in \{4\}, C_2'' = \{\}$$

1.5 Egy CLP nyelv megvalósítása

Egy CLP nyelvhez szükséges definiálni a következőket:

- egyszerű constraintek, amelyekből a constraint-tár épül fel
- constraint-termek, azok a kifejezések, amelyek constraintekben előfordulhatnak
- (általános) constraintek és feldolgozásuk
  - egyszerű constraintek ⟶ constraint-megoldás
  - nem-egyszerű constraintek ⟶ constraint-erősítés (korutinok)
- a szükséges algoritmusok:
  - constraint-megoldás (solving)
  - constraint-ellenőrzés (entailment)
  - constraint-erősítés (strengthening)
  - constraint-tár vetítése (projection)

1.6 CLP megoldó algoritmusok

Az algoritmusok alapművelete: egy constraint hozzáadása a constraint-tárhoz, azaz

$$s' = s \wedge c$$

felépítése, beleértve $s \wedge c$ konzisztenciájának ellenőrzését is.

Követelmények az algoritmussal szemben:

- inkrementálisság (s konzisztenciáját ne bizonyítsa újra)
- a visszalépés támogatása
- hatékonyság
- teljesség, ami itt azt jelenti, hogy az algoritmus minden esetben el tudja dönteni, hogy egy constraint-rendszer kielégíthető vagy nem

2 CLP segédeszközök SICStusban

Ebben a fejezetben nem szorosan a CLP-vel fogunk foglalkozni. A fejezet fő célja megismertetni azzal, hogy egy logikai programozási nyelv egy konkrét példánya (a SICStus Prolog) milyen eszközöket ad egy rá épülő CLP nyelv megvalósításához.

2.1 Korutinszervezés

A korutin egy olyan Prolog rutin, amelynek végrehajtása felfüggeszthető, amíg egy bizonyos feltétel igaz nem lesz. Amint a feltétel igazzá válik, a korutin újraaktiválódik. A feltétel általaban az, hogy egy bizonyos változó be van-e helyettesítve, de lehet más is. Korutinszervezéssel sok programot hatékonyabban és áttekinthetőbben lehet megírni, így érdemes használni.

2.1.1 Blokk-deklarációk

Ha egy hívást csak akkor akarunk végrehajtani, amikor már bizonyos változók be vannak helyettesítve, akkor lehet a blokk-deklarációt használni. Példa:

```
:- block p(-, ?, -, ?, ?).
```

Jelentése: ha az első és a harmadik argumentum is behelyettesítetlen változó (ez a blokkolási feltétel), akkor a `p` hívás felfüggesztődik. A hívás csak azután folytatódik, ha a blokkolási feltétel megszűnik, azaz a példában az első vagy a harmadik argumentum nem-változó behelyettesítést kap. Egy futás végén, ha maradt végre nem hajtott felfüggesztett hívás, akkor azt a rendszer kiírja.
Ugyanarra az eljárásra több feltétel is szerepelhet, ezek vagylagosan értendők, például a

```
:- block p(-, ?), p(?, -).
```

deklaráció esetén ha vagy az első, vagy a második argumentum változó, akkor a hívás blokkolódik.

2.1.2 Végtelen választási pontok kiküszöbölése blokk-deklarációval

```
:- block pick(-, ?, -).
pick([X|L], X, L).
pick([Y|L], X, [Y|L1]):-
      pick(L, X, L1).
```

```
perm([], []).
perm(L, [X|P]):-
      pick(L, X, L1), perm(L1, P).
```

A `perm` eljárás a egy adott lista permutációját állítja elő. Normál esetben (blokkolás nélkül) az első paramétere a bemenő, a második a kimenő. A matematikai érzék azt sugallja, hogy logikus lenne, ha bármelyik argumentum lehetne a bemenő (a permutáció kölcsönös). A `perm` eljárás blokk-deklarációval kiegészítve visszafelé is működik:

```
| ?- perm(L, [1,2]).
   1  1  Call: perm(_69,[1,2]) ?
   -  -  Block: pick(_363,1,_368)
   2  2  Call: perm(_368,[2]) ?
   -  -  Block: pick(_742,2,_747)
   3  3  Call: perm(_747,[]) ?
   -  -  Unblock: pick(_742,2,[])
   4  4  Call: pick(_742,2,[]) ?
   -  -  Unblock: pick(_363,1,[2])
   5  5  Call: pick(_363,1,[2]) ?
   5  5  Exit: pick([1,2],1,[2]) ?
   4  4  Exit: pick([2],2,[]) ?
   3  3  Exit: perm([],[]) ?
   2  2  Exit: perm([2],[2]) ?
   1  1  Exit: perm([1,2],[1,2]) ?
L = [1,2] ?
```

2.1.3 Blokkolás alkalmazása: Generál-és-ellenőriz típusú programok gyorsítása

Generál-és-ellenőriz típusú programok valamilyen módszerrel generálják a lehetséges megoldásokat, és ezután ellenőrzik, hogy az aktuálisan vizsgált lehetőség jó-e. Ezek a programok általában nem hatékonyak, mert túl sok visszalépést használnak. Korutinszervezéssel a generáló és ellenőrző rész "automatikusan" összefésülhető, így a végrehajtás sokkal hatékonyabbá tehető. Ehhez az ellenőrző részt előre kell tenni és megfelelően blokkolni. Az alábbi példa egy buta rendező algoritmust javít fel viszonylag elfogadható sebességűre. A rendezés alapja: generáljuk a rendezendő lista összes permutációját, majd ellenőrizzük, hogy rendezett-e.

```
% az egyszerű generál-és-ellenőriz típusú programhoz
% a sorted és a perm felcserélendő
sort(L, S) :- sorted(L), perm(L, S).

sorted([]).
sorted([_]).
sorted([X,Y|L]):- sorted(L, X, Y).

:- block sorted(?, -, ?), sorted(?, ?, -).
sorted([], X, Y) :- X =< Y.
sorted([Z|L], X, Y) :- X =< Y, sorted(L, Y, Z).
```

Futási idők

Listahossz	7	8	9
gen-test	0.48s	3.86s	35.64s
korutinos	0.04s	0.09s	0.18s
gen-test+korutin	0.55s	4.37s	40.71

Megjegyzés: a táblázat utolsó sora azt jelzi, hogy a blokkolásért árat kell fizetni: ha feleslegesen alkalmazzuk, lelassíthatja a programot.

2.1.4 Korutinszervezés: alternatív lehetőségek

`freeze(X, Hívás)`
`Hívás`-t felfüggeszti mindaddig, amíg `X` behelyettesítetlen változó. Definíciója lehetne:

```
:- block freeze(-, ?).
freeze(_, Goal) :- call(Goal).
```

Ha lehet, váltsuk ki blokk-deklarációval, mivel a SICStus-ban jelenleg a `call/1` hívás meglehetősen nagy overhead-del rendelkezik.

`frozen(X, Hívás)`
Az `X` változó miatt felfüggesztett hívás(oka)t egyesíti `Hívás`-sal.

`when(Feltétel, Hívás)`
`Hívás`-t mindaddig blokkolja, amíg `Feltétel` nem teljesül. `Feltétel` megengedett szintaxisa:

Felt `::= nonvar(X) | ground(X) | ?=(X, Y) |`
Felt, *Felt* | *Felt*; *Felt*

- `nonvar(X)`: `X` nem változó (így a `when` ekvivalens a `freeze`-el)
- `ground(X)`: `X` teljesen behelyettesített, vagyis nincs benne változó
- `?=(X,Y)`: `X`, `Y` egyesíthetősége eldönthető, vagyis vagy X == Y, vagy X és Y biztosan nem lehet már egyenlő
- *Felt*, *Felt* : és feltétel
- *Felt*; *Felt* : vagy feltétel

`dif(X, Y)`
`X` és `Y` nem egyesíthető. Mindaddig felfüggesztve marad, amíg ez el nem dönthető. Definíciója lehetne:

```
dif(X, Y) :- when(?=(X,Y), X \== Y).
```


`call_residue(Hívás, Maradék)`
`Hívás`-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja `Maradék`-ban.

```
| ?- call_residue(dif(X, f(Y)), Maradek).
  Maradek = [[X]-(prolog:dif(X,f(Y)))] ?
  yes
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradek).
  X = f(Z),
  Maradek = [[Y,Z]-(prolog:dif(f(Z),f(Y)))] ?
```

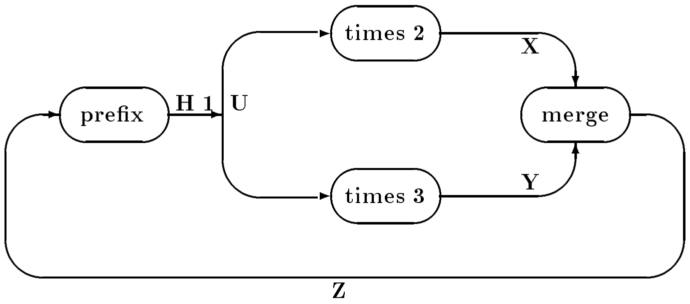
2.1.5 Dataflow programozás korutinszervezéssel

A dataflow programozás nem kapcsolódik a CLP témaköréhez. Ennek ellenére érdemes egy kis kitérőt tenni, mivel az alábbi példa alapján könnyebb megérteni a korutinszervezés logikáját.

Példa: egyszerűsített Hamming feladat

A feladat: keressük a $2^i * 3^j (i \geq 1, j \geq 1)$ alakú számok közül az első `N` darabot nagyság szerint rendezve.
A megoldás: a “stream-and-parallelism” közelítésmódot használva korutinszervezéssel oldjuk meg. Az egyes predikátumok adatfeldolgozó egységeknek felelnek meg, az argumentumaikban szereplő változók (Prolog listák) pedig az ezeket összekapcsoló adatfolyamoknak.

Az adatáramlási hálózat:




```
% A H lista az első N, csak a 2 és 3 tényezőkből álló szám.
hamming(N, H) :-
  U = [1|H], times(U, 2, X), times(U, 3, Y), merge(X, Y, Z),
  prefix(N, Z, H).

% times(X, M, Z): A Z lista az X elemeinek M-szerese
:- block times(-, ?, ?).
times([A|X], M, Z) :-
  B is M*A, Z = [B|U], times(X, M, U).
times([], _, []).

% merge(X, Y, Z): Z az X és Y összefésülése.
:- block merge(-, ?, ?), merge(?, -, ?).
% Csak akkor fusson, ha az első két argumentum ismert
merge([A|X], [B|Y], V) :-
  A < B, !, V = [A|Z], merge(X, [B|Y], Z).
merge([A|X], [B|Y], V) :-
  B < A, !, V = [B|Z], merge([A|X], Y, Z).
```

```
merge([A|X], [A|Y], [A|Z]):-
  merge(X, Y, Z).
merge([], X, X) :- !.
merge(_, [], []).

% prefix(N, X, Y): Az X lista első N eleme Y.
prefix(0, _, []) :- !.
prefix(N, [A|X], [A|Y]) :-
  N > 0, N1 is N-1, prefix(N1, X, Y).
```

2.2 További CLP segédeszközök SICStusban

- **Tetszőleges nagyságú egész számok**

  pl:

  ```
  | ?- fakt(100,F).

  F = 93326215443944152681699238856266700490715968264381
  62146859296389521759999322991560894146397615651828625З
  69792082722375825118521091686400000000000000000000000 ?
  ```

- **Visszaléptethető módon változtatható kifejezések (mutábilisek)**

  Ezek tulajdonképpen a gépközelibb programozási nyelvek pointer fogalmát hozzák be a Prolog világba. Ha például építünk egy Prolog fastruktúrát, aminek két (vagy több) részfájában ugyanarra a változtatható kifejezésre van szükségünk, akkor a mutábilisek használatával ha az egyik helyen megváltoztatjuk a kifejezést, akkor a másik helyen is megváltozik. Mutábilisek használata nélkül ezt ezt nehéz és nem is hatékony megírni, főleg ha a visszalépést is figyelembe kell venni.

  `create_mutable(Adat, ValtKif)`

  `Adat` kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz `ValtKif`. `Adat` nem lehet üres változó.

  `get_mutable(Adat, ValtKif)`

  `Adat`-ba elővezi `ValtKif` pillanatnyi értékét.

  `update_mutable(Adat, ValtKif)`

  A `ValtKif` változtatható kifejezés új értéke `Adat` lesz. Ez a változtatás visszalépéskor visszacsinálódik. `Adat` nem lehet üres változó.

- **Mellékhatás visszacsinálása visszalépéskor**

  `undo(Hivas)`

  Feltétel és mellékhatás nélkül sikerül. Visszalépéskor végrehajtódik a `call(Hivas)` hívás, és a visszalépés folytatódik.

  Példa:

  ```
  assert_b(Cl) :- assert(Cl), undo(retract(Cl)).
  ```

  A vágók nem bántják, nem úgy mint:

  ```
  rosszabb_assert_b(Cl) :-
        ( assert(Cl)
        ; retract(Cl), fail
        ).
  ```


2.3 Egy egyszerű CLP feladat: térképszinezés

Ebben a fejezetben egy konkrét feladatra, a jól ismert gráf színezésre adunk megoldásokat SICStus Prolog-ban. (A feladat: adottak egy gráf pontjai és élei, és meg kell adni a gráf pontjainak egy olyan színezését, hogy minden él végpontjainak színe különböző legyen.) A megoldások az egyszerű visszalépéses kereséstől egészen a `clpfd` könyvtár használatáig terjednek, ami már egy konkrét CLP nyelv. A cél az, hogy a példákon keresztül jobban megértsük a CLP(fd) nyelv működését. A fejezet végén összehasonlítjuk az egyes megoldások hatékonyságát.

2.3.1 Tiszta Prolog megoldás: visszalépéses keresés

Ezen a megoldáson sok magyaráznivaló nincs: egyszerű Prolog visszalépéses keresést valósít meg.

```
terkep_pl(Szk, Ok) :- szinez(Szk), write(Ok), nl.

% szinez(Szk): Az Szk szomszédságok 3 színnel színezhetök
szinez([]).
szinez([P1-P2|Szk]) :- ne(P1, P2), szinez(Szk).

% ne(P1, P2): P1 és P2 különbözö színek.
ne(1,2). ne(1,3). ne(2,1). ne(2,3). ne(3,1). ne(3,2).
```

2.3.2 Korutinos megoldás: `dif/2` segítségével

Ez a megoldás a `dif/2` predikátumot használja. A `dif` korutinokkal operál, látható is, hogy előbb vesszük fel a feltételeket (`dif_feltetelek`, ez a predikátum írja le, hogy minden él végpontjai különbözőek), és csak ezután kezdjük el generálni a pontok színeit.

```
terkep_dif(Szk, Ok) :-
  dif_feltetelek(Szk), ertekek(Ok),
  write(Ok), nl.

dif_feltetelek([]).
dif_feltetelek([P1-P2|Szk]) :- dif(P1, P2), dif_feltetelek(Szk).

ertekek([]).
ertekek([V|Vk]) :-
  szin(V), ertekek(Vk).

szin(1). szin(2). szin(3).
```

2.3.3 CLP(fd) jellegű megoldás: minden változóhoz értékkészlet hozzárendelése

A megoldás már a CLP(fd) közelítésmódot használja, vagyis minden egyes, a feltételekben szereplő változóhoz egy halmazt rendel, amiből az az értékeit felveheti. A halmazok rendezett egész számokat tartalmazó listák. Ezeket a halmazokat egy listában tároljuk, amelynek elemei a változó - értékhalmaz párok. Ha egy predikátumnak szüksége van egy változóhoz tartozó halmazra, akkor azt kikeresi a listából, ha módosítani (szűkíteni) akar, akkor a megváltoztatott listát visszaadja. A program első néhány predikátuma a az fd változók kezelésével foglalkozik (létrehozás, elem törlése az értékhalmazból, ...).

A megoldás kezdetben minden változóhoz hozzárendeli mindhárom lehetséges színt mint értékkészletet. Ezután fogja a sorban (a listában) az első változót, és behelyettesíti, vagyis az értékkészletéből egy elemet kivesz, és hozzárendeli a változóhoz mint értéket. Ezt az értéket azután kiveszi minden, vele éllel összekötött csúcs értékkészletéből, majd az egészet folytatja a sorban következő változóval. Az egész addig megy visszalépésekkel együtt, amíg az összes változót be nem helyettesíti.

```
% erteke(CV, E): CV-nek konkrét értéke van, E.
erteke(E,E):- nonvar(E).

% kihagy(CV, E, Store0, Store)
% CV tartományából kihagyja E-t
kihagy(CV, E, Store0, [CV=UjEL|Store]):-
  select_eq(CV=EL, Store0, Store), delete(EL, E, UjEL),
  ( UjEL = [VE] -> CV = VE
  ; UjEL \== []
  ).

% a programban több predikátum utolsó két paramétereként szereplő
% Store0, Store jelentése:
% Store0 a kezdeti (bemenő), Store a végső (kimenő) constraint-tár
terkep_fdst(Szk, Ok) :-
  valtozok(0, Ok, Vk, Store0), megold(Szk, Ok, Store0, _Store1),
  write(Vk), nl.

% valtozok(N,CVk, Vk, Store):
% A CVk constraint-változók megfelelöi a Vk változók,
% N az változó ID-je, Store a constraint-tár
valtozok(_, [], [], []).
valtozok(N0, [V|NVk], [V|Vk], [V=[1,2,3]|CVk]) :-
  N is N0+1, valtozok(N, NVk, Vk, CVk).

% valaszt(CV,Store0,Store): CV-t a lehetséges értékeivel helyettesíti.
valaszt(V, Store0, [V=[V]|Store]):-
  select(V=EL, Store0, Store), member(V, EL).

% valaszt(CVk0, CVk, Store0, Store) :- értéket ad a CVk0-beli elsö értékkel
% nem bíró változónak ha van. CVk a fennmaradó változók listája.
valaszt([CV|CVk0], CVk, Store0, Store):-
  erteke(CV,_), !, valaszt(CVk0, CVk, Store0, Store).
valaszt([CV|CVk0], CVk0, Store0, Store):- valaszt(CV, Store0, Store).
valaszt([],[], Store, Store).

% A Ck constraint-ek a CVk változókkal megoldhatók
megold([], [], Store, Store).
megold(Ck, CVk, Store0, Store) :-
  CVk \== [], valaszt(CVk, CVk1, Store0, Store1),
  szur(Ck, Ck1, Store1, Store2),
  megold(Ck1, CVk1, Store2, Store).

% A Ck0 constraint-ek közul kihagyva a teljesülteket
% kapjuk a Ck constraint-listát, Ck tovább nem szürhetö.
szur(Ck0, Ck, Store0, Store) :-
  szur1(Ck0, Ck1, Store0, Store1),
  (   Ck1 = Ck0 -> Ck = Ck0, Store = Store1
  ;   szur(Ck1, Ck, Store1, Store)
  ).

% szur1(Ck0, Ck): Ck0-ból kihagyva a teljesülteket kapjuk Ck-t.
szur1([], [], Store, Store).
szur1([CV1-CV2|Ck0], Ck, Store0, Store) :-
  (   erteke(CV1,E) ->
      kihagy(CV2, E, Store0, Store1),  szur1(Ck0, Ck, Store1, Store)
  ;   erteke(CV2,E) ->
```



```
      kihagy(CV1, E, Store0, Store1),  szur1(Ck0, Ck, Store1, Store)
  ;   Ck = [CV1-CV2|Ck1], szur1(Ck0, Ck1, Store0, Store)
  ).

% mint a list konyvtar select/3 eljarasa
select_eq(V=EL, [V0=EL|Store], Store) :-
        V == V0.
select_eq(CV, [CV0|Store0], [CV0|Store1]) :-
        select_eq(CV, Store0, Store1).
```

2.3.4 FD változók megvalósítása mutábilisekkel

Ezeket a predikátumokat a következőkben megadott programok fogják használni. Ezek feladata, hogy egy változóhoz egy véges értékhalmazt rendeljenek (finate domain, fd), amelyből a változó az értékeit felveheti. A predikátumok megfelelői az előző programban szereplő azonos nevű, de más argumentumszámú eljárásoknak, néhány újjal kiegészítve. A cél az, hogy az előző programot, ami a constraint változókat listában tárolta, átalakítsuk úgy, hogy a constraint változókat mutábilis kifejezésekként tárolja. Ezzel a program gyorsabb lesz, és mentesülünk a constraint-tár állandó karbantartásától, amit az előző programban a minden eljárásnak átadott két plusz argumentummal: a `Store0` bemenő tár-állapottal, és a `Store` kimenő tár-állapottal tudtunk csak megvalósítani.

```
:- use_module(library(lists)).

% uj_valtozo(EL, CV, V): EL érték-készlettel létrehoz egy új
% CV-változót (CV), az ennek megfelelő Prolog változó a V.
uj_valtozo(E, V=ME, V) :- create_mutable(E, ME).

% erteke(CV, E): CV-nek konkrét értéke van, E.
erteke(V=_Ert, V) :- nonvar(V).

% valtozoja(CV, V): V a CV-hez tartozó változó.
valtozoja(V=_, V).

% kihagy(CV, E): CV tartományából kihagyja E-t.
kihagy(V=Ert, E) :-
  get_mutable(EL, Ert), delete(EL, E, UjEL),
  update_mutable(UjEL, Ert),
  (   UjEL = [VE] -> V = VE
  ;   UjEL \== []           ).

% kihagyb(CV, E): CV tartományából kihagyja E-t.
% Pontosan ugyanaz, mint kihagy, csak blokk-deklarációval.
:-block kihagyb(?, -).
kihagyb(V=Ert, E) :-
  get_mutable(EL, Ert), delete(EL, E, UjEL),
  update_mutable(UjEL, Ert),
  (   UjEL = [VE] -> V = VE
  ;   UjEL \== []           ).
```

2.3.5 CLP(fd) megoldás: mutábilis kifejezések használata

Ez a 2.3.3. fejezetben található program mutábilisekkel megvalósított változata. Ha jól megnézzük, akkor láthatjuk, hogy a 2.3.3. pontban megadott megoldástól csak annyiban tér el, hogy itt nincs szükség a constraint-tár állapotát minden eljárásnak átadni, így minden eljárás két paraméterrel kevesebbet kap; valamint abban, hogy az előző, 2.3.4. pont predikátumait használja constraint-változók megvalósítására.

```
:- use_module(library(lists)).
```

```
terkep_fd(Szk, Ok) :-
  valtozok(Ok, Vk), megold(Szk, Ok),
  write(Vk), nl.

% valtozok(CVk, Vk):
% A CVk constraint-változók megfelelöi a Vk változók
valtozok([], []).
valtozok([CV|CVk], [V|Vk]) :-
  uj_valtozo([1,2,3], CV, V), valtozok(CVk, Vk).

% valaszt(CV): CV-t a lehetséges értékeivel helyettesíti.
valaszt(V=Ert) :-
  get_mutable(EL, Ert), member(V, EL), update_mutable([V], Ert).

% valaszt(CVk0, CVk) :- értéket ad a CVk0-beli elsö értékkel
% nem bíró változónak ha van. CVk a fennmaradó változók listája.
valaszt([CV|CVk0], CVk) :-
  erteke(CV, _), !, valaszt(CVk0, CVk).
valaszt([CV|CVk0], CVk0) :-
  valaszt(CV).
valaszt([], []).

% A Ck constraint-ek a CVk változókkal megoldhatók
megold([], []).
megold(Ck, CVk) :-
  CVk \== [], valaszt(CVk, CVk1), szur(Ck, Ck1), megold(Ck1, CVk1).

% A Ck0 constraint-ek közul kihagyva a teljesülteket
% kapjuk a Ck constraint-listát, Ck tovább nem szürhetö.
szur(Ck0, Ck) :-
  szur1(Ck0, Ck1),
  (   Ck1 = Ck0 -> Ck = Ck0
  ;   szur(Ck1, Ck)
  ).

% szur1(Ck0, Ck): Ck0-ból kihagyva a teljesülteket kapjuk Ck-t.
szur1([], []).
szur1([CV1-CV2|Ck0], Ck) :-
  (   erteke(CV1, E) -> kihagy(CV2, E),  szur1(Ck0, Ck)
  ;   erteke(CV2, E) -> kihagy(CV1, E),  szur1(Ck0, Ck)
  ;   Ck = [CV1-CV2|Ck1], szur1(Ck0, Ck1)
  ).
```

2.3.6 CLP(fd) jellegű megoldás: `freeze` használata

Az előző megoldás korutinos változata. Itt előbb felvesszük a feltételeket, blokkoljuk őket majd csak ezután kezdjük a pontok színeit behelyettesíteni. Ezzel elérjük, hogy a feltételek minden további adminisztráció nélkül akkor ébredjenek fel, amikor szükség van rájuk. A feltételek ugyanazok mint eddig: minden élre, ha az egyik végpontja értéket kap, azt a másik végpont értékkészletéből törölni kell.

```
terkep_fdfreeze(Szk, Ok) :-
  valtozok(Ok, Vk),
  feltetelek(Szk),
  behelyettesit(Ok),
  write(Vk), nl.
```



```
% feltetelek(Ck): A Ck constraint-eket felveszi.
feltetelek([]).
feltetelek([CV1-CV2|Ck]) :-
  feltetel(CV1, CV2), feltetel(CV2, CV1),
  feltetelek(Ck).

% feltetel(CV1, CV2): ha CV1 és CV2 értéke különbözö.
feltetel(CV1, CV2) :-
  valtozoja(CV1, V1), kihagyb(CV2, V1).

% behelyettesit(CVk): CVk-et a lehetséges értékeikkel
% helyettesíti.
behelyettesit([]).
behelyettesit(CVk0) :-
  valaszt(CVk0, CVk), behelyettesit(CVk).
```

2.3.7 CLP(fd) jellegű megoldás: *first-fail* elv

Az előző megoldásban a változókat eredeti sorrendjükben helyettesítettük be. Most a sorrendet nem a "véletlenre" bízzuk, hanem az ún. first-fail heurisztikát alkalmazzuk, vagyis először a legkisebb értékkészletű változót helyettesítjük be. A cél az, hogy az első néhány változó behelyettesítésénél nagyobb valószínűséggel válasszunk jó értékeket, és így csökkentsük a visszalépések számát.

```
terkep_fdfreezeff(Szk, Ok) :-
  valtozok(Ok, Vk), feltetelek(Szk),
  ff_behelyettesit(Ok),             % csak ez változott
  write(Vk), nl.

% ff_behelyettesit(CVk): az összes CVk-t lehetséges
% értékével helyettesíti.
ff_behelyettesit([]).
ff_behelyettesit(CVk0) :-
  ff_valaszt(CVk0, CVk),
  ff_behelyettesit(CVk).

% Értéket ad a CVk0-ban levö legszükebb tartományú változónak.
ff_valaszt(CVk0, CVk) :-
  hosszak(CVk0, H_CVk), H_CVk \== [], !,
  keysort(H_CVk, [_-CV|R_CVk]),
  hossztalan(R_CVk, CVk), valaszt(CV).
ff_valaszt(_, []).

hosszak([], []).
hosszak([CV|CVk], [H-CV|H_CVk]) :-
  CV=(_=Ert), get_mutable(E, Ert), length(E, H),
  hosszak(CVk, H_CVk).

hossztalan([], []).
hossztalan([_-CV|H_CVk], [CV|CVk]) :-
  hossztalan(H_CVk, CVk).
```

2.3.8 CLP(fd) jellegű megoldás: *first-fail+most-constrained* elv (ffc)

A first-fail heurisztikát itt tovább javítjuk. Az előző esetben az algoritmus az legkisebb domain-ű változók közül a sorban legelsőt helyettesítette be. Most ezek közül azt fogja választani, amelyre a

legtöbb megszorítás van, bízva abban, hogy ha rosszul is dönt, akkor is így a sok megszorítás miatt gyorsan ellentmondásra jut (most-constrained heurisztika).
Megjegyzés: a constraint-ek szerinti rendezés itt statikus, csak egyszer, az algoritmus elején kell lefuttatni, de általában dinamikusnak kell lennie, mivel behelyettesítéskor változhat a megszorítások száma.

```
terkep_fdfreezeffc(Szk, Ok) :-
  valtozok(Ok, Vk),
  feltetelek(Szk),
  c_rendez(Ok, Szk, Ok1),        % ez új
  ff_behelyettesit(Ok1),
  write(Vk), nl.

% CVk1 a CVk változók átrendezése, úgy hogy a Ck
% constraint-ekben többször elöfordulók legyenek elöbb.
c_rendez(CVk, Ck, CVk1) :-
  elofordulasok_m(CVk, Ck, E_CVk),
  keysort(E_CVk, R_CVk), hossztalan(R_CVk, CVk1).

% elofordulasok_m(CVk, Ck, E_CVk): E_CVk a CVk-nek a Ck-ben
% való elöfordulásszámai negáltjával párosított listája.
elofordulasok_m([], _, []).
elofordulasok_m([CV|CVk], Ck, [E-CV|E_CVk]) :-
  elofordulasai(Ck, CV, E0), E is -E0,
  elofordulasok_m(CVk, Ck, E_CVk).

% elofordulasai(Ck, CV, E) : CV Ck-ben E-szer fordul elö.
elofordulasai([], _, 0).
elofordulasai([V1-V2|Ck], CV, E) :-
  elofordulasai(Ck, CV, E0),
  (   azonos(V1, CV) -> E is E0+1
  ;   azonos(V2, CV) -> E is E0+1
  ;   E = E0
  ).

azonos(V1=_, V2=_) :- V1==V2.
```

2.3.9 `library(clpfd)`-re épülő megoldások

Itt egy valódi, CLP(fd) nyelvű megoldás található. Mivel itt már nincs szükség `fd` változók megvalósítására, sem heurisztikák implementálására, a megoldás meglepően rövid lett. Meg kell adni a változók értékkészletét (`domain`), fel kell venni a kezdeti feltételeket (`clpfd_feltetelek`), és máris el lehet kezdeni a keresést a beépített heurisztikákkal (`labeling`).
Megjegyzés: A programban szereplő `clpfd` könyvtárt használó eljárások a későbbiekben bővebben ismertetve lesznek, itt csak az összehasonlítás kedvéért szerepelnek.

```
:- use_module(library(clpfd)).

terkep_clpfd(Lab, Szk, Ok) :-
  domain(Ok, 1, 3), clpfd_feltetelek(Szk),
  labeling(Lab, Ok), write(Ok), nl.

clpfd_feltetelek([]).
clpfd_feltetelek([P1-P2|Szk]) :-
  P1 #\= P2, clpfd_feltetelek(Szk).
```

labeling lehetőségek

A korábbiakban implementált behelyettesítési (cimkézési) algoritmusok a könyvtárban mind integrálva vannak, ezek azonosítóit a `labeling` eljárásnak kell átadni paraméterként.

1. `Lab = []` ⟶ `terkep_fdfreeze` (fd mut+korutin)
2. `Lab = [ff]` ⟶ `terkep_fdfreezeff` (fd mut+korutin+ff)
3. `Lab = [ffc]` ⟶ `terkep_fdfreezeffc` (fd mut+korutin+ffc)

2.3.10 Futási idők összehasonlítása (Sequent Symmetry 486/50 MHz)

Az összehasonlítást a következő gráfokra végezzük el ($n = 16$):

```
adat(sarkany1, [X-Y,Z-V,V-V1,V1-V2,/*...,Vn,*/X-Z,Z-Y,Y-V],
               [X,Y,Z,V,V1,V2/*,...,Vn*/]).
adat(sarkany2, [X-Y,Z-V,V-V1,V1-V2,/*...,Vn,*/X-Z,Z-Y,Y-V],
               [X,Y,V1,V2/*,...,Vn*/,Z,V]).
adat(sarkany3, [X-Y,V-V1,V1-V2,/*...,Vn,*/Z-V,X-Z,Z-Y,Y-V],
               [X,V1,V2/*,...,Vn*/,Y,Z,V]).
adat(sarkany4, [X-Y,V-V1,V1-V2,/*...,Vn,*/Z-V,X-Z,Z-Y,Y-V,X-V],
               [/*Vn,...*/V2,V1,X,Y,Z,V]).
adat(lanc, [X0-Z0,X0-Y0,Y0-Z0,Z0-X1,Y0-X1, ...., X10-X0],
           [X0,Y0,Z0, ...,X9,Y9,Z9,X10]).
```

Az eredmények:

variáns/adat	sarkany1	sarkany2	sarkany3	sarkany4	lanc
visszalépés	4.25 sec	4.29 sec	3.42 sec	68.19 sec	0.30 sec
dif	0.01 sec	20.31 sec	46.5 sec	470.09 sec	4.38 sec
fd listákkal	0.02 sec	0.02 sec	86.64 sec	780.59 sec	11.10 sec
fd mutábilisekkel	0.01 sec	0.01 sec	35.05 sec	327.94 sec	3.14 sec
fd mut+kortn	0.00 sec	0.01 sec	35.70 sec	241.34 sec	3.14 sec
fd mut+kortn+ff	0.01 sec	0.01 sec	0.01 sec	308.82 sec	7.68 sec
fd mut+kortn+ffc	0.03 sec	0.02 sec	0.02 sec	0.02 sec	8.31 sec
clp(fd), lab=[]	0.03 sec	0.03 sec	183.40 sec	913.75 sec	9.06 sec
clp(fd), lab=[ff]	0.03 sec	0.03 sec	0.05 sec	951.50 sec	9.19 sec
clp(fd), lab=[ffc]	0.04 sec	0.05 sec	0.08 sec	0.03 sec	10.98 sec

Jól látható, hogy az egyre bonyolultabb megoldások heurisztikákkal ötvözve általában nagyságrendi gyorsulást produkálnak - bizonyos adatokon. Viszont egy nem megfelelően megválasztott módszer alaposan lelassíthat egy algoritmust. Univerzális recept nem adható: általában az ismertetetett két heurisztika jó eredményt ad, de vannak feladatok, ahol speciális sorrend megfelelőbb. Fontos megjegyezni, hogy a heurisztikák számításigényesek, így nem szabad feleslegesen alkalmazni őket. (pl. ha minden változóra azonos számú megszorítás van, akkor nem kell `ffc`)

3 SICStus `clpfd` könyvtár

Az alábbi fejezetben a SICStus Prolog CLP(fd) kiterjesztésével fogunk foglalkozni. Ennek használatához a `clpfd` könyvtárt kell betölteni:

```
:- use_module(library(clpfd)).
```

3.1 A `clpfd` kiterjesztés alapjai

Ebben az alfejezetben csak egy vázlatos áttekintést adunk a `clpfd` könyvtárról, a későbbi fejezetekben majd részletes magyarázatot adunk az itt felsoroltakról.

3.1.1 A `clpfd` kiterjesztés elemei

- **Tartomány:** egész számok (negatívak is!)
- **Függvények (aritmetika):**
  - kétargumentumúak: `+, -, *, /, mod, min, max`
  - egyargumentumú: `abs`

  Ezeknek a jelentése a programozási nyelvekben és a matematikában megszokott.
- **Constraint-relációk**
  - aritmetikaiak: `#<, #>, #=<, #>=, #= #\=`
    Ezek a jól ismert Prolog relációjelek megfelelői.
  - halmazműveletek:
    - *X* `in` *Halmaz* - X értékét *Halmaz*-ból veszi
    - `domain(`*Xlist*`,` *Min*`,` *Max*`)` -
      Xlist lista minden változója értékét a*Min* ..*Max* intervallumból veszi

    ahol *Halmaz* lehet:
    - felsorolás: `{`*Szám*, ...`}`
    - intervallum: *Min* `..`*Max*
    - két *Halmaz* metszete: *Halmaz* `/\` *Halmaz*
    - két *Halmaz* úniója: *Halmaz* `\/` *Halmaz*
    - *Halmaz* komplemense: `\` *Halmaz*

    Itt
    - *Min* lehet: *Szám* vagy `inf`, ami azt jelenti, hogy a halmaznak nincs alsó korlátja
    - *Max* lehet: *Szám* vagy `sup`, ami azt jelenti, hogy a halmaznak nincs felső korlátja
- **Primitív constraint-ek (a constraint tár elemei):** *X* `in` *Halmaz*
- **Constraint-megoldó algoritmusok:**
  - aritmetikaiak: constraint-szűkítés: ún. intervallum-konzisztencia
  - halmazműveletek: teljes konzisztencia (ún. tartomány-konzisztencia)

Álljon itt egy egyszerű példa illusztrációként:

```
| ?- X in (10..20)/\ (\{15}), Y in 6..sup, Z #= X+Y.
X in(10..14)\/(16..20), Y in 6..sup,
Z in 16..sup ?
yes
| ?- X in 10..20, X #\= 15, Y in {2}, Z #= X*Y.
Y = 2, X in(10..14)\/(16..20),
Z in 20..40 ?
yes
| ?-
```

3.2 A clp(FD) programok felépítése

Egy clp(FD) program tipikus felépítése az alábbi:

1. deklaráljuk a változók tartományait
2. felállítjuk a constraint-eket
3. behelyettesítjük a constraint-változókat (címkézés)

Az első két pont nem igényel bővebb magyarázatot, annál inkább a harmadik. A constraint-ek felállítása után ugyanis nem mindig kapjuk meg azonnal a kívánt megoldás(oka)t. Ez következik a CLP(fd) nyelv jellegzetességeiből - sok constraint-et a rendszer nem tud azonnal feldolgozni, ezeket fel kell függesztenie. Ezért ilyenkor használni kell a címkézést, ami sorra behelyettesíti az addig behelyettesítetlen változókat, és minden lépés után újra ellenőrzi a constraintek konzisztenciáját. Mivel a constraintek mindig kiértékelhetők, ha változóik be vannak helyettesítve, ezért a címkézés után olyan, minden további feltételtől mentes eredményt kapunk a változókban, ami kielégíti a felállított constraint rendszert.

1. **Változók tartományának deklarálása**

   Alapértelmezésben minden változó értéktartománya `inf..sup`. Ezt lehet tovább szűkíteni:

   - az 3.1.1. pontban ismertetett *X* `in` *Halmaz* predikátummal
   - az ugyanott ismertetett `domain(`*Xlist*`,` *Min*`,` *Max*`)` predikátummal
   - constraintek (következő pont) segítségével, pl `X #>= 0`

2. **Constraintek felállítása**

   Constrainteket a megfelelő relációkba helyettesített constraint-változók segítségével lehet felállítani. Az aritmetikai constraintek gyakran a `#` jellel kezdődő relációjelek. Constrainteket a felhasználó is definiálhat később ismertetett módokon.

   Példa: `X*2 #> Y+4, X #\= Z`

3. **Egyszerű címkézési lehetőségek**

   - `indomain(`*X*`)`: X-et a tartománya által megengedett értékkel helyettesíti (visszalépéssel felsorolja as összes értéket)
   - `labeling(`*Opciók*`,` *Változók*`)`: A *Változók* lista minden elemét behelyettesíti, az *Opciók* lista által előírt módon, pl, `[]`: balról jobbra.

     Opciók `= []`: sorban balról jobbra,
     Opciók `= [ff]`: first-fail elv szerint (legkisebb tartományú változót helyettesíti először),
     Opciók `= [ffc]`: a legkisebb tartományú változók közül azt választja, amely a legtöbb constraint-ben szerepel

3.3 Constraint feladatok megoldása

Ebben a fejezetben olyan feladatokat írunk le, amelyek megoldására tipikusan alkalmazható a CLP(FD) nyelv.

3.3.1 Kódaritmetika: SEND+MORE=MONEY

Minden betű egy számjegyet jelent, a különböző betűk különböző számjegyeknek felelnek meg. Keressük az összes olyan betűkombinációt, amire a 10-es számrendszerbeli SEND+MORE szám egyenlő MONEY-val.

Jól követhető a programban a CLP(FD) programok tipikus felépítése. Először megmondjuk a változók értéktartományát, amit itt a `domain/2` predikátum és az `S #\= 0, M #\= 0` kikötés biztosít. Ezután felvesszük a constrainteket, ami esetünkben egyrészt az, hogy a betűknek megfelelő változók páronként különböznek (`alldiff/1`), másrészt pedig az, hogy `SEND + MORE #= MONEY`. Végül alkalmazzuk a címkézést, amit itt a saját magunk által írt `labeling/1` predikátom végez el.

```
send(SEND,MORE,MONEY):-
  List= [S,E,N,D,M,O,R,Y], domain(List, 0, 9),
  alldiff(List), S #\= 0, M#\= 0,
  SEND #=  1000*S+100*E+10*N+D,
  MORE #= 1000*M+100*O+10*R+E,
  MONEY #= 10000*M+1000*O+100*N+10*E+Y,
  SEND+MORE #= MONEY, labeling(List).
```

```
% labeling(Xs) ekvivalens a beépített labeling([],Xs) eljárással
labeling([]).
labeling([X|Xs]) :- indomain(X), labeling(Xs).

% Azonos a beépített all_different eljárással
alldiff([]).
alldiff([X|Xs]) :- outof(X, Xs), alldiff(Xs).

outof(_, []).
outof(X, [Y|Ys]) :- X #\= Y, outof(X, Ys).
```

3.3.2 A zebra feladat

Szintén egy CLP(FD)-ben könnyen megfogható feladat: adott 5 különböző nemzetiségű ember, 5 különböző színű ház, 5 különböző foglalkozás, 5 különböző állat és 5 különböző ital. Egy embernek pontosan egy foglalkozása, nemzetisége, háza, kedvenc itala és állata van. A feladat az, hogy az ismert kötöttségek alapján megmondjuk, hogy melyik nemzetiségű ember kedvenc állata a zebra. Az embereket megsorszámozzuk 1-től 5-ig, majd minden egyes foglalkozáshoz, nemzetiséghez, házhoz, italhoz és állathoz egy constraint változót rendelünk, amely értékét az 1..5 halmazból veszi fel. Egy ilyen változó i értéke azt jelenti, hogy az általa reprezentált tulajdonság az i. számú emberhez tartozik. A kötöttségeket ez alapján könnyen felírhatjuk, például az, hogy a diplomata háza a sárga, egyszerűen a két megfelelő constraint-változó egyenlővé tételével biztosítható. A kötöttségek a programból könnyen kideríthetők, itt nem soroljuk fel őket. Az egyetlen figyelmet érdemlő feltétel az az, hogy valaki egy adott ház szomszédjában lakik, amit a `nextto/2` predikátum valósít meg. Ez tulajdonképpen egy vagylagos szerkezet, amit a Prolog választási pontokkal valósít meg - vagyis spekulatív módon veszi fel a vagy-kapcsolatban szereplő constrainteket, és minden lehetőségre megpróbálja megoldani az aktuális constraint rendszert. Ez nem a legszerencsésebb megoldás, mivel sok választási pont esetén rengeteg időt igényelhet. A vagylagos szerkezetekről a későbbiekben még lesz szó. Adott esetben a

```
nextto(A,B):- abs(A-B) #= 1.
```

megoldás sokkal szerencsésebb lenne , mivel ez nem generál választási pontokat.

```
:- use_module(library(lists)).
:- use_module(library(clpfd)).

zebra(ZOwner, All):-
  All = [England,Spain,Japan,Norway,Italy,
         Green,Red,Yellow,Blue,White,
         Painter,Diplomat,Violinist,Doctor,Sculptor,
         Dog, Zebra, Fox, Snail, Horse,
         Juice,Water,Tea,Coffee,Milk],
  domain(All, 1, 5),
  alldiff([England,Spain,Japan,Norway,Italy]),
  alldiff([Green,Red,Yellow,Blue,White]),
  alldiff([Painter,Diplomat,Violinist,Doctor,Sculptor]),
  alldiff([Dog,Zebra,Fox,Snail,Horse]),
  alldiff([Juice,Water,Tea,Coffee,Milk]),
  England = Red,          Spain = Dog,
  Japan = Painter,        Italy = Tea,
  Norway = 1,             Green = Coffee,
  Green #= White+1,       Sculptor = Snail,
  Diplomat = Yellow,      Milk = 3,
  Violinist = Juice,      nextto(Norway, Blue),
  nextto(Fox, Doctor),    nextto(Horse, Diplomat),
  labeling([ff], All),
  nth(N, [England,Spain,Japan,Norway,Italy], Zebra),
```



```
  nth(N, [england,spain,japan,norway,italy], ZOwner).

nextto(A, B) :-  A #= B+1.
nextto(A, B) :-  A #= B-1.
```

3.3.3 N királynő a sakktáblán

A feladat elhelyezni egy $N * N$-es sakktáblán N királynőt úgy, hogy egyik se üsse a másikat.

A megoldás minden királynőt külön oszlopba tesz, majd mindegyikükhöz egy változót rendel, ami a királynő oszlopbeli pozícióját adja meg. A változók tartományának deklarálása után minden két királynő között felállít egy constraint-et (`no_threat/3`), ami azt adja meg, hogy a két királynő nem üti egymást. Ehhez meg kell adni a két királynő oszlopainak távolságát, ami itt az `I` paraméter. Ezek után a program végrehajtja a címkézést first-fail heurisztikával.

```
:- use_module(library(clpfd)).

% A Qs lista N királynő biztonságos elhelyezését
% mutatja egy N*N-es sakktáblán. Ha a lista
% i. eleme j, akkor az i. királynőt az i. sor j.
% oszlopába kell helyezni.
queens(N, Qs):-
  length(Qs, N), domain(Qs, 1, N),
  safe(Qs),
  labeling([ff],Qs).  % first-fail elv

% safe(Qs): A Qs lista a királynők biztonságos
% elhelyezését írja le.
safe([]).
safe([Q|Qs]):-
  no_attack(Qs, Q, 1),
  safe(Qs).

% no_attack(Qs, Q, I): A Qs lista által leírt
% királynők egyike sem támadja a Q oszlopban levő
% kiralynőt, feltéve hogy Q és Qs távolsága I.
no_attack([],_,_).
no_attack([X|Xs], Y, I):-
  no_threat(X, Y, I),
  J is I+1, no_attack(Xs, Y, J).

% Az X és Y oszlopokban  I sortávolságra levő
% királynők nem támadják egymást.
no_threat(X, Y, I) :-
  Y #= X, Y #= X-I, Y #= X+I.
```

3.4 A constraint-ek működése

3.4.1 A constraint erősítés fajtái

Legyen C egy n-változós constraint:

- jelöljük $D(X, s)$-sel az X változó tartományát az s constraint-tárban
- jelöljük $D'(X, s)$-sel a $min(D(X, s))..max(D(X, s))$ intervallumot.

Tartomány-konzisztencia (domain consistency):

> *C tartomány-konzisztens* s-re nézve, ha bármelyik X_i változójához és annak tetszőleges $V_i \in D(X_i, s)$ megengedett értékéhez található a többi változónak olyan $V_j \in D(X_j, s)$ megengedett értéke ($j = 1, \ldots, i-1, i+1, \ldots, n$), hogy $C(V_1, \ldots V_n)$ fennálljon.

Intervallum-konzisztencia (interval consistency):

> *C intervallum-konzisztens* s-re nézve, ha bármelyik X_i változójához és tartománya mindkét végpontjához, azaz a $V_i = min(D(X, s))$ és $V_i = max(D(X, s))$ értékekhez található a többi változónak olyan $V_j \in D'(X_j, s)$ értéke ($j = 1, \ldots, i-1, i+1, \ldots, n$), hogy $C(V_1, \ldots V_n)$ fennálljon.

A tartomány-konzisztencia a lehető legerősebb következtetést, a legszűkebb constraint-tárat garantálja, de időigénye miatt nem mindig célszerű használni. Ilyenkor használják helyette az intervallum-konzisztenciát. Az intervallum-konzisztencia olcsóbb, viszont gyengébb következtetési képességgel rendelkezik.
Az aritmetikai constraint-ekre a SICStus clpfd könyvtár csak intervallum konzisztenciát garantál, de bizonyos esetekben tartomány-konzisztenciát is biztosít.

Példák

```
| ?- X in {4,9}, Y in {2,3}, Z #= X-Y.    % intervallum-konz.:
X in{4}\/{9}, Y in 2..3, Z in 1..7 ?
| ?- X in {4,9}, Y=2, Z #= X-Y.           % tartomány-konz.:
Y = 2, X in{4}\/{9}, Z in{2}\/{7} ?

| ?- X in 1..10, U=3, X #\= Z-U, Z=8.     % tartomány-konz.:
U = 3, Z = 8, X in(1..4)\/(6..10) ?
| ?- X in 1..10, X #\= Z-U, Z=8, U=3.     % intervallum-konz.:
U = 3, Z = 8, X in 1..10 ?
```

3.4.2 Példa: mágikus sorozatok

Egy $L = (x_0, \ldots, x_{N-1})$ sorozat mágikus ($x_i \in [0..N-1]$), ha L-ben az i szám pontosan x_i-szer fordul elő (minden $i \in [0..N-1]$-re). Például $n = 4$ esetén $(1, 2, 1, 0)$ és $(2, 0, 2, 0)$ mágikus sorozatok. Legyen a feladat adott N-re a a mágikus sorozatok megadása.

Egyszerű megoldás a következő, ami a definíción alapul: generálunk egy N hosszú változólistát, és megadjuk, hogy az i, $0 \leq i \leq N-1$ szám a lista i. eleme számúszor szerepel magában a listában. Azt, hogy egy X szám M-szer fordul elő egy listában, a `pontosan/3` predikátum valósítja meg. Ez jelen esetben Prolog spekulatív kereséssel működik, két predikátumot alkalmaz külön-külön arra az esetre, ha egy elem azonos vagy nem azonos X-szel.

```
% Az L lista egy N hosszúságú mágikus sorozat.
magikus(N, L) :-
        length(L, N),
        N1 is N-1, domain(L, 0, N1),
        elofordulasok(L, 0, L).

% elofordulasok([Ei, ...], i, Sor): Sor-ban i, i+1, ...
% előfordulásainak száma Ei, Ei+1, ...
elofordulasok([], _, _).
elofordulasok([E|Ek], I, Sor) :-
        pontosan(I, Sor, E), J is I+1, elofordulasok(Ek, J, Sor).

% pontosan(I, L, E): Az I szám L-ben pontosan E-szer fordul elö.
pontosan(I, L, 0) :- outof(I, L).
pontosan(I, [I|L], N) :- N #> 0, N1 #= N-1, pontosan(I, L, N1).
pontosan(I, [X|L], N) :- N #> 0, X #\= I,   pontosan(I, L, N).
```


A megoldást gyorsíthatjuk, ha további constrainteket veszünk fel. Ezek a constraint megoldó keresési terét szűkítik, kizárnak a térben olyan pontokat, ahol biztosan nincs megoldás. Ezek constraintek itt egyrészt azt a tényt írják le, hogy a listában szereplő számok összege N (`osszege/2`), másrészt pedig azt a könnyen bebizonyítható állítást, hogy a lista i. helyén álló számot i-vel megszorozva és ezt a lista minden elemére összegezve az eredmény N lesz (`szorzatosszege/3`).

```
% Hatékonyság-fokozás redundáns (surrogate) constraint-ekkel.
magikus1(N, L) :-
        length(L, N), N1 is N-1, domain(L, 0, N1),
        osszege(L, S), call(S #= N),
        szorzatosszege(L, 0, SS), call(SS #= N),
        elofordulasok(L, 0, L).

osszege([], 0).
osszege([X|L], X+S) :- osszege(L, S).

szorzatosszege([], _, 0).
szorzatosszege([X|L], I, I*X+S) :-
        J is I+1, szorzatosszege(L, J, S).
```

3.4.3 Reifikáció

Az előzőekben megadott `pontosan/3` eljárással gondok vannak. Egyrészt az első argumentumában konkrét értéket ad, vagyis azt nem constraint-változóként kezeli, másrészt Prolog választási pontokat hoz létre. Tulajdonképpen nem illik a constraint-világba, mivel egyszerűen egy visszalépésen alapuló próbálgatásos megoldást ad, aminek az időigénye meglehetősen nagy lehet. Az ilyen jellegű problémákra (is) nyújt megoldást a reifikáció.

Egy constraint reifikációja:
a constraint igazságértékének "tükrözése" egy $0-1$ értékű constraint-változóban.

Egy reifikált constraint formája a `clpfd` könyvtárban:

> *Constraint* `#<=>` *Constraint-változó*

Általában nem minden constraint reifikálható. Az aritmetikai és halmaz constraint-ek mind reifikálhatóak.

Egy *Constraint* `#<=>` B reifikált constraint végrehajtása

1. egy démon figyeli, hogy a *Constraint* a constraint-tárból következik-e, ha igen, $B = 1$;
2. egy démon figyeli, hogy a constraint-tárból következik-e a *Constraint* negáltja, ha igen, $B = 0$;
3. egy démon figyeli, hogy B értéket kap-e, ha $B = 1$, felveszi a *Constraint*-et, ha $B = 0$, felveszi a *Constraint* negáltját a constraint-tárba.

Látható, hogy a reifikáció pontosan, oda-vissza megvalósítja a *Constraint* igazságértékének a "tükrözését" a B változóba, vagyis mind a B értéket kapása, mind a *Constraint* levezethetőségének vagy le nem vezethetőségének biztossá válására reagál.

Példa: `(X #= 3) #<=> B`. `B` jelentése: az `X=3` egyenlőség igazságértéke.

3.4.4 Constraint-következtetés (entailment)

Itt 3.4.1. pontban megadott jelöléseket használjuk fel.

Tartomány-következtetés:

> *C tartomány-következik* s-ből, ha X_i változói minden megengedett $V_j \in D(X_j, s)$ értékére ($j = 1, \ldots, n$), $C(V_1, \ldots V_n)$ fennáll.

Intervallum-következtetés:

> C *intervallum-következik* s-ből, ha minden $V_j \in D'(X_j, s)$ értékre $(j = 1, \ldots, n)$, $C(V_1, \ldots V_n)$ fennáll.

A tartomány-következtetés használatával a lehető leghamarabb észre lehet venni, hogy egy constraint levezethető a constraint-tárból. A intervallum-következtetés erősebb kikötés, mint a tartomány-következtetés, de a tartomány-következtetés időigényes volta miatt gyakran ezt használják. Ez esetben lehetséges, hogy egy constraint már levezethető a tárból, de azt az intervallum-következtetés miatt a rendszer még nem veszi észre.
A SICStus clpfd könyvtára a reifikált aritmetikai constraint-ekre intervallum-következtetést garantál.

3.4.5 Reifikált constraintek felhasználása

- **Példák tüközött constraint-ekre**

```
| ?- X in 1..4, X #< Y #<=> B, X+Y #=9.
B = 1, X in 1..4, Y in 5..8 ?
| ?- X+Y #= Z #<=> B, X=1, Z=6, Y in 1..10, Y#\=5.
X = 1, Z = 6, Y in(1..4)\/(6..10), B in 0..1 ?
```

- **`pontosan/3` tükrözött constraint-ekkel**

  Itt a 3.4.2. pontban megadott mágikus sorozatokat előállító program "szívét", a `pontosan/3` eljárást adjuk meg reifikált constraintek felhasználásával.

```
pontosan(_, [], 0).
pontosan(I, [X|L], N) :-
  X #= I #<=> B, N #=N1+B, pontosan(I, L, N1).
```

  Ebben az `I`-vel azonos elemek számát, `N`-et adjuk meg az `L` listában, és ezt reifikált constraintek segítségével tesszük meg: minden `X` listaelemhez felveszünk egy tükrözött constraint-et, aminek `B` változója a `X = I` igazságértékét tartalmazza. A `pontosan/3` eljárás ezeket a `B` változókat összegzi, és teszi egyenlővé `N`-nel.

  A reifikáció hatékonysága: mágikus sorozatok (586/133Mhz)

  Az alábbi táblázat egy összehasonlítást tartalmaz, a mágikus sorozatokat előállító programok példáján. Jól látható, hogy a reifikáció felhasználása nagyságrendi gyorsulást eredményezett.

variáns/adat	n=8	n=10	n=12	n=14	n=20	n=30
választós	10.17	227.96				
választós+osszege	1.35	6.26	28.69	127.29		
vál.+szorzatosszege	0.27	0.66	1.32	2.54		
vál.+ossz.+szorzossz.	0.20	0.48	1.02	1.90	10.59	108.94
reifikálós	1.20	2.76				
reifikálós+osszege	0.47	0.95	1.73	2.54		
reif.+szorzatosszege	0.20	0.40	0.54	0.72		
reif.+ossz.+szorzossz.	0.25	0.43	0.62	0.82	1.89	4.83

- **A tükrözés általánosítása: logikai müveletek constraint-ekkel**

  A műveletek, amiket meg akarunk valósítani:

  - `#\ Q`: `Q` negáltja;
  - `P #/\ Q`: konjunkció;
  - `P #\/ Q`: diszjunkció;
  - `P #\ Q`: kizáró vagy;
  - `P #=> Q` vagy `Q #<= P`: implikáció;
  - `P #<=> Q`: ekvivalencia.

  ahol `P` és `Q` tetszőleges reifikálható constraint (pl aritmetikai), vagy 0..1 értékű constraint-változó.

  A logikai müveletek reifikált és aritmetikai constraint-ekre vezethetők vissza, pl:
  `(X #= 4) #\/ (Y #> 6)` egy megvalósítása:

```
(X #= 4) #<=> B1, (Y #> 6) #<=> B2, B1+B2 #> 0
```

  Megjegyzés: A logikai műveletek reifikációval történő megvalósítása nem mindig a legjobb megoldás. Például diszjunkció esetén a reifikált változat egészen addig fel sem ébred, amíg legalább az egyik constraint levezethetősége vagy le nem vezethetősége el nem dönthető, holott sok esetben más módszerrel már ez előtt is szűkítéseket lehet eszközölni a constraint-tárban. Például

$$X \in (0..6), Y \in (0..6), (X + 5 \leq Y) \vee (Y + 5 \leq X)$$

  esetén a

$$X \in (0..1) \vee (5..6), Y \in (0..1) \vee (5..6)$$

  kikötések tehetők, de ezt reifikációval nem tudjuk észrevenni, mivel egyik constraint sem lett levezethető vagy nem levezethető.

3.5 Constraintek feldolgozása

Egy constraint feldolgozása a következő lépésekből áll:

1. **A constraint kifejtése elemi constraintekre**
   Ez a lépés a felhasználói constrainteket a constraint-megoldó számára érthető, egyszerű constraintekre bontja szét. Alapértelmezésként fordítási időben történik a kifejtés, de ez a `call/1`-be ágyazással elhalasztható a constraint végrehajtásáig. Az elhalasztásnak a célja az lehet, hogy bizonyos esetekben egyszerűbb elemi constraintekre fordul a felhasználói constraint, ha nem fordítási időben történik a kifejtés. Ilyen eset az, ha egy változó futási időben kap értéket, és így a constraint végrehajtásakor a rendszer tudja, hogy egy konstanssal nem pedig egy constraint-változóval van dolga. Erre már láttunk példát a 3.3.3. pontban, az *N királynő a sakktáblán* problémánál. Lehet kísérletezni, hogy a felhasználói constrainteket milyen elemi constraintekre fejti ki a `clpfd` könyvtár: ehhez felhasználható a

   `goal_expansion(`*Constraint*`, user, Kifejtés)`

   felhasználó által definiálható hook (kampó) eljárás.
2. **A constraint végrehajtása (posting)**
   Ebben a lépésben "élesítjük" a constraintet, vagyis először megállapítjuk, hogy mikor kell felébreszteni, majd felébresztjük a constraintet. A felébresztés történhet a változói alsó, felső határának változásakor, a tartomány bármilyen változásakor vagy értékadáskor.
3. **A constraint felébresztése**
   Ez a lépés sokszor ismétlődhet, ez valósítja meg a constraint-erősítést.

3.5.1 Elemi constraintek

Ezek a constraint-megoldó által ismert elemi megszorítások, amelyek Prolog predikátumokként vannak megvalósítva, olyan nevekkel, amelyek hűen tükrözik jelentésüket. A nevekben szereplő `x`, `y`, `t` szimbólum változót jelent, az `a`, `b`, `c` konstansot.

- **Aritmetikai elemi constraintek:**
  `'x+y=t'/3 'x*y=z'/3 'x/y=z'/3 'x mod y=z'/3 '|x|=y'/2 'max(x,y)=z' 'min(x,y)=z'`

- **Összehasonlítást végző elemi constraintek:**
  `'x=y'/2, 'x=<y'/2, 'x\=y'/2`
  és reifikált változataik:
  `iff_aux('x` *Rel* `y'(X,Y), B)`,
  ahol *Rel* a `=` `=<` `\=` jelek valamelyike.
- **Optimalizálások:**
  `'x*x=y'/2 'ax=t'/3 'ax+y=t'/4 'ax+by=t'/5 't+u=<c'/3 't=u+c'/3 't=<u+c'/3`
  `'tū+c'/3 't>=c'/2` stb.

Példák:

`abs(X-Y)#>1` ⟶ `'x+y=t'(Y,A,X),'|x|=y'(A,B),'t>=c'(B,2)`
`(X#<3)#<=>B` ⟶ `iff_aux('x=<y'(X,2),B)`
`X*X+Y*Y#=<16` ⟶ `'x*x=y'(X,A),'x*x=y'(Y,B),'t+u=<c'(A,B,16)`

3.5.2 Constraintek kifejtése

Amikor egy constraintet egyszerűbb constraintek konjunkciójává fejtünk ki, akkor gyengülhet a konzisztenciája. A SICStus `clpfd` könyvtára garantálja, hogy lineáris kifejezés esetén megmarad az intervallum-konzisztencia. Az általános esetre ez nem mindig igaz, mert például a

```
| ?- X in 0..10, X*X*X*X#=16.
X in 1..4 ? ;
no
```

nem intervallum-konzisztens eredményt szolgáltat. (Megjegyzés: megfelelő zárójelezéssel a helyes eredmény megkapható). Sok esetben viszont nem-lineáris constraintekre is biztosít konzisztenciát, például prímszámok ellenőrzésére alkalmas az alábbi constraint:

```
| ?- domain([X,Y], 0, 5000), X-Y#>1, X*X-Y*Y#=6211.
no
```

3.5.3 Constraint erősítés speciális esetei

Emlékeztető: a constraint-erősítés logikai megfogalmazása:

$$< C, s > \longrightarrow < C', s \wedge c >$$

ahol

$$s \models C \equiv C' \wedge c$$

Az erősítésnek az alábbi speciális esetei merülnek fel a clp(fd) végrehajtásakor:

- ha $s \wedge c$ inkonzisztens, akkor visszalépünk;
- ha $C' =$ `true`, a constraint levezethetővé vált, azaz sikeresen lefutott;
- ha $C' \neq C$, akkor C lefutott, helyette végrehajtjuk C'-t, pl. reifikációnál, vagy `max(X,Y) #= Z` ⟶ `X #= Z`, ha `Y #=< X`;
- ha $C' = C$, akkor (újra) elaltatjuk a constraintet.

3.5.4 Constraint erősítés megvalósítása

A constraint-erősítés célja: az adott constraint és egy adott tár esetén kiszűrni azokat az értékeket, amelyekre a constraint nem teljesülhet. Ennek megvalósítása különbözik reifikált és nem reifikát constraintek esetén.

Közvetlen (nem reifikált) constraintek esetén képezzük a constraint és a tár metszetét, majd:

- tartomány-konzisztencia esetén: a változó-tartományok minden pontját próbáljuk szűrni, pl. `abs(X-Y)#>1, Y in 2..4` → `X#\=3`;

- intervallum-konzisztencia biztosításakor: a tartomány végpontjait próbáljuk kiszűrni, a többi változó tartományát intervallumnak tekintve (ha sikerül, ismételjük);

Reifikált constraintek esetén az eljárás bonyolultabb. Már említettük, hogy minden egyes reifikált constraintet három démon figyel: az egyik a constraint logikai változóját, hogy értéket kap-e, másik kettő pedig a constraint, illetve negáltja tárból való levezethetőségét vizsgálja. Az megvalósítás:

- Ha a logikai változó értéket kap, a kérdezett constraint vagy negáltja lép a reifikált constraint helyébe.
- Ha a kérdezett constraint vagy negáltja a tárból levezethetővé válik, akkor a logikai változó szűkül: az 1 vagy 0 értéket kapja, annak megfelelően, hogy a constraint vagy a negáltja lett levezethető. A levezethetőséget kétféleképpen lehet vizsgálni:
  - tartomány-levezethetőség vizsgálata: ha minden a tár által megengedett behelyettesítésre igaz a constraint, pl. `Y in {0,5}, X in 2..3` tárból `abs(X-Y)#>1` tartomány-levezethető;
  - intervallum-levezethetőség vizsgálata: ha a tárban minden tartományt intervallummá általánosítunk, akkor is mindig igaz, pl `abs(X-Y)#>1` a fenti tárból nem intervallum-levezethető.

3.6 További beépített constraintek és eljárások

Ebben a fejezetben a SICStus `clpfd` könyvtárának beépített eljárásaival ismerkedünk meg. Nem ismertetjük az beépített aritmetikai constrainteket, mivel azokkal már megismerkedtünk.

3.6.1 Kombinatorikus (szimbolikus) constraintek

Ezek a constraintek nem reifikálhatóak.

`element(X, CLista, Y)`
Jelentése: `CLista` `X`-edik eleme `Y`, ahol `X` és `Y` constraint-változók (vagy egészek), `CLista` egészekből álló lista (a következő SICStus Prolog változatban (#6-ban) lehetnek benne változók is).
Tartomány-konzisztenciát biztosít. Példák:

```
| ?- element(X, [0,1,2,3,4], Y), X in {2,5}.  % Y #= X-1
                        X in{2}\/{5}, Y in{1}\/{4} ?
```

```
% X #= c #<=> B megvalósítása, 1 =< X,c =< 6 esetére.
beq(X, C, B) :-
  X in 1..6, call(I #= X+6-C),
  element(I, [0,0,0,0,0,1,0,0,0,0,0], B).
```

`relation(X, Hozzárendelés, Y)`
Itt `X` és `Y` constraint-változók (vagy egészek), `Hozzárendelés` *egész* - *Halmaz* alakú párok listája (ahol mindegyik *egész* csak egyszer fordulhat elő). Jelentése: `Hozzárendelés` tartalmaz egy `X-Halmaz` párt, ahol `Y` eleme a `Halmaz`-nak. Tetszőleges bináris reláció definiálására használható.
Tartomány-konzisztenciát biztosít. Példa:

```
'abs(x-y)>1'(X,Y) :-
  relation(X, [0-(2..5), 1-(3..5), 2-{0,4,5},
               3-{0,1,5}, 4-(0..2), 5-(0..3)], Y).
```

`'abs(x-y)>1'(X,Y), X in 2..3.` → `Y in(0..1)`∨`(4..5)`

`all_different(Változók)`
A Változók páronként különbözőek. Megvalósítása nem teljes, a korábbi `alldiff` eljárással azonos.

`all_distinct(Változók)`
Jelentése ugyanaz, mint `all_different`, de teljes (tartomány-konzisztenciát biztosít). A `library('clpfd/all_dist')` könyvtárban van. Például:

```
| ?- domain([X,Y,Z], 1, 2), all_distinct([X,Y,Z]).
no
```

3.6.2 Címkéző eljárások

Az alábbi eljárások a címkézést, az úgynevezett labelinget valósítják meg. A címkézésről már korábban beszéltünk, és az eljárások egy részét is ismertettük már, de itt egy teljes leírást adunk minden egyes eljárás mellé az esetleges opciók magyarázatával.

`indomain(X)`
X-et egy, a tartománya által megengedett értékkel helyettesíti. Visszalépésnél felsorolja as összes lehetséges értéket.

`minimize(Cél, X)` ill. `maximize(Cél, X)`
A **`Cél` ismételt hívásával** megkeresi az `X` változó minimális ill. maximális értékét. A módszer az, hogy először megkeres egy értéket, `a`-t, majd felveszi a `X < a` constraintet, és megint elkezdi a keresést. Így folytatja addig, amíg a keresés meghiúsul. Ilyenkor visszadja `X`-ben az utolsónak megtalált helyes értéket, ez lesz a minimális. Maximálisat analóg módon, az `X > a` constraint felvételével keres. Például egy szám legkisebb prím osztójának keresése:

```
| ?- domain([X,P], 2, 4000), P*X#=7373,
     minimize(labeling([], [P,X]), P).
```

`labeling(Opciók, Változók)`:
A `Változók` lista minden elemét behelyettesíti, az `Opciók` lista által előírt módon. Az alábbi négy csoport mindegyikéből legfeljebb egy opció szerepelhet.

- Változó kiválasztása:
  - `leftmost` — a legelső változó (alapértelmezés);
  - `min` ill. `max` — a legkisebb alsó határú ill. a legnagyobb felső határú változó;
  - `ff` — first-fail elv: a legkisebb tartományú változók közül a legbaloldalibb;
  - `ffc` — a legkisebb tartományúak közül a legtöbb constraintben előforduló (ha több van, ezek közül a legbaloldalibb).
- A tartomány felbontása a kiválasztott `X` változóra:
  - `step` — `X #= B` és `X #\= B` közötti választás, ahol `B` az `X` tartományának alsó vagy felső határa, a bejárási iránytól függően (alapértelmezés);
  - `bisect` — `X #< M` és `X #>= M` közötti választás, ahol `M` az `X` tartományának középső eleme.
- A tartomány bejárási iránya:
  - `up` — alulról felfelé (alapértelmezés);
  - `down` — felülről lefelé.

  Ez `step` esetén azt határozza meg, hogy az `B` az alsó vagy a felső határ legyen, `bisect` esetén pedig azt, hogy `X #< M` vagy az `X #>= M` constraintet vegye fel először a rendszer.
- A keresett megoldások :
  - `all` — visszalépéssel az összes megoldást felsorolja; (alapértelmezés)
  - `minimize(X)` ill. `maximize(X)` — csak egy megoldást keres, `X`-re vonatkozóan a minimálisat ill. a maximálisat. Abban különbözik a `minimize/2`, ill. a `maximize/2` eljárástól, hogy itt a szélsőérték keresése nem új constraint felvételével és újraindítással történik, hanem a címkéző eljárás van úgy megvalósítva, hogy mindig a megfelelő szélsőértéket találja meg.

3.6.3 Egyéb eljárások

Tartományok lekérdezése, statisztika, válaszok

`fd_min(X, Min)`
`fd_max(X, Max)`
`fd_size(X, Size)`
Az `X` változó tartományának pillanatnyi alsó határa `Min`, felső határa `Max`, mérete `Size`. Ha valamelyik nem korlátos, az eljárás meghiúsul.

`fd_dom(X, Dom)`
`Dom` az `X` tartományát leiró halmazkifejezés. Az eljárás a következő verzióban megszűnik.

`fd_set(X, Set)`
`Set` az `X` tartományának belső ábrázolása. A belső ábrázolás kezelése belső adattípusként valósul meg, lásd a 3.9.2. pontot.

`fd_statistics(Kulcs, Érték)`
A `Kulcs`-hoz tartozó számláló `Érték`-ét kiadja és lenullázza. Lehetséges kulcsok és számlált események:

- `constraints` — constraint létrehozása;
- `tells` — közvetlen constraint felébresztése;
- `asks` — reifikált constraint felébresztése; `tell_entailments` — közvetlen constraint levezethetővé válása;
- `ask_entailments` — reifikált constraint levezethetővé válása;
- `prunings` — tartomány szűkítés; `backtracks` — a tár ellentmondásossá válása.

`fd_statistics`
Az összes számláló állását kiírja és lenullázza őket.

`clpfd:full_answer`
Ez egy dinamikus kampó eljárás. Ha nem sikerül, akkor a rendszer egy kérdésre való válaszoláskor csak a kérdésben előforduló változókat írja ki, és a még alvó constrainteket nem írja ki. Ha sikerül, kiírja az összes változót és a le nem vezetett constrainteket.

Ütemezési (szimbolikus) constraint

`cumulative(Starts, Durations, Resources, Limit)`
Az első három argumentum változókból/egészekből álló egyforma hossszú lista, az utolso egy változó/egész. Jelentése: a `Starts` kezdőidőpontokban elkezdett, `Durations` ideig tartó és `Resources` erőforrásigényű feladatok bármely időpontban összesített erőforrásigénye nem haladja meg a `Limit` határt. Jelöljük: *a* `= min(`*S1*`,...,`*Sn*`)`, *b* `= max(`*S1*`+`*D1*`,...,`*Sn*`+`*Dn*`)`, *Rij* `=` *Rj*, ha *Sj* `=<` *i* `<` *Sj*`+`*Dj*, egyébként *Rij* `= 0`; ekkor a constraint jelentése: *Ri1*`+...+`*Rin* `=<` *Limit*, minden *a* `=<` *i* `<` *b* esetén.

3.6.4 Példa: egy egyszerű ütemezési feladat

A feladat: a rendelkezésre álló erőforrások száma: 13, az egyes tevékenységek időtartama és erőforrásigénye:

Tevékenység	t1	t2	t3	t4	t5	t6	t7
Időtartam	16	6	13	7	5	18	4
Erőforrásigény	2	9	3	7	10	1	11

Keressük tevékenységek kezdési időpontjait, feltéve, hogy egy időpontban sem lehet 13-nál több erőforrást használni. A megoldás:

```
:- use_module(library(clpfd)).
:- use_module(library('clpfd/cumul')).

schedule(Ss, End) :-
        length(Ss, 7),
        Ds = [16, 6,13, 7, 5,18, 4],
        Rs = [ 2, 9, 3, 7,10, 1,11],
        domain(Ss, 1, 30),
        domain([End], 1, 50),
        after(Ss, Ds, End),
        cumulative(Ss, Ds, Rs, 13),
        labeling([ff,minimize(End)], [End|Ss]).

after([], [], _).
after([S|Ss], [D|Ds], E) :- E #>= S+D, after(Ss, Ds, E).
```

A felhasználása:

```
| ?- schedule(Ss, End).

Ss = [1,17,10,10,5,5,1],
End = 23 ?
```

3.7 Felhasználói constraintek definiálása

Ebben a fejezetben megismerkedünk az új constraintek definiálásának módjaival. Eddig a constraintek átlátszatlanok voltak a felhasználó számára, azokat "black box"-nak, azaz "fekete doboz"-nak tekintettük. Most ezt a megközelítésmódot felváltja a "glass box", azaz "üvegdoboz" megközelítésmód.

Egy constraint definiálásakor meg kell határozni az alábbiakat:

1. milyen tár-szűkítést végezzen el;
2. mikor ébredjen fel újabb szűkítés megkísérlése érdekében;
3. mikor fejezheti be a működését (mikor válik levezethetővé).

Ha a constraintet reifikálni is akarjuk:

1. hogyan kell végrehajtani a negáltját;
2. hogyan döntsük el a tárból való levezethetőségét;
3. hogyan döntsük el a negáltjának a levezethetőségét.

Az eddigiekre nézzünk egy példát. Az előzőekben ismertetett `'x+y=t'(X, Y, T)` elemi constraintet akarjuk megvalósítani intervallum konzisztenciával. Egy lehetséges megoldási elv:

1. `T` tartományából kihagyjuk a `min(X)+min(Y)`-nél kisebb és a `max(X)+max(Y)`-nál nagyobb elemeket; ezt ismételjük, valahányszor `X` vagy `Y` alsó vagy felső határa változik, mindaddig, míg `X` és `Y` értéket nem kap.
2. `X`-ből kihagyjuk a `min(T)-max(Y)`-nél kisebb és a `max(T)-min(Y)`-nál nagyobb elemeket; ezt ismételjük ha `T` vagy `Y` határa változik.
3. hasonlóan szűkítjük `Y`-t is.

Constraint definiálási lehetőségek SICStusban

- FD predikátumok: az úgynevezett indexikálisok segítségével deklaratív módon lehet constrainteket definiálni, reifikáltakat is.
- Globális constraintek: Prologban kell megadni a fentieket, reifikálás nem lehetséges. Ez a SICStus 3#5 változatban nem publikus.

3.8 FD predikátumok

A SICStus Prologban az úgynevezett indexikális nyelv segítségével deklaratív módon definiálhatunk új constrainteket. A `'x+y=t'/3` elemi constraintre az előbbiekben megadott megoldási elvet a következő FD predikátummal valósíthatjuk meg:

```
'x+y=t'(X,Y,T) +:
        X in min(T) - max(Y)..max(T) - min(Y),
        Y in min(T) - max(X)..max(T) - min(X),
        T in min(X) + min(Y)..max(X) + max(Y) .
```

Az indexikálisok halmazértékű függvények, amelyeket FD predikátumokban használhatunk. Egy FD predikátum 1 vagy 4 klózból áll, nyakjeleik a `+:`, `+?`, `-+`, és a `-?`. A `+:` nyakjelű kötelező, a `-:`, `+?`, `-?` nyakjelűek csak reifikálandó constraintek esetén szükségesek.

- A `+:` `-:` ún szűkítő (mondó, *tell*) indexikálisok az adott constraint és negáltja tárra gyakorolt hatását írják le.
- A `+?` `-?` ún kérdező (*ask*) indexikálisok az adott constraint és negáltja levezethetőségi feltételeit definiálják.

Az FD predikátum fejében az argumentumok különböző változók; törzsében csak ezeket a változókat tartalmazó indexikálisok lehetnek.

Az indexikálisok alakja:

Változó `in` *Tartománykifejezés*

ahol *Változó* egy, a fejben felsorolt változó, *Tartománykifejezés* az **összes** többi változót tartalmazó kifejezés. A *Tartománykifejezés* (angolul *Range*) egy halmazértékű függvény, amely a benne szereplő változóktól függően egy tartományt generál, azt a tartományt, amelyre a *Változó* értéktartományát a predikátum által leírt constraint szerint szűkíteni kell.

Alapszabály: Ha `X in R` szerepel egy `C` constraint definiciójában, akkor ha az `R`-beli változóknak konkrét értéket adunk, akkor az `R` tartománykifejezés értéke **pontosan** a `C` constraintet kielégítő `X` értékek halmaza kell hogy legyen.

3.8.1 Indexikálisok szintaxisa

```
Indexical --> X in Range

X --> variable                  { constraint-változó }

Constant --> integer
  |   inf   |   sup             { minusz/plusz végtelen }
  |   X                         { csak behelyettesítve értékelhető }
                                { ki, amíg nincs behelyettesítve, }
                                { felfüggesztődik, ez az ún.  }
                                { szabad változó }
Term --> Constant
  |   card(X)                   { X tartományának mérete }
  |   min(X)                    { X alsó határa }
  |   max(X)                    { X felső határa }
  |   Term + Term | Term - Term
  |   Term * Term | Term mod Term
  |   - Term
  |   Term /> Term              { felfelé kerekített osztás }
  |   Term /< Term              { lefelé kerekített osztás }

ConstantSet -->  {Term,...,Term}
```

```
Range --> ConstantSet
  |   dom(X)                          { X tartománya }
  |   Term..Term                      { intervallum }
  |   Range/\Range                    { metszet }
  |   Range\/Range                    { únió }
  |   \Range                          { komplementer halmaz }
  |   - Range                         { pontonkénti negáció }
  |   Range + Range | Range + Term    { pontonkénti összszeadás}
  |   Range - Range | Range - Term
  |   Term - Range                    { pontonkénti kivonás }
  |   Range mod Range | Range mod Term { pontonkénti modulo }
  |   Range ? Range                   { feltételes kifejezés }
  |   unionof(X,Range,Range)          { únió-kifejezés }
  |   switch(Term,MapList)            { kapcsoló-kifejezés }
```

3.8.2 Szűkítő indexikálisok kezelése

Tartománykifejezések monotonitása

Jelöljük $D(X,S)$-sel az X változó tartományát az S tárban, $S(R)$-rel egy R tartománykifejezés értékét az S tárban.
Legyen egy S tár pontosítása (kiterjesztése) S', ha minden X változóra $D(X,S') \subseteq D(X,S)$.

- Egy R tartománykifejezés egy S tárra nézve monoton, ha S minden S' pontosítására $S'(R) \subseteq S(R)$.
- R S-re nézve antimonoton, ha S minden S' pontosítására $S'(R) \supseteq S(R)$.
- R S-ben konstans, ha monoton és antimonoton (nem változik).
- egy indexikális (anti)monoton, ha a benne levő tartománykifejezés (anti)monoton.

Vegyük észre, hogy egy tartománykifejezés monoton és antimonoton is, ha a benne lévő változók be vannak helyettesítve, így ha egy indexikálisnak a kiértékeléshez (anti)monotonnak kell lennie, akkor az legkésőbb a benne levő változók behelyettesítése után kiértékelhető.

Tétel: ha egy `X in R` indexikális monoton egy S tárban, akkor az `X` értéktartománya S(`R`)-rel szűkíthető.

Az `X in R` indexikális feldolgozása

Az indexikálist felfüggesztjük amíg kiértékelhető és monoton nem lesz. Ezek a feltételek mindenképpen teljesülnek , ha a tartománykifejezésben levő változók behelyettesítődnek. Ezután az `R`-beli nem behelyettesített változókra megállapítjuk az ébresztési feltételeket. Ha az Y változó fordul elő

- `dom(Y), card(Y)` környezetben, akkor tartománya bármilyen változásakor;
- `min(Y)` környezetben, akkor alsó határ változásakor;
- `max(Y)` környezetben, akkor felső határ változásakor kell felébreszteni.

Ha az indexikális felébred, a következőket csinálja:

1. Ha $D(X,S)$ és $S(R)$ diszjunktak, akkor meghiúsul, azaz visszalépést vált ki.
2. A tárat az X `in` $S(R)$ constrainttel **szűkíti**, azaz $D(X,S) := D(X,S) \wedge S(R)$
3. Ha $S(R)$ konstans, az **indexikálist tartalmazó constraint** levezethetővé vált - befejeződik az indexikális végrehajtása.
4. Egyébként újra elalszik.

3.8.3 Példák szűkítő indexikálisokra

```
% clpfd könyvtárbeli elemi constraint
't+u=c'(T,U,C) +:                    % Tartomány-konzisztens!
        T in C - dom(U),       U in C - dom(T).

% clpfd könyvtárbeli elemi constraint
't+u>=c'(T,U,C) +:
        T in C - max(U)..sup,  U in C - max(T)..sup.

% clpfd könyvtárbeli elemi constraint
'ax+c=t'(A,X,C,Y) +:
        X in (min(Y) - C) /> A .. (max(Y) - C) /< A,
        Y in  min(X)*A + C     .. max(X)*A + C.

% logikai vagy-művelet aritmetikai műveletekkel megvalósítva
'p\\/q'(P, Q, B) +:
        B in (min(P)+min(Q))/>2 .. (max(P)+max(Q))/>2,
        P in min(B)*2 - max(Q) - 1  .. max(B)*2 - min(Q),
        Q in min(B)*2 - max(P) - 1  .. max(B)*2 - min(P).
% | ?- domain([P,Q,B], 0, 1), 'p\\/q'(P, Q, B), P=1.
%                            B = 1, P = 1, Q in 0..1 ?

% intervallum-konzisztens 'különbségük-nagyobb-mint' constraint
'abs(x-y)>c'(X, Y, C) +:
        X in (inf .. max(Y)-C-1) \/ (min(Y)+C+1 .. sup),
        Y in (inf .. max(X)-C-1) \/ (min(X)+C+1 .. sup).
% | ?- 'abs(x-y)>c'(X,Y,3), X in 3..5.
%                            X in 3..5, Y in(inf..1)\/(7..sup) ?

% az ismertetett N királynő feladat alapconstraintje indexikálisokkal
no_threat_3(X, Y, I) +:
        X in \({Y} \/ {Y+I} \/ {Y-I}),
        Y in \({X} \/ {X+I} \/ {X-I}).

% Hibás constraint, sérti az alapszabályt!
'x=<y=<z'(X, Y, Z) +:
        Y in min(X)..max(Z),
        Z in min(Y).. sup,
        X in inf..max(Y).
% | ?- 'x=<y=<z'(15, 5, Z).
%                            Z in 5..sup ?
```

Az utolsó, hibás constraintet érdemes tanulmányozni. Fontos megjegyezni, hogy **minden** indexikális felelős az **egész** constraintért! Ezért minden indexikálisban szerepelnie kell valamennyi, a constraintben találálható változónak, **és az alapszabályt be kell tartani**. A példában a constraint megoldó a `Z in min(Y)..sup` indexikálist választja. Az az alapszabály megsértésének következtében konstans, ezért a constraintet a rendszer igaznak és teljesültnek nyilvánítja - hibásan.

3.8.4 A bonyolultabb tartománykifejezések

Únió-kifejezés

`unionof(X, H, T)` — ahol `X` változó, `H` és `T` tartománykifejezések. Kiértékelése egy S tárban: kiértékeljük `H`-t, legyen ennek értéke $\{x_1, \ldots, x_n\}$. Képezzük a T_i kifejezéseket úgy, hogy `T`-ben `X` helyébe x_i-t írjuk. Ekkor az únió-kifejezés értéke a $S(T_1), \ldots, S(T_n)$ halmazok úniója. Ha $S(H)$ végtelen, felfüggesztjük. Példa:

```
% Maximálisan szűkítő, de hatékonytalan!
```

```
no_threat_4(X, Y, I) +:
        X in unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
        Y in unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
```

Feltételes kifejezés
`Felt ? Tart` – ahol `Felt` és `Tart` tartománykifejezések. Ha S(`Felt`) üres halmaz, akkor a feltételes kifejezés értéke is üres halmaz, egyébként pedig azonos S(`Tart`) értékével. Példa:

```
'x=<y=<z'(X, Y, Z) +:       % Ez már helyes!
        Y in min(X)..max(Z),
        Z in ((inf..max(Y)) /\ dom(X)) ? (min(Y).. sup),
        X in ((min(Y)..sup) /\ dom(Z)) ? (inf..max(Y)).
```

Használható a kiértékelés késleltetésére: `( Felt?(inf..sup) \/ Tart )` eredménye S(`Tart`), ha S(`Felt`) üres, egyébként `inf..sup`. Az ilyen szerkezetekben `Tart` értékét ki sem számoljuk, amíg `Felt` nem üres. Példa:

```
% Maximálisan szűkít, kicsit kevésbé hatékonytalan
no_threat_5(X, Y, I) +:
    X in (4..card(Y)) ? (inf..sup) \/
          unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
    Y in (4..card(X)) ? (inf..sup) \/
          unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
```

Kapcsoló-kifejezés
`switch(T, MapList)` — ahol `T` a `term` szintaxisának megfelelő kifejezés, `MapList` pedig `integer-Range` alakú párokból álló lista. Ha S(`T`) értéke Key és `MapList` tartalmaz egy $Key - Expr$ párt, akkor a kapcsoló értéke $S(Expr)$, egyébként üres. Például az alábbi két definíció hatása azonos:

```
p(X, Y) :- relation(X, [1-{1},2-{1,2},3-{1,2,3}], Y).

q(X, Y) +:
  X in unionof(B,dom(Y),switch(B,[1-{1,2,3},2-{2,3},3-{3}])),
  Y in unionof(B,dom(X),switch(B,[1-{1},2-{1,2},3-{1,2,3}])).
```

3.8.5 Reifikálható constraintek definiálása

Reifikálható constraintek definiálásához négy klózból állo FD-predikátum szükséges. Ezeket a négy különböző nyakjellel lehet definiálni. A végrehajtási mechanizmus:

- a `+?` nyakjelű klóz figyeli, hogy a constraint a tárból levezethetővé vált-e;
- a `-?` nyakjelű klóz figyeli, hogy a constraint negáltja a levezethetővé vált-e;
- egy külön démon figyeli, hogy, hogy a constraint igazságértékét tartalmazó tükrözött változó értéket kapott-e, ha igen, és értéke 1, akkor a `+:` nyakjelű klóz lép érvénybe, ha 0, akkor a `-:` nyakjelű.

A kérdező klózok csak egyetlen indexikálist tartalmazhatnak.

Nézzünk egy példát, a `'x\=y'(X,Y)` elemi constraint megvalósítását:

```
'x\\=y'(X,Y) +:        % 1. a constraintet szükitő indexikálisok
        X in \{Y},
        Y in \{X}.
'x\\=y'(X,Y) -:        % 2. a negáltját szükitő indexikálisok
        X in dom(Y),
        Y in dom(X).
'x\\=y'(X,Y) +?        % 3. a levezethetőséget kérdező
        X in \dom(Y).  %                            indexikális
'x\\=y'(X,Y) -?        % 4. a negált levezetheto4ségét kérdező
        X in {Y}.      %                            indexikális
```


Kérdező indexikálisok feldolgozása
Itt a 3.8.2. pont jelöléseit fogjuk használni. Az indexikálist felfüggesztjük amíg kiértékelhető és antimonoton nem lesz. Az ébresztési feltételek, feltéve, hogy az indexikális `X in R` alakú és `Y` az `R`-ben előforduló változó:

- `X` tartományának bármilyen változáskor
- `dom(Y)`, `card(Y)` környezetben — bármilyen változáskor
- `min(Y)` környezetben – alsó határ változásakor
- `max(Y)` környezetben – felső határ változásakor

Ha az indexikális felébred:

1. Ha $D(X, S) \subseteq S(R)$ akkor a constraint levezethetővé vált.
2. Egyébként, ha $D(X, S)$ és $S(R)$ diszjunktak, valamint $S(R)$ konstans, akkor a constraint negáltja levezethetővé vált.
3. Egyébként újra elaltatjuk az indexikálist.

Példa

```
'x=<y'(X,Y) +:
        X in inf..max(Y),
        Y in min(X)..sup.
'x=<y'(X,Y) -:
        X in (min(Y)+1)..sup,
        Y in inf..(max(X)-1).
'x=<y'(X,Y) +?
        X in inf..min(Y).
'x=<y'(X,Y) -?
        X in (max(Y)+1)..sup.
```

3.8.6 FD-predikátumok, indexikálisok összefoglalása

Legyen `C(Y1, ...)` egy FD-predikátum, amelyben szerepel egy

`Yi in R(Y1, ...)`

indexikális. A baloldali `Yi` változót néha `X`-nek nevezzük. Az `R` tartománykifejezéshez tartozó halmazfüggvényt nevezzük R-nek.

- **Definició:** Az indexikális az alábbi relációt definiálja:

  $$C = \{ < y_1, \ldots > \mid y_k \in \texttt{inf..sup} \wedge y_i \in R(\{y_1\}, \ldots)\}$$

- **Alapszabály:** Egy FD-predikátum csak akkor értelmes, ha a pozitív klózaiban levő összes indexikális ugyanazt a relációt definiálja; továbbá a negatív klózokban levő összes indexikális ennek a relációnak a negáltját (komplemensét) definiálja.
- Ha R **monoton** egy S tárra nézve, akkor $S(R)$-ről belátható, hogy minden olyan x értéket tartalmaz, amelyek (az S által megengedett y_j értékekkel együtt) a C relációt kielégítik. Ezért szűkítő indexikálisok esetén jogos az X tartományát $S(R)$-rel szűkíteni.
- Ha R **antimonoton** egy S tárra nézve, akkor $S(R)$-ről belátható, hogy minden olyan x értéket kizár, amelyekre (az S által megengedett y_j értékekkel együtt) a C reláció nem áll fenn. Ezért kérdező indexikálisok esetén, ha $D(X, S) \subseteq S(R)$, jogos a constraintet az S tárból levezethetőnek tekinteni.
- A fentiek miatt természetesen adódik az indexikálisok felfüggesztési szabálya: a szűkítő indexikálisok végrehajtását mindaddig felfüggesztjük, amíg monotonná nem válnak; a kérdező indexikálisok végrehajtását mindaddig felfüggesztjük, amíg antimonotonná nem válnak.

- **Az indexikálisok deklaratív volta:** Ha a fenti alapszabályt betartjuk, akkor a `clpfd` megvalósítás az FD-predikátumot helyesen valósítja meg, azaz mire a változók teljesen behelyettesítetté válnak az FD-predikátum akkor és csak akkor for sikeresen lefutni, vagy 1-gyé tükröződni (reifikálódni), ha a változók értékei a predikátum által definiált relációhoz tartoznak. Az indexikális megfogalmazásán csak az múlik, hogy a nem-konstans tárak esetén milyen jó lesz a szűkítő ill. kérdező viselkedése.

3.8.7 Constraintek automatikus fordítása indexikálisokká

A SICStus Prologban lehetőség van bizonyos speciális constraintek indexikálisokká törénő fordítására. A lefordítandó constraint lehet:

- lineáris aritmetikai kifejezés;
- `relation/3` és `element/3` szimbólikus constraint.

Az indexikálissá fordítandó constraint formája:

Head `+:` *Constraint*.

Ezek a constraintek nem reifikálhatóak, így csak a `+:` nyakjel használható. Adódik a kérdés, hogy mi a különbség az indexikálissá fordított és a "normális" constraintek között. A nem indexikálissá fordított constraintek elemi constraintekké, vagyis `clpfd` könyvtárbeli hívásokká fordulnak, míg az indexikálissá fordítottak egyetlen indexikálissá. Az első változat kevesebb helyet foglal el, a második gyorsabb.

- Az aritmetikai kifejezésekből fordított indexikális mérete négyzetesen nő a forráskód méretével.
- A `relation/3` és `element/3` szimbólikus constraintek lineáris méretű unió- és kapcsoló-kifejezésekké fordulnak, ezért vigyázni kell a nagyméretű kezdő-tartományokkal.

A `SEND+MORE=MONEY` feladvány constraintje indexikálissá fordítva kb. 7-szer nagyobb helyigényű, 1.5-szer gyorsabb, mint a `:-`-os változat:

```
sum2(S,E,N,D,M,O,R,Y) +:
          1000*S+100*E+10*N+D
 +        1000*M+100*O+10*R+E
 #=       10000*M+1000*O+100*N+10*E+Y.
```

Vegyük példaként az $X + Y < U + V$ constraintet:

- `p(X,Y,U,V) :- X+Y #< U+V.` törzse `clpfd` könyvtári hívásokra fordul:
  `clpfd:('x+y=t'(X,Y,_A),'x+y=t'(U,V,_B),'t>=u+c'(_B,_A,1))`
- `p(X,Y,U,V) +: X+Y #< U+V.` egyetlen FD predikátummá válik: `p(X,Y,U,V) +: X in` R`(Y,U,V), Y in ..., U in ..., V in ....`

Megjegyzés A #6-os SICStus változatban egy gyors `scalar_product` eljárás van, ennél viszont a fenti példa 2-szer lassabb!

3.9 Újdonságok a következő SICStus-ban (#6)

3.9.1 Új és bővült eljárások

- `scalar_product(Coeffs, Xs, Relop, Value)`: Igaz, ha `Coeffs * Xs Relop Value`, ahol `Relop` aritmetikai összehasonlító operátor, `*` skalárszorzást jelöl, `Coeffs` egészekből álló lista.
- `sum(Xs, Relop, Value)`: Σ `Xs Relop Value`. Ekvivalens a `scalar_product([1,1,...,1], Xs, Relop, Value)` hívással.
- `count(Val, List, Relop, Count)`: Igaz, ha a `Val` egész szám `Lista`-ban való előfordulásainak száma n, és n `Relop Count`.
- `element(N, List, X)`: `List`-ben lehetnek constraint-változók.

- `permutation(Xs, Ys, Ctrl)`: `Xs` és `Ys` n hosszúságú listák. Igaz ha `X[i]` és `Y[i]` mind az `1..`n tartományba esnek, és `X[i]=j` $\Leftrightarrow$ `Y[j]=i`.
- `circuit(Xs)`: `Xs` n hosszúságú lista. Igaz ha minden `X[i]` az `1..`n tartományba esik és `X[1]`, `X[X[1]]`, `...` az `1..`n egy permutációja.
- `cumulative(Starts, Durations, Resources, Limit)`: sokkal jobb a szűkítés, mint a #5-ös változatban.
- `serialized(Starts, Durations)`: a `cumulative` speciális esete, ahol az összes erőforrás-igény és a korlát is 1.

3.9.2 FD-halmazok

Ez az az absztrakt adattípus, amivel SICStus Prolog a változók értéktartományát ábrázolja. Az `{1} \/ 5..10` tartomány FD-halmazokkal történő belső ábrázolása `[[1|1],[5|7]]`, de ezt nem szabad közvetlenül használni: az absztrakt adattípus erre saját műveleteket biztosít.

- `fd_set(X, Set)`: `X` tartománya a `Set` FD-halmaz (a #5-ös változatban ez az `fd_dom` eljárás volt).
- `X in_set Set`: `X`-et `Set`-re szűkíti.
- Az FD-halmaz: diszjunkt intervallumok rendezett listája. Mint absztrakt adattípus, alapműveletei:
  - `empty_fdset(S)`: `S` az üres FD-halmaz.
  - `fdset_parts(S, Min, Max, Rest)`: `S` első intervalluma `Min..Max`, a maradék FD-halmaz: `Rest`. FD-halmaz építésére is jó.
  - `is_fdset(S)`: `S` egy korrekt FD-halmaz.
- További, kényelmi szolgáltatások, pl.
  - `fdset_singleton(Set, Elt)`: `Set` az egyetlen `Elt`-ből áll.
  - `fdset_union/3`, `fdset_intersection/3`, `fdset_complement/2`.
  - `fdset_member(Elt, Set)`: `Elt` eleme a `Set` FD-halmaznak.

3.9.3 Globális constraintek

- `fd_global(Constraint, State, Susp)`: `Constraint` végrehajtásának elindítása, `State` kezdőállapottal, `Susp` ébresztési listával. Az ébresztési lista elemei és jelentésük:
  - `dom(X)`: az X változó tartományának bármilyen változásakor;
  - `min(X)`: az X változó tartományának minimumának megváltozásakor;
  - `max(X)`: az X változó tartományának maximumának megváltozásakor;
  - `minmax(X)`: az X változó tartományának bármely szélső értékének megváltozásakor;
  - `val(X)`): az X változó behelyettesítésekor a constraint felébred.
- `clpfd:dispatch_global(Constraint, State0, State, Actions)`: A kampó-eljárás törzse definiálja a `Constraint` felébredésekor elvégzendőket: `State0` a régi, `State` az új állapot. Az `Actions` lista elemei lehetnek: `exit`, `fail`, `X=V`, `X in R`, `X in_set S`, `call(Module:Goal)`. Nagyon fontos, hogy minden, FD változókra vonatkozó szűkítést az `Action` listába rakott megfelelő atomokkal csináljunk, mert globális constraint belsejében *tilos szűkíteni*!

3.9.4 Példa: `exactly/3` (korábbi `pontosan/3`) globális constraintekkel

```
% An implementation of exactly(I, [X1,...Xm], N). Rewrite rules:

% [1] |= Xi=I  ==> exactly(I, [X1,...Xi-1,Xi+1,...Xm], N-1)
% [2] |= Xi\=I ==> exactly(I, [X1,...Xi-1,Xi+1,...Xm], N)
% [3] |= N=0   ==> X1\=I, ... Xm\=I
% [4] |= N=m   ==> X1=I,  ... Xm=I
%                        Necessary condition: 0 =< N =< m.

% the entrypoint
exactly(I, Xs, N) :-
    dom_susps(Xs, Susp),
    fd_global(exactly(I,Xs,N), state(Xs,N), Susp).

dom_susps([], []).
dom_susps([X|Xs], [dom(X)|Susp]) :- dom_susps(Xs, Susp).

% the solver method
:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly(I,_,_),
           state(Xs0,N0), state(Xs,N), Actions) :-
    ex_filter(Xs0, Xs, N0, N, I), length(Xs, M),
    (   N=:=0 -> Actions = [exit|Ps], ex_neq(Xs, I, Ps)
    ;   N=:=M -> Actions = [exit|Ps], ex_eq(Xs, I, Ps)
    ;   N>0, N<M -> Actions = []
    ;   Actions = [fail]
    ).

% rules [1,2]: filter the X's, decrementing N
ex_filter([], [], N, N, _).
ex_filter([X|Xs], Ys, L, N, I) :-
    X==I, !, M is L-1, ex_filter(Xs, Ys, M, N, I).
ex_filter([X|Xs], Ys0, L, N, I) :-
    fd_set(X, Set), fdset_member(I, Set), !,
    Ys0 = [X|Ys], ex_filter(Xs, Ys, L, N, I).
ex_filter([_|Xs], Ys, L, N, I) :- ex_filter(Xs, Ys, L, N, I).

% rule [3]: all must be neq I
ex_neq(Xs, I, Ps) :- fdset_singleton(Set0, I),
    fdset_complement(Set0, Set), eq_all(Xs, Set, Ps).

% rule [4]: all must be eq I
ex_eq(Xs, I, Ps) :-
    fdset_singleton(Set, I), eq_all(Xs, Set, Ps).

eq_all([], _, []).
eq_all([X|Xs], Set, [X in_set Set|Ps]) :- eq_all(Xs, Set, Ps).

% sample queries:
| ?- exactly(5,[A,B,C],1), A=5.
     A = 5, B in(inf..4)\/(6..sup), C in(inf..4)\/(6..sup)
| ?- exactly(5,[A,B,C],1), A in 1..2, B in 3..4.
     C = 5, A in 1..2, B in 3..4
```


3.10 Példa: Négyzetdarabolás

Ebben a fejezetben egy feladatra, a négyzetdarabolásra adunk különböző `clpfd` alapú megoldásokat. Érdemes tanulmányozni őket, mert nagyon jól látható rajtuk, hogy nem elég csak használni a constraint alapú programozást, de a hatékonysághoz jól is kell használni. A fejezet végén a futási idő adatokból mindez jól látható lesz.

A feladat

Adott egy nagy négyzet oldalhosszúsága (`N`), és adottak kis négyzetek oldalhosszúságai egy listában (`L`), úgy, hogy területösszegük megegyezzen a nagy négyzet területével. A feladat a kis négyzetekkel pontosan lefedni a nagyot, és visszaadni a kis négyzetek koordinátáit.

Példa:

- `N=10`
- `L=[6,4,4,4,2,2,2,2]`

ekkor egy megoldás:

- `Xs = [1,7,7,1,5,5,7,9]`
- `Ys = [1,1,5,7,7,9,9,9]`
- `Ss = [6,4,4,4,2,2,2,2]`

Próba-adatok a programok kipróbálásához

- `10, [6,4,4,4,2,2,2,2],`
- `20, [9,8,8,7,5,4,4,4,4,4,3,3,3,2,2,1,1],`
- `112, [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2],`
- `175, [81,64,56,55,51,43,39,38,35,33,31,30,29,20,18,16,14,9,8,5,4,3,2,1].`

3.10.1 Négyzetdarabolás: Prolog megoldás

Ebben a részben egy, az egyszerű Prolog visszalépéses keresésen alapuló programot adunk meg.

```
:- use_module(library(lists)).

squares_1(Ss, Size, SXYs) :-
        MSize is -Size,
        squares(Ss, [MSize/Size,Size/0], Size, SXYs).

squares([], _, _, [])  :- !.
squares(Ss0, L0, Size, SXYs) :-
        L0=[V0/_|_], Y0 is Size+1+V0,
        hole(L0, 1-Y0, XY, L1, L1T, L2),
        fill_hole(L2, L1T, XY, Size, Ss0, Ss, SXYs1, SXYs),
        squares(Ss, L1, Size, SXYs1).

fill_hole([V0/0,V/H|L], [V1/H|L], _XY, _Size,
                      Ss0, Ss0, SXYs0, SXYs0) :-
        !, V1 is  V0+V.
fill_hole([V/H|L0], [V1/S|L1], X-Y, Size,  Ss0, Ss,
                      SXYs0, [s(S,X,Y)|SXYs]) :-
        MaxS is min(H, Size+1-Y),
        select(S, Ss0, Ss1), S =< MaxS,
        V1 is V+S, MS is -S, H1 is H-S, X1 is X+S,
        fill_hole([MS/H1|L0], L1, X1-Y, Size, Ss1, Ss, SXYs0, SXYs).
```

```
hole([V1/0,V2/H|L], XY0, XY, L1, L1T, L2) :-
        !, V is V1+V2,
        hole([V/H|L], XY0, XY, L1, L1T, L2).
hole([V/H1,0/H2|L], XY0, XY, L1, L1T, L2) :-
        !, H is H1+H2,
        hole([V/H|L], XY0, XY, L1, L1T, L2).
hole([V1/H1|L], XY, XY, L1, L1, [V1/H1|L]) :-
        H1 > 0, V1 < 0, L = [V2/_|_], V2>0, !.
hole([VH1,VH2|L], X0-Y0, XY, [VH1|L1], L1T, L2) :-
        VH1=_/H1, VH2=V2/_, X1 is X0+H1, Y1 is Y0+V2,
        hole([VH2|L], X1-Y1, XY, L1, L1T, L2).
```

3.10.2 Négyzetdarabolás: naív clpfd megoldás

Ez a megoldás veszi a kis négyzetek összes koordinátáját, és beállítja őket úgy, hogy értéküket a nagy négyzeten belül vegyék fel. Ezután minden kisnégyzet-párra felveszi a `no_overlap/6` constraintet, ami azt írja le Prolog választási pontok segítségével, hogy a két négyzet nem fedi egymást. Ezután címkézéssel megadja az eredményt.

```
:- use_module(library(clpfd)).

squares_2(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        labeling([], Xs), labeling([], Ys).

generate_coordinates([], [], [], _).
generate_coordinates([X|Xs], [Y|Ys], [S|Ss], Size) :-
        Sd is Size-S+1,
        domain([X,Y], 1, Sd),
        generate_coordinates(Xs, Ys, Ss, Size).

state_no_overlap([], [], []).
state_no_overlap([X|Xs], [Y|Ys], [S|Ss]) :-
        state_no_overlap(X, Y, S, Xs, Ys, Ss),
        state_no_overlap(Xs, Ys, Ss).

state_no_overlap(_, _, _, [], [], []).
state_no_overlap(X, Y, S, [X1|Xs], [Y1|Ys], [S1|Ss]) :-
        no_overlap(X, Y, S, X1, Y1, S1),
        state_no_overlap(X, Y, S, Xs, Ys, Ss).

no_overlap(X1, _Y1, S1, X2, _Y2, _S2) :-
        leqc(X1, S1, X2).
no_overlap(X1, _Y1, _S1, X2, _Y2, S2) :-
        leqc(X2, S2, X1).
no_overlap(_X1, Y1, S1, _X2, Y2, _S2) :-
        leqc(Y1, S1, Y2).
no_overlap(_X1, Y1, _S1, _X2, Y2, S2) :-
        leqc(Y2, S2, Y1).

leqc(X1, S1, X2) +: X1+S1 #=< X2.
```

3.10.3 Négyzetdarabolás: diszjunktív constraintek

Az előző megoldás gyenge pontját a diszjunkció Prolog visszalépésen alapuló megadása jelenti. Általában is sokszor előfordul a diszjunkció igénye constraint feladatoknál, ezért érdemes külön

figyelmet szentelni neki.

A diszjunkció lehetséges megvalósításai

- spekulatív: Prolog választási pontokkal (lásd `squares_2`). Hátránya maga a próbálgatás, ami a kombinatórikus robbanás miatt használhatatlanná teheti ezt a módszert.
- reifikáció(számosság)-alapú: a `#∨` müvelettel, pl:
  `X1+5 #=< X2 #∨ X2+5 #=< X1.`
  Ennek hátránya az, hogy a constraint-megoldó csak akkor tud valamit tenni, ha valamelyik constraint levezethetősége eldönthető.
- (gyenge) konstruktív diszjunkció, pl:
  `| ?- domain([X1,X2], 0, 6), X1+5 #=< X2 ∨ X2+5 #=< X1.`
  `X1 in (0..1)∨(5..6), X2 in (0..1)∨(5..6)`
  Ez adott két constraintre maximálisan szűkít, de ha más constraint is közbeszólhat, akkor már nem.
- teljes konstruktív diszjunkció:
  `| ?- domain([X,Y], 0, 20), X+Y #= 20, X #=<5 ∨ Y #=< 5.`
  `X in (0..5)∨(15..20), Y in (0..5)∨(15..20)`
  Ez tartomány konzisztenciát biztosít, de nagyon drága művelet, ezért általában nem is használják.

Itt megadjuk, hogy az előző program `no_overlap/6` predikátumát mire kell lecserélni, hogy számosság-alapú, illetve gyenge konstruktív diszjunkciót megvalósító megoldást kapjunk.

Számosság-alapú `no_overlap`

```
no_overlap_3(X1, Y1, S1, X2, Y2, S2) :-
        X1+S1 #=< X2 #<=> B1,
        X2+S2 #=< X1 #<=> B2,
        Y1+S1 #=< Y2 #<=> B3,
        Y2+S2 #=< Y1 #<=> B4,
        B1+B2+B3+B4 #>= 1.
```

Gyenge konstruktív diszjunkción alapuló `no_overlap`

```
no_overlap_4(X1, Y1, S1, X2, Y2, S2) :-
  call( abs(2*(X1-X2)+(S1-S2)) #>= S1+S2 #\/
        abs(2*(Y1-Y2)+(S1-S2)) #>= S1+S2 ).
```

Gyenge konstruktív diszjunkción alapuló `no_overlap` indexikálisokkal

```
no_overlap_5(X1, Y1, S1, X2, Y2, S2) +:
  X1 in ((((min(Y1)+S1)..max(Y2))\/((min(Y2)+S2)..max(Y1))) ? (inf..sup))
        \/ ((inf..max(X2)-S1) \/ (min(X2)+S2..sup)),
  X2 in ((((min(Y1)+S1)..max(Y2))\/((min(Y2)+S2)..max(Y1))) ? (inf..sup))
        \/ ((inf..max(X1)-S2) \/ (min(X1)+S1..sup)),
  Y1 in ((((min(X1)+S1)..max(X2))\/((min(X2)+S2)..max(X1))) ? (inf..sup))
        \/ ((inf..max(Y2)-S1) \/ (min(Y2)+S2..sup)),
  Y2 in ((((min(X1)+S1)..max(X2))\/((min(X2)+S2)..max(X1))) ? (inf..sup))
        \/ ((inf..max(Y1)-S2) \/ (min(Y1)+S1..sup)).
```

3.10.4 Négyzetdarabolás: redundáns constraintek

Általános módszer a CLP(FD) világban, hogy a szükséges constraintek mellé még felveszünk olyan constrainteket, amelyek az előzőekből következnek ugyan, de a keresési teret szűkítik, emiatt pedig a program gyorsabbá valik. Ez esetben a kapacitás-constrainteket vesszük fel. Ezek kikötik, hogy minden sorban és minden oszlopban a metszett négyzetek oldalhossz-összege a nagy négyzet oldalával megegyezik.

```
squares_6(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        state_capacity(Xs, Sizes, Size),
        state_capacity(Ys, Sizes, Size),
        labeling([], Xs), labeling([], Ys).

state_capacity(Cs, Sizes, Size) :-
        state_capacity(1, Size, Cs, Sizes).

state_capacity(Pos, Size, _, _) :-
        Pos>Size, !, true.
state_capacity(Pos, Size, Cs, Sizes) :-
        accumulate(Cs, Sizes, Pos, Size),
        Pos1 is Pos+1,
        state_capacity(Pos1, Size, Cs, Sizes).

accumulate([], [], _, 0).
accumulate([C|Cs], [S|Ss], Pos, Sum) :-
        Crutch is Pos-S+1,
        C in Crutch .. Pos #<=> B,
        B*S+Sum1 #= Sum,
        accumulate(Cs, Ss, Pos, Sum1).
```

3.10.5 Négyzetdarabolás: speciális cimkézés

Eddig a címkézés a rendszer `labeling/2` eljárásával ment. Most speciális címkézést alkalmazunk, aminek lényege az, hogy a nagy négyzetet alulról felfelé, azon belül pedig balról jobbra próbáljuk kitölteni a kis négyzetekkel. Így hamarabb észrevesszük, hogy az aktuális elrendezés nem vezet eredményre.
A módszer:

1. keressük meg a legkisebb alsó határt,
2. sorra próbáljuk a változókat a legkisebb határ-értékkel behelyettesíteni,
3. ha egy változóra ez nem sikerül, zárjuk ki az alsó határt a tartományából.

```
squares_7(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        state_capacity(Xs, Sizes, Size),
        state_capacity(Ys, Sizes, Size),
        dual_labeling(Xs, 1, Size),
        dual_labeling(Ys, 1, Size).

dual_labeling([], _, _) :- !.
dual_labeling(L, I, Limit) :-
        dual_labeling(L, L1, I, Limit, J),
        dual_labeling(L1, J, Limit).

dual_labeling([], [], _, J, J).
dual_labeling([X|L1], L2, I, J0, J) :-
        (   integer(X) -> dual_labeling(L1, L2, I, J0, J)
        ;   X = I, dual_labeling(L1, L2, I, J0, J)
        ;   X #> I,
            fd_min(X, J1), J2 is min(J0,J1), L2 = [X|L3],
            dual_labeling(L1, L3, I, J2, J)
        ).
```


3.10.6 A négyzetdarabolás megoldásainak összehasonlítása

A négyzetdaraboló programokat maximum 500 másodpercig futtattuk, ha nem futott le ennyi idő alatt, akkor az alábbi táblázatokban egy - jel szerepel. A variáns a programok alaptulajdonságainak a kombinálásával kapott programot jelenti, a 7+6+3 például azt, hogy a variáns speciális címkézést használ, redundáns constraintjei vannak és a `no_overlap` constraintje számosság alapú. A méret a nagy négyzet mérete.

Futási idő (másodperc)

variáns/méret	10	20	112	175
1	0.00	6.29	152.63	-
2	5.68	-	-	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-
6+2	5.68	-	-	-
6+3	0.44	2.97	-	-
6+4	0.56	3.74	-	-
6+5	0.30	1.93	-	-
7+2	5.59	-	-	-
7+6+2	5.83	-	-	-
7+6+3	0.44	2.22	69.95	125.79
7+6+4	0.55	2.75	71.08	123.75
7+6+5	0.32	1.50	63.29	115.83

Visszalépések száma

variáns/méret	10	20	112	175
1	0	1678	3398379	-
2	46954	-	-	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-
6+2	46954	-	-	-
6+3	0	18	-	-
6+4	0	18	-	-
6+5	0	18	-	-
7+2	46954	-	-	-
7+6+2	46954	-	-	-
7+6+3	0	0	150	105
7+6+4	0	0	150	105
7+6+5	0	0	150	105

Vegyük észre, hogy a tiszta Prolog megoldás azért ilyen jó, mert okosan helyezi el a kis négyzeteket: hasonlóan, mint a speciális címkézésű clp(FD) variánsok. A "bután" címkéző egyszerű clp(FD) variánsok még a 10-es nagyságú négyzetet sem rakják ki a megadott időn belül.