Programování somezujícími podmínkami 6

Roman Barták, KTIML

bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

Co bylo minule

Hranová konzistence (AC)

- směrová hranová konzistence (DAC)
  - algoritmus DAC-1
  - aplikace na stromové CSP
- AC neodstraňuje všechny nekonzistence

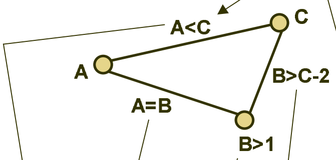
Konzistence po cestě (PC)

- stačí cesty délky 2
- PC pokrývá AC (cesty (i,j,i))
- reprezentace podmínek maticemi
- operace s podmínkami
  - průnik R_{ij} & R'_{ij}
  - složení $R_{ik} * R_{kj} \rightarrow R_{ik}$

Skládáme podmínky na cestě

A,B,C in {1,2,3}, B>1

A<C, A=B, B>C-2



$$\begin{pmatrix}011\\001\\000\end{pmatrix} \;\&\; \begin{pmatrix}100\\010\\001\end{pmatrix} * \begin{pmatrix}000\\010\\001\end{pmatrix} * \begin{pmatrix}110\\111\\111\end{pmatrix} = \begin{pmatrix}000\\001\\000\end{pmatrix}$$

Algoritmus PC-1

Jak udělat cestu (i,k,j) konzistentní?

$R_{ij} \leftarrow R_{ij}$ & $(R_{ik} * R_{kk} * R_{kj})$

Jak udělat CSP konzistentní?

opakovat konzistenci všech cest (délky 2) dokud se mění domény

Algoritmus PC-1

```
procedure PC-1(Vars,Constraints)
   n ← |Vars|, Yⁿ ← Constraints
   repeat
      Y⁰ ← Yⁿ
      for k = 1 to n do
         for i = 1 to n do
            for j = 1 to n do
               Yᵏ⁻¹ij ← Yᵏ⁻¹ij & (Yᵏ⁻¹ik * Yᵏ⁻¹kk * Yᵏ⁻¹kj)
   until Yⁿ=Y⁰
   Constraints ← Y⁰
end PC-1
```

Pokud zde použijeme $Y^{k-1}_{ii} \leftarrow Y^{k-1}_{ii}$ & $(Y^{k-1}_{ik} * Y^{k-1}_{kk} * Y^{k-1}_{ki})$ dostaneme AC-1

Jak zlepšit PC-1?

Je snad na PC-1 něco neefektivního?

pár „drobností“

- není potřeba držet všechny kopie Y^k stačí jedna kopie a indikátor změny
- některé výpočty nemají žádný efekt ($Y^k_{kk} = Y^{k-1}_{kk}$)
- polovinu výpočtů lze ušetřit ($Y_{ji} = Y^T_{ij}$)

zásadní problém

- při změně domény se znova revidují všechny cesty stačí procházet jen cesty ovlivněné revizí

Algoritmus revize hrany

```
procedure REVISE_PATH((i,k,j))
   Z ← Yij & (Yik * Ykk * Ykj)
   if Z=Yij then return false
   Yij ← Z
   return true
end REVISE_PATH
```

Pokud dojde ke změně budeme re-revidovat zasažené hrany

Cesty ovlivněné revizí

Protože $Y_{ji} = Y^T_{ij}$, stačí brát pouze cesty (i,k,j) pro i≤j.

Nechť při revizi cesty (i,k,j) došlo ke změně domény cesty (i,j)

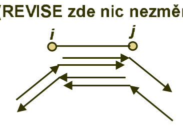
Situace a: i<j

je potřeba znova revidovat cesty obsahující (i,j) nebo (j,i)

cesty (i,j,j) a (i,i,j) není potřeba revidovat (REVISE zde nic nezmění)

$S_a =$ {(i,j,m) | i ≤ m ≤ n & m≠j}
∪ {(m,i,j) | 1 ≤ m ≤ j & m≠i}
∪ {(j,i,m) | j < m ≤ n}
∪ {(m,j,i) | 1 ≤ m < i}

$|S_a| = 2n-2$



Situace b: i=j

je potřeba znova revidovat cesty obsahující *i* uvnitř

cesty (i,i,i) a (k,i,k) není potřeba revidovat

$S_b =$ {(p,i,m) | 1 ≤ m ≤ n & 1 ≤ p ≤ m} - {(i,i,i),(k,i,k)}

$|S_b| = n*(n-1)/2 - 2$

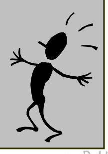
Algoritmus PC-2

Cesty bereme jen s jednou orientací (pozor neplést s DPC)

Po revizi kontrolujeme jen zasažené hrany

Algoritmus PC-2

```
procedure PC-2(G)
    n ← |nodes(G)|
    Q ← {(i,k,j) | 1 ≤ i ≤ j ≤ n & i≠k & j≠k}
    while Q non empty do
        select and delete (i,k,j) from Q
        if REVISE_PATH((i,k,j)) then
            Q ← Q ∪ RELATED_PATHS((i,k,j))
    end while
end PC-2
```



```
procedure RELATED_PATHS((i,k,j))
    if i<j then return S_a else return S_b
end RELATED_PATHS
```


Další PC algoritmy

PC-3 (Mohr, Henderson - 1986)

- založen na principu počítání podpor (jako AC-4)
- algoritmus je chybný!
  - Pokud zjistí, že dvojice (*a*,*b*) nemá na hraně (i,j) podporu u další proměnné, vyřadí *a* z D_i a *b* z D_j.

PC-4 (Han, Lee - 1988)

- opravuje PC-3 algoritmus
- založen na počítání podpor pro dvojice (b,c) hrany (i,j)

PC-5 (Singh - 1995)

- využívá myšlenky AC-6
- pamatuje si pouze jednu podporu a při její ztrátě hledá další

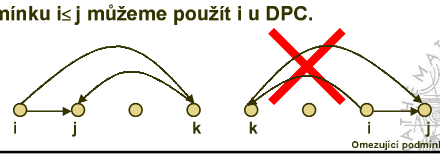
Směrová konzistence po cestě (DPC)

Podobně jako u AC můžeme i složitost PC algoritmů zmenšit zavedením směrovosti cesty.

Definice: CSP je *směrově hranově konzistentní* (*directional path consistent*) při daném uspořádání proměnných, právě když všechny cesty (i,k,j), takové, že i<k a j<k, jsou konzistentní.

Pozor podmínky i<k a j<k jsou jiné než i ≤ j použité pro odstranění symetrie!

Podmínku i≤ j můžeme použít i u DPC.



Algoritmus DPC-1

Podobně jako u DAC-1, každou cestu procházíme právě jednou (jdeme od konce).

Navíc můžeme ušetřit díky symetrii podmínek (i≤ j).

Algoritmus DPC-1

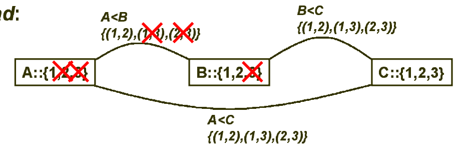
```
procedure DPC-1(Vars,Constraints)
    n ← |Vars|, E ← { (i,j) | i<j & C_i,j∈Constraints}
    for k = n to 1 by -1 do
        for i = 1 to k-1 do
            for j = i to k do
                if (i,k)∈E & (j,k)∈E then
                    C_ij ← C_ij & (C_ik * C_kk * C_kj)
                    E ← E ∪ {(i,j)}
            end for
        end for
    end for
end DPC-1
```


Vztah DPC k PC a AC

Zřejmě PC implikuje DPC.

Platí to také naopak?

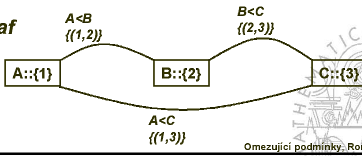
Příklad:



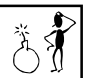
Graf je DPC, ale není PC!

Dokonce není ani AC.

PC a AC graf



Omezení PC algoritmů



Paměťové nároky

- protože PC eliminuje dvojice hodnot z podmínek, potřebuje používat extenzionální reprezentaci podmínky ({0,1}-matice)

Poměr výkon/cena

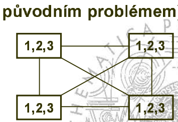
- PC eliminuje více (nebo stejně) nekonzistencí jako AC, poměr výkonu ke zjednodušení problému je ale mnohem horší než u AC

Změny grafu podmínek

- PC přidává hrany (podmínky) i tam, kde původně nebyly a mění tak konektivitu grafu
- to vadí při dalším řešení problému, kdy se nemohou používat heuristiky odvozené od grafu (resp. dané původním problémem)

PC stejně není dostatečné

- A,B,C,D in {1,2,3}
  A≠B, A≠C, A≠D, B≠C, B≠D, C≠D
  je PC a přesto nemá řešení



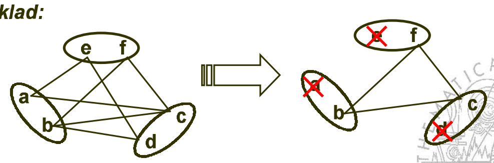
Na půli cesty od AC k PC

Jak oslabit PC, aby algoritmus:

- neměl paměťové nároky PC
- neměnil graf podmínek
- byl silnější než AC?

Testujeme PC jen v případě, když je šance, že to povede k vyřazení hodnoty z domény proměnné!

Příklad:



Omezená konzistence po cestě (RPC)

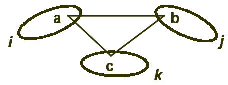
PC hrany se testuje pouze tehdy, pokud vyřazení dvojice může vést k vyřazení některého z prvků z domény příslušné proměnné.

Jak to poznáme?

Jedná se o jedinou vzájemnou podporu.

Definice: Vrchol *i* je *omezeně konzistentní po cestě* (*restricted path consistent*) právě když:

- každá hrana vedoucí z *i* je hranově konzistentní (AC)
- pro každé $a \in D_i$ platí:
  je-li *b* jediná podpora *a* ve vrcholu *j*, potom v každém vrcholu *k* (spojeném s *i* a *j*) existuje hodnota *c* tak, že (*a*,*c*) a (*b*,*c*) jsou kompatibilní s příslušnými podmínkami (PC).



Algoritmus RPC - inicializace

Založeno na AC-4, kde máme počítání podpor + seznam cest pro PC

Algoritmus inicializace RPC

```
procedure INITIALIZE(G)
    Q_AC ← {} , Q_PC ← {} , S ← {}                % vyprázdnění datových struktur
    for each (i,j)∈arcs(G) do
        for each a∈Di do
            total ← 0
            for each b∈Dj do
                if (a,b) is consistent according to the constraint Ci,j then
                        total ← total + 1,    Sj,b ← Sj,b ∪ {<i,a>}
            end for
            counter[(i,j),a] ← total
            if counter[(i,j),a] = 0 then
                Q_AC ← Q_AC ∪ {<i,a>},  delete a from Di
            else if counter[(i,j),a] = 1 then
                    for each k such that (i,k)∈arcs(G) & (k,j)∈arcs(G) do
                        Q_PC ← Q_PC ∪ {(<i,a>,j,k)}
            end if
        end for
    end for
    return (Q_AC, Q_PC)
end INITIALIZE
```


Algoritmus RPC - kontrola AC

Algoritmus AC v rámci RPC

```
procedure PRUNE(Q_AC, Q_PC)
    while Q_AC non empty do
        select and delete any pair <j,b> from Q_AC
        for each <i,a> from S_j,b do
            counter[(i,j),a] ← counter[(i,j),a] - 1
            if counter[(i,j),a] = 0 & "a" is still in D_i then
                    delete "a" from D_i
                    Q_AC ← Q_AC ∪ {<i,a>}
            else if counter[(i,j),a] = 1 then
                    for each k such that (i,k)∈arcs(G) & (k,j)∈arcs(G) do
                        Q_PC ← Q_PC ∪ {(<i,a>,j,k)}
                else
                    for each k such that (i,k)∈arcs(G) & (k,j)∈arcs(G) do
                      if counter[(i,k),a] = 1 then
                        Q_PC ← Q_PC ∪ {(<i,a>,k,j)}
            end if
        end for
    end while
    return Q_PC
end PRUNE
```


Algoritmus RPC

Nejprve uděláme AC a potom testuje vybrané PC, případně se vracíme k AC.

Algoritmus RPC

```
procedure RPC(G)
    (Q_AC, Q_PC) ← INITIALIZE(G)
    Q_PC ← PRUNE(Q_AC, Q_PC)              % první běh AC
    while Q_PC non empty do
        select and delete any triple (<i,a>,j,k) from Q_PC
        if a ∈D_i then
            {<j,b>} ← {<j,x> ∈ S_ia | x∈D_j }   % jediná podpora pro a
            if {<k,c>∈ S_ia ∩ S_jb | c∈D_k }=∅ then
              counter[(i,j),a] ← 0
              delete "a" from D_i
              Q_PC ← prune({<i,a>}, Q_PC)   % opakujeme AC
            end if
        end if
    end while
end RPC
```