CLP(FD) - prohledávání

Vestavěné predikáty pro labeling

- Instanciace proměnné `Variable` hodnotami v její doméně

  `indomain( Variable )`

  hodnoty jsou instanciovány při backtrackingu ve vzrůstajícím pořadí

```
?- X in 4..5, indomain(X).
   X = 4 ? ;
   X = 5 ?

labeling( [] ).
labeling( [Var|Rest]  ) :-    % výběr nejlevější proměnné k instanciaci
     indomain( Var ),         % výběr hodnot ve vzrůstajícím pořadí
     labeling( Rest ).
```

- `labeling( Options, Variables )`

```
?- A in 0..2, B in 0..2, B#< A, labeling([], [A,B]).
```


Uspořádání hodnot a proměnných

- Při prohledávání je rozhodující **uspořádání hodnot a proměnných**
- Určují je **heuristiky výběru hodnot a výběru proměnných**

```
labeling( [] ).
labeling( Variables  ) :-
     select_variable(Variables,Var,Rest),
     select_value(Var,Value),
     ( Var #= Value,
       labeling( Rest )
     ;
       Var #\= Value ,         % nemusí dojít k instanciaci Var
       labeling( Variables )   % proto pokračujeme se všemi proměnnými včetně Var
     ).
```

- **Statické uspořádání:** určeno už před prohledáváním
- **Dynamické uspořádání:** počítá se během prohledávání

Výběr hodnoty

- Obecný princip výběru hodnoty: **první úspěch *(succeed first)***
  - volíme pořadí tak, abychom výběr nemuseli opakovat
  - `?- domain([A,B,C],1,2), A#=B+C.` optimální výběr A=2,B=1,C=1 je bez backtrackingu
- Parametry `labeling/2` ovlivňující výběr hodnoty př. `labeling([down], Vars)`
  - up: doména procházena ve vzrůstajícím pořadí (default)
  - down: doména procházena v klesajícím pořadí
- Parametry `labeling/2` řídící, jak je výběr hodnoty realizován
  - `step`: volba mezi `X #= M, X #\= M` (default)
    - viz dřívější příklad u "Uspořádání hodnot a proměnných"
  - enum: vícenásobná volba mezi všemi hodnotami v doméně
    - podobně jako při `indomain/1`

Výběr proměnné

- Obecný princip výběru proměnné: ***first-fail***
  - výběr proměnné, pro kterou je nejobtížnější nalézt správnou hodnotu
    pozdější výběr hodnoty pro tuto proměnnou by snadněji vedl k failu
  - vybereme proměnnou **s nejmenší doménou**
  - `?- domain([A,B,C],1,3), A#<3, A#=B+C.` nejlépe je začít s výběrem A
- Parametry `labeling/2` ovlivňující výběr proměnné
  - `leftmost`: nejlevější (default)
  - `ff`: s (1) nejmenší velikostí domény `fd_size(Var,Size)`
    (2) (pokud s nejmenší velikostí domény více, tak) nejlevější z nich
  - `ffc`: s (1) nejmenší velikostí domény
    (2) největším množstvím omezením „čekajících" na proměnné `fd_degree(Var,Size)`
    (3) nejlevější z nich
  - `min/max`: s (1) nejmenší/největší hodnotou v doméně proměnné
    (2) nejlevnější z nich `fd_min(Var,Min)/fd_max(Var,Max)`

Hledání optimálního řešení

(předpokládejme minimalizaci)

- Parametry `labeling/2` pro optimalizaci: `minimize(F)/maximize(F)`
  - `Cena #= A+B+C, labeling([minimize(Cena)], [A,B,C])`
- **Metoda větví a mezí *(branch&bound)***
  - algoritmus, který implementuje proceduru pro minimalizaci (duálně pro maximalizaci)
  - uvažujeme nejhorší možnou cenu řešení UB (např. cena už nalezeného řešení)
  - počítáme dolní odhad LB ceny částečného řešení
    LB je tedy nejlepší možná cena pro rozšíření tohoto řešení
  - procházíme strom a vyžadujeme, aby prozkoumávaná větev měla cenu $LB < UB$
    pokud je $LB \geq UB$, tak víme, že v této větvi není lepší řešení a odřízneme ji
    - přidává se tedy inkrementálně omezení LB#<UB pro snižující se UB tak, jak nalézáme kvalitnější řešení

Opakování: kumulativní rozvrhování

- Vytvořte rozvrh pro následující úlohy, tak aby nebyla překročena kapacita 13 zdroje, a **minimalizujte** celkovou dobu trvání

úloha	doba trvání	kapacita
t1	16	2
t2	6	9
t3	13	3
t4	7	7
t5	5	10
t6	18	1
t7	4	11

Řešení: kumulativní rozvrhování

```
| ?-  schedule(13, [16,6,13,7,5,18,4], [2,9,3,7,10,1,11], 69, Ss, End).
Ss = [0,16,9,9,4,4,0], End = 22 ?

schedule(Limit, Ds, Rs, MaxCas, Ss, End) :-
    domain(Ss, 0, MaxCas), End in 0..MaxCas,
    vytvor_ulohy(Ss,Ds,Rs,1,Tasks),
    cumulative(Tasks, [limit(Limit)]),
    after(Ss, Ds, End),      % koncový čas
    append(Ss, [End], Vars),
    labeling([minimize(End)],Vars).

vytvor_ulohy([],[],[],_Id,[]).
vytvor_ulohy([S|Ss], [D|Ds], [R|Rs], Id, [task(S,D,E,R,Id)|Tasks]):-
      NewId is Id+1,
      E #= S+D,
      vytvor_ulohy(Ss,Ds,Rs, NewId,Tasks).

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

Algoritmy pro řešení problému splňování podmínek (CSP)

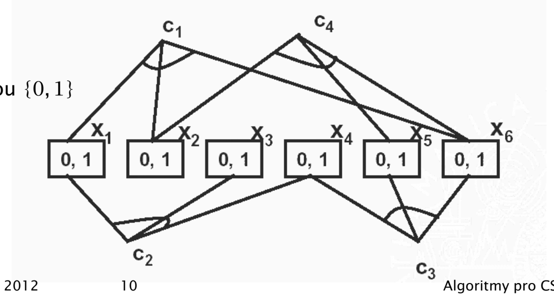
Grafová reprezentace CSP

- **Reprezentace podmínek**
  - intenzionální (matematická/logická formule)
  - extenzionální (výčet k-tic kompatibilních hodnot, 0-1 matice)
- **Graf:** vrcholy, hrany (hrana spojuje dva vrcholy)
- **Hypergraf:** vrcholy, hrany (hrana spojuje množinu vrcholů)
- Reprezentace CSP pomocí **hypergrafu podmínek**
  - vrchol = proměnná, hyperhrana = podmínka
- Příklad
  - proměnné $x_1, \ldots, x_6$ s doménou $\{0, 1\}$
  - omezení $c_1 : x_1 + x_2 + x_6 = 1$

    $c_2 : x_1 - x_3 + x_4 = 1$

    $c_3 : x_4 + x_5 - x_6 > 0$

    $c_4 : x_2 + x_5 - x_6 = 0$

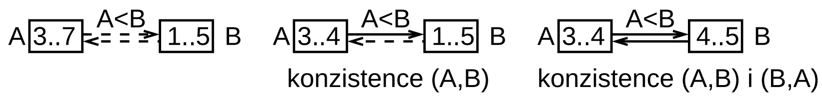


Binární CSP

- **Binární CSP**
  - CSP, ve kterém jsou pouze binární podmínky
  - unární podmínky zakódovány do domény proměnné
- **Graf podmínek** pro binární CSP
  - není nutné uvažovat hypergraf, stačí graf (podmínka spojuje pouze dva vrcholy)
- **Každý CSP lze transformovat na "korespondující" binární CSP**
- Výhody a nevýhody binarizace
  - získáváme unifikovaný tvar CSP problému, řada algoritmů navržena pro binární CSP
  - bohužel ale značné zvětšení velikosti problému
- Nebinární podmínky
  - složitější propagační algoritmy
  - lze využít jejich sémantiky pro lepší propagaci
    - příklad: `all_different` vs. množina binárních nerovností

Vrcholová a hranová konzistence

- **Vrcholová konzistence *(node consistency)* NC**
  - každá hodnota z aktuální domény V_i proměnné splňuje všechny unární podmínky s proměnnou V_i
- **Hranová konzistence *(arc consistency)* AC** pro **binární CSP**
  - **hrana** (V_i, V_j) **je hranově konzistentní**, právě když pro každou hodnotu x z aktuální domény D_i existuje hodnota y tak, že ohodnocení $[V_i = x, V_j = y]$ splňuje všechny binární podmínky nad V_i, V_j.
  - hranová konzistence je **směrová**
    - konzistence hrany (V_i, V_j) nezaručuje konzistenci hrany (V_j, V_i)



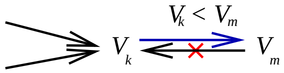
  - **CSP** je **hranově konzistentní**, právě když jsou všechny jeho hrany (v obou směrech) hranově konzistentní

Algoritmus revize hrany

- Jak udělat hranu (V_i, V_j) hranově konzistentní?
- Z domény D_i vyřadím takové hodnoty x, které nejsou konzistentní s aktuální doménou D_j (pro x neexistuje žádá hodnoty y v D_j tak, aby ohodnocení $V_i = x$ a $V_j = y$ splňovalo všechny binární podmínky mezi V_i a V_j)
- procedure revise$((i,j))$
  Deleted := false
  for $\forall x$ in D_i do
      if neexistuje $y \in D_j$ takové, že (x,y) je konzistentní
      then D_i := D_i $-$ $\{x\}$
          Deleted := true
      end if
  return Deleted
  end revise
- domain([V_1,V_2],2,4), V_1#< V_2 revise((1,2)) smaže 4 z D_1,D_2 se nezmění

Dosažení hranové konzistence problému

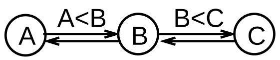
- Jak udělat CSP hranově konzistentní?
  - revize je potřeba opakovat, dokud se mění doména nějaké proměnné
  - efektivnější: opakování revizí můžeme dělat pomocí **fronty**
    - přidáváme do ní hrany, jejichž konzistence mohla být narušena zmenšením domény
- Jaké hrany přesně revidovat po zmenšení domény?
  - ty, jejichž konzistence může být zmenšením domény proměnné narušena jsou to hrany (i,k), které vedou do proměnné V_k se zmenšenou doménou



  - hranu (m,k) vedoucí z proměnné V_m, která zmenšení domény způsobila, není třeba revidovat (změna se jí nedotkne)

Algoritmus AC-3

- procedure AC-3(G)
  Q := $\{(i,j) \mid (i,j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi
  while Q non empty do
      vyber a smaž (k,m) z Q
      if revise((k,m)) then   % pridani pouze hran, ktere
        Q := Q $\cup$ $\{(i,k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve fronte
  end while
  end AC-3
- Příklad:



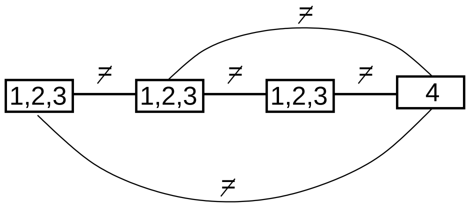
  A<B, B<C: ($\underline{3..7}$,1..5,1..5) $\xrightarrow{AB}$ (3..4,$\underline{1..5}$,1..5) $\xrightarrow{BA}$
  (3..4,$\underline{4..5}$,1..5) $\xrightarrow{BC}$ (3..4,4,$\underline{1..5}$) $\xrightarrow{CB}$ ($\underline{3..4}$,4,5)
  $\xrightarrow{AB}$ (3,4,5)
- Technika AC-3 je dnes asi nejpoužívánější, ale stále není optimální
- Jaké budou domény A,B,C po AC-3 pro: domain([A,B,C],1,10), A #= B + 1, C #< B

Je hranová konzistence dostatečná?

- Použitím AC odstraníme mnoho nekompatibilních hodnot
  - Dostaneme potom řešení problému?   NE
  - Víme alespoň zda řešení existuje?   NE
- domain([X,Y,Z],1,2), X#\= Y, Y#\= Z, Z#\= X
  - hranově konzistentní
  - nemá žádné řešení
- Jaký je tedy význam AC?
  - někdy dá řešení přímo
    - nějaká doména se vyprázdní ⇒ řešení neexistuje
    - všechny domény jsou jednoprvkové ⇒ máme řešení
  - v obecném případě se alespoň zmenší prohledávaný prostor

k-konzistence

- Mají NC a AC něco společného?
  - NC: konzistence jedné proměnné
  - AC: konzistence dvou proměnných
  - . . . můžeme pokračovat
- CSP je **k-konzistentní** právě tehdy, když můžeme libovolné konzistentní ohodnocení (k-1) různých proměnných rozšířit do libovolné k-té proměnné



4-konzistentní graf

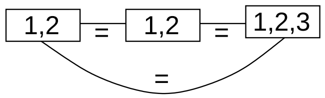
- Pro obecné CSP, tedy i pro nebinární podmínky

Silná k-konzistence

3-konzistentní graf

$(1,1)$ lze rozšířit na $(1,1,1)$

$(2,2)$ lze rozšířit na $(2,2,2)$



není 2-konzistentní

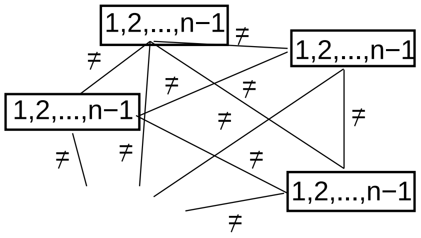
(3) nelze rozšířit

$(1,3)$ ani $(2,3)$ nejsou konzistentní dvojice (nerozšiřujeme je)

- CSP je **silně k-konzistentní** právě tehdy, když je j-konzistentní pro každé j≤k
- Silná k-konzistence ⇒ k-konzistence
- Silná k-konzistence ⇒ j-konzistence $\forall j \leq k$
- k-konzistence ⇏ silná k-konzistence
- NC = silná 1-konzistence = 1-konzistence
- AC = (silná) 2-konzistence

Konzistence pro nalezení řešení

- Máme-li graf s n vrcholy, jak silnou konzistenci potřebujeme, abychom přímo našli řešení?
  - silná n-konzistence je nutná pro graf s n vrcholy
    - n-konzistence nestačí (viz předchozí příklad)
    - silná k-konzistence pro k<n také nestačí



graf s n vrcholy

domény 1..(n-1)

silně k-konzistentní pro každé k<n

přesto nemá řešení

Řešení nebinárních podmínek

- k-konzistence má exponenciální složitost, v reálu se nepoužívá
- S n-árními podmínkami se pracuje přímo
- Podmínka je **obecně hranově konzistentní (GAC)**, právě když pro každou proměnnou V_i z této podmínky a každou hodnou $x \in D_i$ existuje ohodnocení zbylých proměnných v podmínce tak, že podmínka platí
  - `A + B #= C, A in 1..3, B in 2..4, C in 3..7` je obecně hranově konzistentní
- Využívá se sémantika podmínek
  - speciální typy konzistence pro globální omezení
    - viz `all_distinct`
  - konzistence mezí
    - propagace pouze při změně nejmenší a největší hodnoty v doméně proměnné
- Pro různé podmínky lze použít různý druh konzistence
  - A#<B: hranová konzistence, konzistence mezí

Konzistenční algoritmus pro nebinární podmínky

- Algoritmus s **frontou proměnných** (někdy též nazýván AC-8)
  - opakovaně se provádí revize podmínek, dokud se mění domény

```
procedure Nonbinary-AC-3-with-Variables((V,D,C))
Q := V
while Q non empty do
      vyber a smaž V_j ∈Q
      for ∀C takové, že V_j ∈ scope(C) do
          W := revise(V_j,C)
             // W je množina proměnných jejichž, doména se změnila
          if ∃ V_i ∈ W taková, že D_i = ∅ then return fail
          Q := Q ∪ {W}
end  Non-binary-consistency
```

  - **rozsah omezení** $scope(C)$: množina proměnných, na nichž je C definováno
- Implementace: u každé proměnné je seznam **vybraných podmínek** pro propagaci, REVISE procedury pro tyto podmínky definuje uživatel v závislosti na typu podmínky

Konzistence mezí

- ***Bounds consistency*** **BC**: slabší než obecná hranová konzistence
  - podmínka má **konzistentní meze (BC)**, právě když pro každou proměnnou V_j z této podmínky a každou hodnou $x \in D_j$ existuje ohodnocení zbylých proměnných v podmínce tak, že je podmínka splněna a pro vybrané ohodnocení y_i proměnné V_i platí $\min(D_i) \leq y_i \leq \max(D_i)$
  - stačí propagace pouze při **změně minimální nebo maximální hodnoty (při změně mezí)** v doméně proměnné
- **Konzistence mezí pro nerovnice**
  - `A #> B => min(A) = min(B)+1, max(B) = max(A)-1`
  - příklad: `A in 4..10, B in 6..18, A #> B`
    `min(A) = 6+1` ⇒ `A in 7..10`
    `max(B) = 10-1` ⇒ `B in 6..9`
  - podobně: `A #< B`, `A #>= B`, `A #=< B`

Konzistence mezí a aritmetická omezení

- `A #= B + C => min(A) = min(B)+min(C), max(A) = max(B)+max(C)`
  `min(B) = min(A)-max(C), max(B) = max(A)-min(C)`
  `min(C) = min(A)-max(B), max(C) = max(A)-min(B)`
  - změna `min(A)` vyvolá pouze změnu `min(B)` a `min(C)`
  - změna `max(A)` vyvolá pouze změnu `max(B)` a `max(C), ...`
- Příklad: `A in 1..10, B in 1..10, A #= B + 2, A #> 5, A #\= 8`
  `A #= B + 2` ⇒ `min(A)=1+2, max(A)=10+2` ⇒ `A in 3..10`
  ⇒ `min(B)=1-2, max(B)=10-2` ⇒ `B in 1..8`
  `A #> 5` ⇒ `min(A)=6` ⇒ `A in 6..10`
  ⇒ `min(B)=6-2` ⇒ `B in 4..8` (nové vyvolání `A #= B + 2`)
  `A #\= 8` ⇒ `A in (6..7) \/ (9..10)` (meze stejné, k propagaci `A #= B + 2` nedojde)
- Vyzkoušejte si: `A #= B - C, A #>= B + C`

Globální podmínky

- Propagace je lokální
  - pracuje se s jednotlivými podmínkami
  - interakce mezi podmínkami je pouze přes domény proměnných
- Jak dosáhnout více, když je silnější propagace drahá?
- Seskupíme několik podmínek do jedné tzv. **globální podmínky**
- Propagaci přes globální podmínku řešíme
  speciálním algoritmem navrženým pro danou podmínku
- Příklady:
  - `all_different` omezení: hodnoty všech proměnných různé
  - `serialized` omezení:
    rozvržení úloh zadaných startovním časem a dobou trvání tak, aby se nepřekrývaly

Propagace pro `all_distinct`

učitel	min	max
Jan	3	6
Petr	**3**	**4**
Anna	2	5
Ota	**2**	**4**
Eva	**3**	**4**
Marie	1	6

- **U = {X2, X4, X5}, dom(U) = {2, 3, 4}**:

  `{2,3,4}` nelze pro `X1, X3, X6`

  `X1 in 5..6, X3 = 5, X6 in {1} \/ (5..6)`

- **Konzistence:** $\forall \{X_1, \ldots, X_k\} \subset V : card\{D_1 \cup \cdots \cup D_k\} \geq k$
  stačí hledat **Hallův interval** I: velikost intervalu I je rovna
  počtu proměnných, jejichž doména je v I

- **Inferenční pravidlo**
  - $U = \{X_1, \ldots, X_k\}$, $dom(U) = \{D_1 \cup \cdots \cup D_k\}$
  - $card(U) = card(dom(U)) \Rightarrow \forall v \in dom(U), \forall X \in (V - U), X \neq v$
  - hodnoty v Hallově intervalu jsou pro ostatní proměnné nedostupné

- **Složitost**: $O(2^n)$ – hledání všech podmnožin množiny n proměnných (naivní)

  $O(n \log n)$ – kontrola hraničních bodů Hallových intervalů (1998)