Logické programování s omezujícími podmínkami
CLP: Constraint Logic Programming

Příklad: Magický čtverec

Řešení 1:

Úlohou je přiřadit číslice 0 až 9 jednotlivým proměnným **A, B, ..., I** tak, aby součet ve všech řádcích, sloupcích a v obou diagonálách byl stejný (= 15).

A	B	C
D	E	F
G	H	I

2	7	6
9	5	1
4	3	8

```
ctverec1([A,B,C,D,E,F,G,H,I]):-
      generuj1([A,B,C,D,E,F,G,H,I]),
      testuj1([A,B,C,D,E,F,G,H,I]).
```


Naivní řešení 1

```
ctverec1([A,B,C,D,E,F,G,H,I]):- generuj1([A,B,C,D,E,F,G,H,I]),
                                testuj1([A,B,C,D,E,F,G,H,I]).

generuj1(X):- c_seznam1(X).

c_seznam1([]).

c_seznam1([H|T]):- cislice(H), c_seznam1(T).

cislice(1). cislice(2). cislice(3). cislice(4). cislice(5). cislice(6). cislice(7).
cislice(8). cislice(9).

testuj1([A,B,C,D,E,F,G,H,I]):- ruzne([A,B,C,D,E,F,G,H,I]),
        A+B+C =:= 15, D+E+F =:= 15, G+H+I =:= 15,A+D+G =:= 15,
        B+E+H =:= 15, C+F+I =:= 15,C+E+G =:= 15, A+E+I =:= 15.

ruzne([]).
ruzne([H|T]):- neni_v(H,T), ruzne(T).

neni_v(_,[]).
neni_v(H,[HH|T]):- H=\=HH, neni_v(H,T).
```


Příklad: Magický čtverec

Řešení 1:

Úlohou je přiřadit číslice 0 až 9 jednotlivým proměnným A, B, ...I tak, aby součet ve všech řádcích, sloupcích a v obou diagonálách byl rovný 15.

A	B	C
D	E	F
G	H	I

```
ctverec1([A,B,C,D,E,F,G,H,I]):-
      generuj1([A,B,C,D,E,F,G,H,I]),
      testuj1([A,B,C,D,E,F,G,H,I]).
```

A = 2 B = 7 C = 6

D = 9 E = 5 F = 1

G = 4 H = 3 I = 8

2	7	6
9	5	1
4	3	8

Yes (**75.00s cpu**, solution 1, maybe more)

Kritika + řešení 2

Nevýhody: `generuj1` vytváří velké množství seznamů, které nesplňují ani základní podmínku na řešení (opakování číslic).
Počet volání predikátu `cislice` je 9^9 = 387 420 489.
Hledání prvního řešení trvalo **75 s**, další **84 s**, **146 s** .. .

Nový pokus o řešení: efektivnější predikát `generuj2`, v němž testujeme rozdílnost číslic už při procesu generování.

```
ctverec2([A,B,C,D,E,F,G,H,I]):-
            generuj2([A,B,C,D,E,F,G,H,I]),
            testuj2([A,B,C,D,E,F,G,H,I]).
generuj2(X):- c_seznam2(X).

c_seznam2([]).
c_seznam2([H|T]):- c_seznam2(T), cislice(H), neni_v(H,T).

testuj2([A,B,C,D,E,F,G,H,I]):-
   A+B+C =:= 15,D+E+F =:= 15,G+H+I =:= 15, A+D+G =:= 15,
   B+E+H =:= 15,C+F+I =:= 15,C+E+G =:= 15, A+E+I =:= 15.
```


Řešení 3

Výsledný program **ctverec2(L)** je výrazně rychlejší

- počet volání predikátu `cislice` klesl na 5 611 770
- hledání prvního řešení trvalo **0.50 s, 0.63 s, 1.17 s...**

Dalšího zlepšení lze dosáhnout úplným překrytím obou činností „generuj" a „testuj" tak, aby se v dílčí generované struktuře provedly všechny testy ihned, jakmile je to možné.

Je výhodné začít generování od proměnné, která se v omezeních vyskytuje nejčastěji.

```
ctverec3([A,B,C,D,E,F,G,H,I]):-
        gentest3(E,[]),
        gentest3(A,[E]),
        gentest3(I,[E,A]),          A+E+I =:= 15,
        gentest3(C,[E,A,I]),
        gentest3(G,[E,A,I,C]),      G+E+C =:= 15,
        gentest3(B,[E,A,I,C,G]),    A+B+C =:= 15,
        gentest3(H,[E,A,I,C,G,B]),  B+E+H =:= 15, G+H+I =:= 15,
        gentest3(D,[E,A,I,C,G,B,H]), A+D+G =:= 15,
        gentest3(F,[E,A,I,C,G,B,H,D]),D+E+F =:= 15, C+F+I =:= 15.
gentest3(D,L) :- cislice(D), neni_v(D,L).
```

V programu **ctverec3** se počet volání predikátu `cislice` sníží na **6498**, hledání prvního řešení trvalo 0.00 s (mimo rozlišovací schopnost). Hledání dalších řešení pak **0.02,0.02 s, ..**

Řešení 4

Testům na rozdílnost čísel se lze vyhnout sledováním seznamu ještě nevygenerovaných číslic:

```
ctverec4([A,B,C,D,E,F,G,H,I]):-
                vyber(E,[1,2,3,4,5,6,7,8,9],R1),
                vyber(A,R1,R2),
                vyber(I,R2,R3),  A+E+I =:= 15,
                vyber(C,R3,R4),
                vyber(G,R4,R5),  G+E+C =:= 15,
                vyber(B,R5,R6),  A+B+C =:= 15,
                vyber(H,R6,R7),  B+E+H =:= 15,  G+H+I =:= 15,
                vyber(D,R7,R8),  A+D+G =:= 15,
                vyber(F,R8,R9),  D+E+F =:= 15,  C+F+I =:= 15.

vyber(P,[P|T],T).
vyber(P,[H|T],[H|TT]):- vyber(P,T,TT).
```

Počet volání predikátu `cislice` se dále sníží na **3361**.

Hledání prvního řešení trvalo 0.00, dalších pak take 0 s. Všech řešení je 8.

Alternatity k metodě „generuj a testuj“

∴ **Metoda "testuj a generuj"**

Naznačené zvyšování efektivity programu překrýváním činnosti pro generování a testování vybízí k úplné záměně původního pořadí predikátů:

```
hledej(Reseni):- testuj(Reseni), generuj(Reseni).
```

∴ Nové výpočetní **schéma zadej omezení a generuj**

– postup by měl ***automaticky přerušit generování*** v okamžiku, kdy je porušeno některé omezení. **Naznačený postup není použitelný ve standardním Prologu, neboť**

– testující predikáty vyžadují, aby jejich argumenty byly ve chvíli volání vázané

– k vázání však dochází až v predikátu `generuj`

∴ Některé Prology poskytují speciální prostředek – **zmrazování cílů**

Využívá vestavěný predikát **`constrain(Promenna,Cil)`** (nebo jeho obdobu)

– je-li **`Promenna`** ve chvíli volání vázaná, volá se přímo **`Cil`**

– je-li **`Promenna`** ve chvíli volání volná, pokus o splnění cíle je odložen (zmrazen), až do chvíle vázání této proměnné

Prolog a **unifikace** jako jeho základní operace

∴Další problémy standardního Prologu: **unifikace**

Výhody: - umožňuje jednotnou manipulaci se všemi objekty jazyka

– programy jsou přehledné a predikáty jsou invertibilní

– dovoluje však srovnávat pouze syntakticky příbuzné objekty

Problém nastává, když chceme srovnávat i **objekty sémanticky ekvivalentní**

- např. pokus o unifikaci **2+3 = 5** je ve standardním Prologu neúspěšný, ačkoliv sémantika obou objektů je shodná (syntakticky jde o dva rozdílné objekty).

Částečné řešení. Standardní Prolog řeší tento problém použitím extralogického operátoru **is**, který však selhává při pokusech o unifikaci ve výrazech s volnými proměnnými, např. **`?- 5 is 3 + X.`** dá ve standardním Prologu negativní odpověď.

Společné řešení obou problémů nabízí **logické programování s omezujícími podmínkami :** unifikace je nahrazena **rozhodovací procedurou**, která umožňuje využívat jak **syntax**, tak **sémantiku** zpracovávaných objektů.

CLP

∴ **CLP není pouhým rozšířením standardního Prologu, je jeho zobecněním.**

Omezení jsou obvykle formulována pomocí operátorů pro srovnávání lineárních aritmetických výrazů.

Při tvorbě programů lze využít toho, že systémy obsahují mnoho predikátů, které jsou **zmrazovány automaticky**, pokud jejich argumenty obsahují volné proměnné.

Můžeme tak realizovat metodu zadej omezení a generuj v podobě prologovského programu, který neobsahuje žádné pomocné predikáty.

∴ Dva základní přístupy k řešení problémů s omezeními jsou

– inkrementální lineární metody = systém zpracovává najednou celou skupinu lin. omezení. Při přidání nového omezení ověří konzistenci celé nové soustavy.

– doménové technologie = jsou vytvořeny vazby mezi různými omezeními, která sdílejí stejné proměnné. Změna domény jedné proměnné vyvolá změny domén ostatních proměnných. Tento postup se řetězovitě šíří a pokračuje. Označuje se jako propagace omezení.

ECLiPSe

- Systém ECLiPSe je jedním ze systému rozšiřujících standardní Prolog o CLP.
- Automaticky zmrazuje aritmetické predikáty označené takto
  - **Reálný obor**: <, > >=,=<, =:=, =\=
  - **Racionální čísla**: $=, $<, …
  - **Celá čísla**: #=, ## (různý), #<, … [-10 000 000, 10 000 000], Proměnné lze přiřadit užší doménu pomocí operátoru **::**, např. B:: 0..100
- ***Řešení souboru omezujících podmínek*** probíhá efektivními deterministickými metodami : Simplexová m., Gaussova eliminace, atd.
- Uvedené aritm. predikáty jsou dále doplněny pomocí dalších knihoven, např. EPLEX (viz dále)

Pavouci a brouci

Máte zavřenou krabičku, ve které jsou zdraví pavouci a brouci. Dupe tam celkem 34 nožiček. Kolik máte brouků?

```
:-lib(fd).  % Knihovna fd: "finite domain"
insects(Beatles, Spiders,Legs):- Beatles #>0,Spiders#>0,
    8*Spiders + 6* Beatles #= Legs.
?- insects(B,S,34).
```

Systém doplní informaci o def.oboru **B:: 0..10^7 ,S:: 0..10^7**

0 + 6*B <= 8*S + 6*B (=34) <= 8*10^7 + 6*B, tedy 6*B <= 34 a **0<= B <6**

8*S + 0 <= 8*S + 6*B (=34) <= 8*S +6*10^7, tedy 8*S <= 34 a **0<= S <5**

Nyní využijeme informace o horních hranicích:

8*S + 6*B (=34) <= 8***S** + 6***B_{max}** = **8* S + 6*5**, tj. **34 <= 8*S + 30** čili **1<= S <5**

8*S + 6*B (=34) <= 8*S_{max} + 6*B = **8*4 + 6*B**, tj. **34 <= 32 + 6*B** čili **1<= B <6**

Využijeme změněnou informaci o dolních hranicích

8*S_{min} + 6*B <= 8*S + 6*B (=34), tj. **8*1+6*B <= 34** čili 1 **<= B < 5**

8*S + 6*B_{min} <= 8*S + 6*B (=34), tj. **8*S +6*1 <=34** čili 1 **<= S < 4**

Pokračujeme, dokud dochází k nějakým změnám, tj. výsledné řešení je

?- insects(B, S,34). B = 3, P = 2.

Pavouci a brouci

Máte zavřenou krabičku, ve které jsou zdraví pavouci a brouci. Dupe tam celkem 34 nožiček. Kolik máte brouků?

```
:-lib(fd).
insects(Beatles, Spiders,Legs):- Beatles #>0,Spiders#>0,
     8*Spiders + 6* Beatles #= Legs.
?- insects(B, S,34).
```

Řešení: Beatles=3, Spiders=2

```
?- insects(Beatles, Spiders,46).
```

Spiders= 2, Beatles=5, dál pak Spiders= 5, Beatles=1

Jiná implementace: Spiders= Spiders{[2..5]},Beatles={[1..5]}

Delayed goals:-46 + 8*Spiders{[2..5]}+6*Beatles={[1..5]}#=0

Důvod? Domény obou proměnných obsahují víc než 1 prvek!

```
?- insects(Beatles, Spiders,46).
No.
```


Pavouci a brouci

```
?- insects(B, S,46).
?- insects (B, P, 46).
B = B{[1..5]}
P = P{[2..5]}
There is 1 delayed goal.
Yes (0.00s cpu)

?- insects(B, P, 46), indomain(B).
B = 1
P = 5
Yes (0.00s cpu, solution 1, maybe more)
B = 5
P = 2
Yes (0.00s cpu, solution 2)
```

Zabudovaný predikát **indomain(Proměnna)**

Naváže hodnotu proměnné na nejnižší hodnotu její domény, která vyhovuje všem omezujícím podmínkám.

Při zpětném chodu nalézá všechna řešení !

Řešení úlohy magický čtverec v ECLiPSe

Řešení úlohy metodou zadej omezení a generuj:

```
:-lib(fd).
ctverec(L):-testuj5(L),generuj5(L).
testuj5(L):- L=[A,B,C,D,E,F,G,H,I],L::1..9,
      alldistinct(L),
      A+B+C #= 15,
      D+E+F #= 15,
      G+H+I #= 15,
      A+D+G #= 15,
      B+E+H #= 15,
      C+F+I #= 15,
      C+E+G #= 15,
      A+E+I #= 15.

generuj5([]).
generuj5([H|T]):- indomain(H), generuj5(T).
```


min_max(Cíl,Výraz)

Vestavěný predikát, který nalezne to řešení cíle **Cíl**, pro které je hodnota celočíselného výrazu **Výraz** minimální.

Použití? Např. úlohy plánování

Mějme 5 procesů **A,B,C,D** a **E**, které mohou probíhat i paralelně. Jejich časová náročnost nechť je **4,8,9,2** a **12.**

Označme **X před Y :** *„proces* **X** *musí skončit dřív než začne* **Y**"

Dále musí platit následující omezení:

A před B, A před D, B před C, D před E

Kdy nejdřív mohou být všechny procesy hotovy?

úlohy plánování

Uvažujeme procesy **A,B,C,D** a **E** o délce **4,8,9,2** a **12.**

Dále nechť platí:**A před B, A před D, B před C, D před E**

Kdy nejdřív mohou být všechny procesy hotovy?

```
:-lib(fd).
rozvrh(L,K) :- omezeni(L,K), generuj(L).
omezeni(L,K):- L=[A,B,C,D,E],L::0..35,
A+4 #<= B, A+4 #<= D, B+8 #<= C,
D+2 #<= E, C+9 #<= K, E+12#<= K,
min_max(indomain(K),K).
generuj([]).
generuj([H|T]):- indomain(H),generuj(T).
  ?- rozvrh(L,K).
  L = [0,4,12,4,6], K=21, ...
```


úlohy plánování

```
:-lib(fd).
rozvrh(L,K) :- omezeni(L,K), generuj(L).
omezeni(L,K):- L=[A,B,C,D,E],L::0..35,
  A+4 #<= B, A+4 #<= D, B+8 #<= C,
  D+2 #<= E, C+9 #<= K, E+12#<= K,
  min_max(indomain(K),K).
generuj([]).
generuj([H|T]):- indomain(H),generuj(T).
?- rozvrh(L,K).
  L = [0,4,12,4,6], K=21, ... 10 různých řešení
?- omezeni(L, K).
L = [0, 4, 12, D{[4..7]}, E{[6..9]}]
K = 21
There is 1 delayed goal, namely E-D #>= 2.
Yes (0.00s cpu)
```

Transportní úloha

Cena dopravy	Zákazník1	Zákazník2	Zákazník3	Denní produkce
Výrobce1	5 **X11**	1 **X12**	1 **X13**	4
Výrobce2	2 **X21**	6 **X22**	9 **X23**	6
Požadavky zákazníků	3	5	2	10

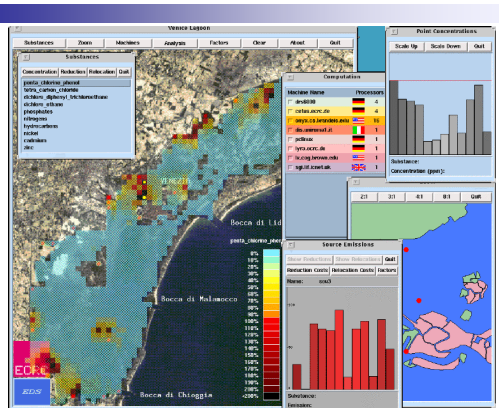
```
:-lib(fd).
doprava(L,C):-L=[X11,X12,X13,X21,X22,X23], L::0..9,
      X11 #>=0,X12 #>=0,X13 #>=0,X21 #>=0,
      X22 #>=0,X23 #>=0,X11+X21 #>=3,X12+X22 #>=5,
      X13+X23 #>=2,(X11+X12+X13)#< 5,(X21+X22+X23)#<7,
      min_max(generuj(L),5*X11+X12+X13+2*X21+6*X22+9*X23),
      C #= 5*X11 + X12 + X13 + 2*X21 + 6*X22 + 9*X23.
generuj([]).
generuj([H|T]):-indomain(H),generuj(T).
```

?-doprava(L,C). **X = [0,2,2,3,3,0], C =28.**

The Venice Lagoon DSS (parallel Eclipse)

Goal (Esprit project APPLAUSE): to assist the Venice Water Board in controlling the pollution of the Venice lagoon by identification of violations and recommending appropriate **interventions**, ie.:

- Reducing the total amount of emissions of a particular substance. In this case, it is necessary to locate a minimal number of sources whose emissions have to be reduced.
- Locating one or more offending sources if they emit more polluting substances than they are allowed by the Water Board.
- Identifying sources whose relocation would reduce the overall pollution to an acceptable level while keeping the emissions at the same level.
- Since both reduction and relocation of certain emissions implies some costs (e.g. to reduce or relocate production or to install a cleaning plant), it is necessary to find a solution which minimizes the overall cost.



Kde se dozvíte víc?

Mařík M., Štěpánková O., Lažanský J. *Umělá inteligence (2).* Academia, Praha, 1997.

Mach M., Paralič J. *Úlohy s ohraničeniami – od teórie k programovaniu.* Alfa, Košice, 2000.

Apt, Krzysztof (2003). *Principles of constraint programming.* Cambridge University Press.

On-line Guide to Constraint Programming, Roman Barták, 2001. http://kti.ms.mff.cuni.cz/~bartak/constraints/

The ECLiPSe Constraint Logic Programming System, Imperial College London, 2001. http://www.icparc.ic.ac.uk/eclipse