

# A korlátozás programozás alapjai

Kovács András  
akovacs@mit.bme.hu

## Bevezetés

Ez a segédlet a Mesterséges Intelligencia Labor c. tárgyat felvett hallgatókhoz szól, és feltételezi a logikai programozás elméleti alapjainak, a fontosabb fa-kereso algoritmusoknak és a Prolog nyelvnek az ismeretét. Célja a laboron kiadott feladatok elvégzéséhez szükséges ismeretek bemutatása és az órai munka segítése néhány kódminta által. A korlátozás programozás elméletének elsajátításához figyelmetekbe ajánljuk Szeredi Péter Nagyhatékonyságú Logikai Programozás c. tárgyát. Az itt leírt példaprogramok egy része is a tárgy jegyzetéből lett adaptálva.

## Definíciók

A *korlátozás programozás (Constraint Programming / Constraint Logic Programming, CLP)* egy deklaratív programozási paradigma kombinatorikus (optimalizálási) problémák megoldására. Hatékonyságának, rugalmasságának és az egyszerű formalizmusoknak köszönhetően számos gyakorlati alkalmazással büszkélkedhet, pl. az ütemezés, órarendkészítés, grafikai alkalmazások, stb. területén.

Röviden, egy *korlátozás-kielégítési probléma (Constraint Satisfaction Problem, CSP)* a következőkből áll:

- változók egy halmaza:  $X = \{x_1, \dots, x_n\}$ ;
- minden  $x_i$  változóhoz egy  $D_i$  véges értékészlet (*domain*);
- korlátozások (*constraint*) egy  $C$  halmaza, amely leírja, hogy a változók által felvett értékek közt milyen összefüggéseknek kell fennállnia;
- esetleg egy  $O(X)$  célfüggvény.

A *CSP* megoldása az  $X$  változók egy lekötése, melynek során minden  $x_i$   $D_i$ -ből vesz fel értéket, és kielégíti a  $C$  korlátokat. Kereshetünk

- egy megoldásra;
- az összes megoldásra, vagy
- az  $O(X)$  szerint minimális vagy maximális megoldásra.

Nem szükségképpen, de általában a  $D_i$  értékészletek a pozitív egészek halmazának részhalmazai. A továbbiakban mi is élni fogunk ezzel a feltételezéssel.

## Korlátok

Egy  $c$   $x_1, \dots, x_k$  változókra értelmezett korlát formális definíciója szerint egy  $k$  változós reláció a  $D_1 \times \dots \times D_k$  direkt-szorzat felett, amely a változók megengedett kombinációira áll fenn. Például:

<u>SICStus Prolog korlát</u>	<u>Jelentése</u>
<code>A #&gt; B</code>	$A > B$
<code>A #\= B</code>	$A \neq B$
<code>(A #= 3 #\ / B#&gt; C-2)</code>	$A = 3$ vagy $B > C - 2$
<code>all_different(L)</code>	Az $L$ lista elemei mind különbözőek.

A fenti  $k$  tetszőleges értéket felvehet, de kiemelünk két esetet:

**Unáris korlátok** ( $k = 1$ ). Egy  $c_u$  unáris korlát arra használható, hogy egy  $x_i$  változó értékészletéből eltávolítsuk a nemkívánatos értékeket. Pl. az  $x_i \neq 3$  korlát esetén a 3 értéket eltávolíthatjuk  $D_i$ -ből. Ettől a pillanattól kezdve a  $c_u$  korlát biztosan ki van elégítve, és akár el is távolítható  $C$ -ből.

**Bináris korlátok** ( $k = 2$ ). Ha egy CSP minden korlátja bináris, akkor *bináris CSP*-rol beszélhetünk. Egy bináris CSP ábrázolható egy gráffal, melynek csúcsai a változók, élei a megfelelő változó párok közt fennálló korlátok. Elméleti jelentősége van annak, hogy tetszőleges CSP átalakítható bináris CSP-vé, bár ez új változók bevezetését teheti szükségessé. Ma ezt a technikát már nem használják, hatékonyabb algoritmusok ismertek magasabb fokszámú korlátokat is tartalmazó CSP-k megoldására.

## Példa egy korlátozás programra

Az  $n$ -királyno feladatban  $n$  db királynot kell elhelyezni úgy egy  $n \times n$ -es sakktáblán, hogy semelyik ketto ne üsse egymást. Az első megvalósítás SICStus Prologban:

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

% A Qs lista N királyno biztonságos elhelyezését mutatja
% egy N*N-es sakktáblán: ha a lista i. eleme j, akkor
% az i. királynot az i. sor j. oszlopába kell helyezni.

queens(N, Qs) :-
    length(Qs, N),
    domain(Qs, 1, N),
    safe(Qs).
```

```

% safe(Qs): A Qs királyn?o-lista biztonságos.

safe([]).

safe([Q|Qs]):-
    no_attack(Qs, Q, 1),
    safe(Qs).

% no_attack(Qs, Q, I): A Qs lista által leírt királyn?ok
% egyike sem támadja a Q által leírt királyn?ot, ahol
% Qs a (j, j+1, ...) sorbeli királyn?okat írja le,
% Q a i. sorbeli királyn?ot, és I = j-i > 0.

no_attack([],_,_).

no_attack([X|Xs], Y, I):-
    no_threat(X, Y, I),
    I1 is I+1,
    no_attack(Xs, Y, I1).

% Az X és Y oszlopokban I sortávolságra lev?o
% királyn?ok nem támadják egymást, azaz nincsenek
% azonos oszlopban, / vagy \ átlóban

no_threat(X, Y, I) :-
    Y #\= X, Y #\= X-I, Y #\= X+I.

```

## Propagáció, él-konzisztencia

Ha  $x_i$  és  $x_j$  változók közt fennáll egy  $c$  bináris korlát, akkor a fenti gráf  $(x_i, x_j)$  éle *él-konzisztens* (*arc consistent*), ha minden  $v \in D_i$  értékhez létezik egy  $u \in D_j$ , hogy az  $x_i = v$ ,  $x_j = u$  változó lekötés kielégíti a  $c$  korlátot.

Minden más  $v \in D_i$  érték, azaz amelyekhez nem létezik ilyen  $u$ , eltávolítható  $D_i$ -bol, hiszen nem képezheti részét konzisztens megoldásnak. Ezen értékek eltávolítása él-konzisztenssé teszi a gráf  $(x_i, x_j)$  élet. Ezt a műveletet nevezzük a  $c$  korlát *propagálásának*. Megjegyezzük, hogy  $(x_i, x_j)$  él-konzisztenciájából nem következik  $(x_j, x_i)$  él-konzisztenciája. Ez a definíció és eljárás általánosítható magasabb fokszámú korlátok esetére is.

Ha a *CSP* minden éle él-konzisztens, akkor a magát *CSP*-t is él-konzisztensnek nevezzük. Egy *CSP* él-konzisztenssé tétele relatíve gyorsan végrehajtható, ezért a *CLP* rendszerek a feladatot a teljes megoldási folyamat során igyekeznek él-konzisztensen tartani. Ezt úgy érik el, hogy valahányszor egy  $x_i$  változó értékészlete változik, az összes  $x_i$ -t tartalmazó korlátot propagálják. Ez természetesen további inkonzisztens értékeket távolíthat el más változók értékészletéből, egy propagáció-lavinát indítva ezzel. Ha

ennek során egy változó értékkészlete üressé válik, akkor magától értetődően nincs megoldása a feladatnak.

Sajnos egy CSP él-konzisztenciája nem jelenti azt, hogy a feladatnak van megoldása. A következő példa egy ilyen esetet mutat. A korlátok *bármelyike* teljesülhet benne, a korlátok *mindegyike egyidejűleg* azonban nem. (A feladat megoldhatatlan, mert három változónak kellene felvennie az  $\{1,2\}$  kételemű halmazból páronként különböző értéket.)

$$\begin{aligned}x_1 &\in D_1 = \{1,2\}, \\x_2 &\in D_2 = \{1,2\}, \\x_3 &\in D_3 = \{1,2\}, \\x_4 &\in D_4 = \{2,3,4\}, \\ \forall i, j (i \neq j) &\Rightarrow (x_i \neq x_j)\end{aligned}$$

Láthatjuk tehát, hogy a propagáció holtpontra juthat anélkül, hogy megtalálná a feladat létező megoldását vagy bebizonyítaná annak megoldhatatlanságát. Ilyenkor a CLP rendszer keresési technikákat hív segítségül.

## Keresés

Míg a legegyszerűbb feladatokat pusztán propagációval meg lehet oldani, bonyolultabb feladatok esetén a propagációt egy fa-keresési sémába kell ágyazni, a következők szerint:

1. Inicializáljuk keresésünket egy  $n_0$  gyöker csomóponttal, amelyet úgy kaptunk, hogy a feladat leírásán lefuttattunk egy él-konzisztencia algoritmust.
2. Vegyük egy él-konzisztens, de lekötetlen változókat is tartalmazó  $n$  csomópontot. Legyenek  $c_1, \dots, c_k$  olyan korlátok, amelyekre  $\forall i \neq j (c_i \wedge c_j = \emptyset)$  és  $\bigvee_{i=1..k} c_i = I$ , azaz közülük minden változó lekötésben pontosan egy teljesül.
3. Generáljuk az  $n$  csomópont  $k$  db gyermekét rendre úgy, hogy lemásoljuk az  $n$ -ben érvényes értékkészleteket és korlátokat, majd hozzávesszük a megfelelő  $c_i$ -t. Minden gyermekben futtassunk le egy él-konzisztencia algoritmust.
  - Ha valamely  $n_i$  gyermek csomópontban egy változó értékkészlete kiürül, akkor  $n_i$  eldobható, mert belőle konzisztens megoldás nem származtatható.
  - Ha valamely  $n_i$  gyermek csomópontban minden  $x_i$  változó értékkészlete egyetlen  $v_i$  elemet tartalmaz, akkor az  $\forall j (x_j = v_j)$  változó lekötés egy konzisztens megoldás. Adjuk vissza ezt a megoldást. Ha további megoldásokra már nincs szükség, zárjuk le a keresést.
  - Az összes többi csomópont él-konzisztens, és (további) megoldások lehetőségét hordja magában, ezért illesszük be azokat a keresési fába.
4. Folytassuk a keresést a 2. pontnál.

Könnyű belátni, hogy ez az algoritmus teljes, és az általa visszaadott megoldások konzisztens megoldásai a feladatnak. Az algoritmus hatékonyságát az adja, hogy a propagátorok nagyban leszűkítik a változók értéktartományát, így az épülő keresési fa mérete sokszor több nagyságrenddel kisebb lesz, mint egyszerű visszalépéses keresés esetén, amit pl. a standard Prolog alkalmaz.

A gyakorlatban szokás a 2. pontban szereplő  $c_i$ -ket rendre  $x_q = v_i$ -nek választani, ahol  $x_q = v_i$  egy valamilyen heurisztika szerint választott lekötetlen változó, a  $x_q = v_i$ -k pedig  $x_q = v_i$  elemei. A Prologos nevezéktanban az ilyen elágazási stratégiát használó keresést szokás *címkezésnek*, *labelingnek* nevezni. Míg ha a SICStusban egy korlátozás programot *labeling nélkül* indítunk el, akkor csak az él-konzisztencia algoritmust futtatja le rajta, *labelinggel* az első konzisztens megoldásig keres. További megoldásokat a ; jel bebillentyűzésével kérhetünk.

```
% Labeling nélkül elakad a megoldás megtalálása előtt.
| ?- queens(4, Qs).
Qs = [_A,_B,_C,_D],
_A in 1..4,
_B in 1..4,
_C in 1..4,
_D in 1..4 ?
yes
| ?-

% Labelinggel megtaláljuk az összes helyes megoldást.
| ?- queens(4, Qs), labeling([], Qs).
Qs = [2,4,1,3] ? ;
Qs = [3,1,4,2] ? ;
no
| ?-
```

Érdeemes megjegyezni, hogy míg a korlátozás propagációs algoritmusok általában gyorsak (polinom időben lefutnak), a keresés során akár  $d^n$  méretű fa építésére lehet szükség, ahol  $d$  az értékészletek méretét,  $n$  a változók számát jelöli. Mivel komoly következményei lehetnek a számítási időre nézve, nagyméretű feladatok megoldása során érdemes mindent elkövetni a keresési fa méretének csökkentéséért, az alábbi módszerek valamelyikével:

- Erősebb, az értékészleteket jobban szűkítő korlátok felírása, ld. redundáns korlátok, globális korlátok, felhasználói korlátok. A szűkebb értékészlet kisebb elágazási tényezőt biztosít.
- A keresési döntések egyes megválasztása, ld. változó és érték sorrendezés.

Az, hogy a fenti algoritmus 2. pontjában hogyan választjuk az  $n$  csomópontot, meghatározza a keresés típusát. Noha számos feladat ismert, melynek megoldására a CLP-be sikerrel ágyaztak be valamilyen informált, sot, lokális kereso eljárást, a CLP rendszerek legtöbbször egyszerű mélységi keresést használnak. Ennek oka a következő. Minden egyes keresési csomópontban minden változó aktuális értékészletének eltárolása túlzottan memóriai igényes lenne. Ezért inkrementális tárolást alkalmaznak, azaz a

keresési fa csomópontjaiban csak a szülő csomópontokhoz viszonyított értékkészlet-szukítéseket tartják nyilván. Így két, a fában egymástól távoli csomópont közti váltás csak a fa élei mentén történő lépegetéssel oldható meg, szerencsétlen esetben egészen a gyökérig tartó visszagörgetés után. Ennek költségét csak kivételes esetben, nagyon jó heurisztika mellett ellensúlyozza az informált keresés csomópont-hatékonysága.

## Változó sorrendezés

A *labeling* során az elágazás tárgyát képező  $x_i$  változó kiválasztására szolgáló heurisztikákat nevezzük változó sorrendezésnek (*variable ordering*). Gyakran hatékony az a heurisztika, hogy a feladat „nehéz magját”, a legproblematikusabb részfeladatot oldjuk meg eloször.

Ezt igyekeznek megvalósítani az ún. *first-fail* eljárás: feltételezzük, hogy minden érték lekötés azonos eséllyel vezet konzisztens megoldáshoz, így minél több eleme van  $D_i$ -nek, annál valószínűbb, hogy szükség esetén találunk  $x_i$ -hez konzisztens értéket. Fordítva pedig, minél kevesebb eleme van  $D_i$ -nek annál nehezebb  $x_i$ -hez értéket választani, ezért érdemes az aktuálisan legkisebb értékkészlettel rendelkező változón elágazni. A SICStus Prologban a beépített first-fail labeling eljárás a következőképpen hívható:

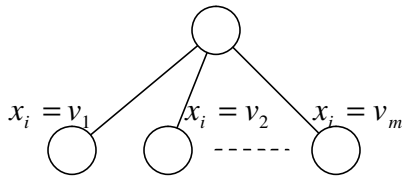
```
| ?- queens(4, Qs), labeling([ff], Qs).
```

## Érték sorrendezés

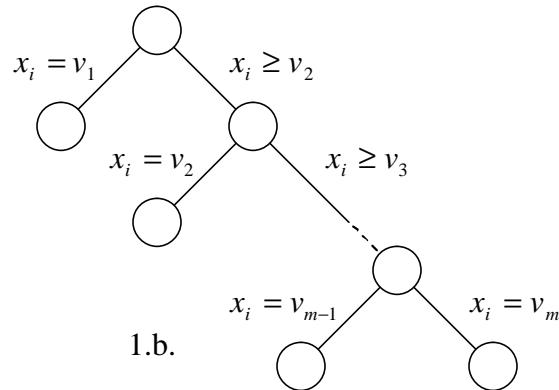
(*Value ordering*) Miután meghatároztuk az  $x_i$  változót, amelyen elágazunk, el kell döntenünk, hogy milyen sorrendben próbáljuk hozzárendelni annak lehetséges értékeit,  $D_i$  elemeit, azaz milyen sorrendben járjuk be a keresési fa ágait. Az érték sorrendezésnek akkor lehet jelentősége, ha csak az első megoldásra keresünk vagy egy branch-and-bound algoritmussal optimális megoldást keresünk. Ha mindenképpen be kell járnunk a teljes keresési fát, pl. mert az összes megoldást elő akarjuk állítani vagy mert nincs megoldás, akkor az érték sorrend választása irreleváns.

Ha az érték sorrend számít, érdemes eloször azt  $D_i$ -ből azt az értéket hozzárendelni  $x_i$ -hez, amelyik legnagyobb eséllyel vezet egy helyes megoldáshoz ill. az optimális megoldáshoz. Az ennek az „esélynek a mérésére” ismert heurisztikák a CSP gráfjában  $x_i$ -hez éllel kapcsolt, még lekötetlen változók értékkészleteiben számolják össze az  $x_i = v_j$  hozzárendeléssel konzisztens értékeket. Néhány tipikus alkalmazásban, pl. ütemezésben, specifikus, a tapasztalat által igazolt érték sorrend heurisztikák állnak rendelkezésre.

Alapértelmezett érték sorrendezési stratégiaként az elterjedt *CLP* rendszerek, ha a felhasználó másképp nem rendeli, növekvő sorrendben végigpróbálják  $D_i$  összes  $v_i$  elemét. A fa építése általában nem az egyszerűbbnek tűnő 1.a., hanem az 1.b. ábra szerint történik. Itt ugyanis az egyes ágakról történő esetleges sikertelen visszatérést követően a tárhoz adódik az  $x_i \geq v_{j+1}$  korlát, melynek propagálásával szerencsés esetben további értékkészlet-szűkítések mehetnek végbe.



1.a.



1.b.

A SICStus Prolog úgy, mint a legtöbb *CLP* rendszer, lehetőséget nyújt saját változó ill. érték sorrendezési heurisztikák írására is, erről az irodalomjegyzékben szereplő művek tartalmaznak részleteket.

## Redundáns korlátok

Egy redundáns korlát explicit megfogalmazása egy olyan feltételnek, amit egy *CSP* implicit módon tartalmaz: a formalizálásakor nem vettük fel korlátként, mégis tudható, hogy *CSP* összes konzisztens megoldására teljesül. A redundáns korlátok hozzávétele a *CSP*-hez nagyban gyorsíthatja a megoldási folyamatot a keresési fa hatékonyabb nyesése által.

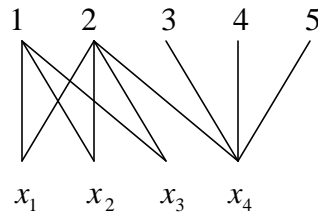
Természetesen felesleges olyan redundáns korlátot hozzávenni a *CSP*-hez, aminek kielégítését a már meglévő korlátokon futó él-konzisztencia algoritmusok is közvetlenül biztosítanak. Pl. ha a *CSP*-ben szerepelnek az  $a = b + c$  és  $c = d + e$  korlátok, akkor az  $a = b + d + e$  korlát hozzávétele további nyesést nem eredményez, sőt, a megoldást kis mértékben lassítja a propagációs lépések idejének növekedésével.

## Globális korlátok

Különböző feladatokban gyakran előfordulnak bizonyos jellegzetes összefüggések a feladat változói, vagy a változók egy részhalmazának elemei közt. Pl. megkövetelhetjük, hogy változók egy  $[x_1, \dots, x_k]$  listájának elemei páronként különböző értéket vegyenek

fel. Az ilyen, sok, nem határozott számú változó közti, azaz „globális” összefüggések leírására a *CLP* rendszerek gyakran implementálnak hatékony propagáló algoritmust, ún. *globális korlátot (global constraint)*.

A fenti „mind különböző” korlátot implementálja a SICStus-ban az *all\_different/1* predikátum. Ugyanezt a korlátot természetesen deklarálnánk úgy is, hogy minden  $x_i, x_j, i \neq j$  változó-párra felírjuk az  $x_i \neq x_j$  aritmetikai korlátot, de az messze nem vezetne olyan hatékony propagációhoz:



Tekintsük a fenti példát. A 4 változó mindegyikének értékkészletében azon értékek azon értékek szerepelnek, melyekhez éllel kötve van. Ekkor a  $x_i \neq x_j$  korlátok egyike sem tud tüzelni, hiszen bármely két változó leköthető két különböző értékre. Mégis, látható, hogy a feladat megoldhatatlan, mert az 1 és 2 értékekhez három változó is van, melyek csak a két érték valamelyikét vehetik fel.

Az *all\_different* korlát ezt felismeri: a fenti feladatot mint párosítási problémát oldja meg: a páros gráfban keresünk alulról teljes párosítást. Általában elmondható, hogy a globális korlátok számításigényesebbek, mint az egyszerű korlátok, de ez boven kifizetődik a hatékonyabb propagáció által. Alább az *n*-királyno feladat egy alternatív, hatékonyabb megvalósítását látjuk, globális korlátok használatával.

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

% A Qs lista N királyno biztonságos elhelyezését mutatja
% egy N*N-es sakktáblán: ha a lista i. eleme j, akkor
% az i. királynot az i. sor j. oszlopába kell helyezni.

queens(N, Qs) :-
    length(Qs, N),
    domain(Qs, 1, N),
    safe(Qs).

% safe(Qs): A Qs királyno-lista biztonságos.

safe(Qs) :-
    diag_rightrightdown(Qs, QsRD),
    diag_rightrightup(Qs, QsRU),
    all_different(Qs),
    all_different(QsRD),
    all_different(QsRU).

% Az oszlopok szerinti számozásból előállítjuk a
% \ átlók szerinti számozást.

diag_rightrightdown(Qs, QsRD) :-
```

```

diag_rightdown0(Qs, QsRD, 0).

diag_rightdown0([], [], _).
diag_rightdown0([L0|L], [LRD0|LRD], N):-
    LRD0 #= L0 - N,
    N1 is N+1,
    diag_rightdown0(L, LRD, N1).

% Az oszlopok szerinti számozázból eloállítjuk a
% / átlók szerinti számozást.

diag_rightup(Qs, QsRU):-
    diag_rightup0(Qs, QsRU, 0).

diag_rightup0([], [], _).
diag_rightup0([L0|L], [LRU0|LRU], N):-
    LRU0 #= L0 + N,
    N1 is N+1,
    diag_rightup0(L, LRU, N1).

```

## Felhasználói korlátok

Találkozhatunk olyan feladattal, amely nem írható le kézenfekvo módon a *CLP* rendszer beépített korlátaival, vagy a feladatra vonatkozó speciális ismeretek birtokában a beépített korlátokénál hatékonyabb propagációt tudnánk megvalósítani. Ilyenkor lehetőség van saját felhasználói korlátok definiálására. Egy  $c(x_1, \dots, x_k)$  korlát definiálásához meg kell adnunk, hogy amikor korlátban szereplo valamely  $x_i$  változóról kiderül, hogy nem veheti fel értékkészletének valamely  $v_i$  értékét, akkor a többi  $x_{j, j \neq i}$  változó értékkészletéből a  $c(x_1, \dots, x_k)$  alapján mely értékekről látható be, hogy inkonzisztensek. Errol az irodalomjegyzékben szereplo muvek tartalmaznak részleteket.

## Szimmetriatörések

Sok feladatban előfordulnak ekvivalens megoldások. Pl. a 8-királyno problémában lényegileg azonosnak (*szimmetrikusnak*) tekinthetjük azokat, amelyek egymásból forgatással előállíthatók. Egyrészt előfordulhat, hogy nem szeretnénk az összes szimmetrikus megoldást viszontlátni a megoldások közt. Másrészt, ha nagy számban fordulnak elő, lényegesen lassíthatják a megoldási folyamatot. A keresési fában ugyanis a szimmetrikus megoldás- vagy részmegoldás-csoportok minden tagja külön ágként szerepel, így a keresési algoritmus azok mindegyikét külön-külön bejárja. Ez teljesen

felesleges, hiszen ha az egyik ilyen ágon nincs megoldás, akkor a másik, szimmetrikus ágon sem lehet.

Szimmetriatörésnek nevezzük mindazon eljárásokat, melyek a szimmetrikus megoldáosztályok tagjaiból egy reprezentánst meghagynak, a többit kizárják. Ez újabb korlátok hozzáadásával történhet. Pl. a 8-királyno probléma forgatási szimmetriáját megtörhetjük azáltal, hogy eloirjuk, hogy az első sorban a bal térfélen, az első oszlopban pedig a felső térfélen van a királyno.

A szimmetriák egy speciális esete, amikor az adatstruktúrák egyszerűsítése (pl. négyzetes mátrixok bevezetése) céljából vagy egyéb megfontolásból *dummy* változókat veszünk fel, melyeknek értékkészlete egynél több elemet tartalmaz. Nyilvánvalóan irreleváns, hogy ezen értékek közül melyiket veszik fel. Ezért a *dummy* változókat a korlátozás programban egy-egy unáris korláttal le kell kötni.

## Optimalizálási feladatok

A különböző *CLP* eszközök azonos megközelítést használnak az optimalizálási feladatok kielégíthetőségi problémára való visszavezetésére. Tétélezzük fel, hogy minimalizálási problémáról van szó. Változóként felveszik a célfüggvény-értéket reprezentáló  $x_o$ -t, és a programhoz hozzáadják az  $x_o = O(X)$  korlátot, majd előállítanak egy kezdeti megoldást. Ezután ciklikusan elmentik az utolsó konzisztens megoldásból  $x_o$  értékét az *opt* változóba, és a *CSP*-hez hozzáadják az  $x_o \leq opt - 1$  korlátot. Így előállítanak egy újabb megoldást, és ezt ismételtetik, míg létezik az új korlátokat kielégítő megoldás.

Nyilvánvaló, hogy ekkor a megoldások célfüggvény-értéke szigorúan monoton csökken, továbbá, az utolsóként megtalált megoldás az (egyik) optimális. Az algoritmus a kezdeti megoldás megtalálása után *any-time* algoritmusként viselkedik, azaz komplex feladatok esetén, ha nincs lehetőség az optimális megoldás ésszeru időben történő megtalálására, megszakítható, az aktuális utolsó megoldás konzisztens lesz, minősége pedig a keresésre szánt idő növekedésével javul.

## **Felhasznált és ajánlott irodalom:**

Smith, B. M.: A tutorial on constraint programming.

[http://scom.hud.ac.uk/staff/scombms/Papers/ResearchReports/95\\_14.ps](http://scom.hud.ac.uk/staff/scombms/Papers/ResearchReports/95_14.ps)

Szeredi P.: Nagyhatékonyságú logikai programozás. BME egyetemi jegyzet.

<http://www.inf.bme.hu/~szeredi/nlp/nlp03a.html>

Marriott, K. and Stuckey, P. J.: Programming with constraints: an introduction.  
The MIT Press, 1998.

ILOG Solver 5.1. User's manual.

## Ellenorzo kérdések

- Mutass egy minél kisebb korlátozás programot, amelyet szerinted nem oldana meg keresés nélkül a Prolog.
- Milyen lehetőségeket ismersz korlátozás programok hatékonyságának növelésére?
- Adj megoldást a 8-királyno problémában az átlókra való tengelyes tükrözés szimmetriájának törésére.