Programování
s omezujícími podmínkami

5

Roman Barták
Katedra teoretické informatiky a matematické logiky

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

Konzistenční techniky

Dosud jsme podmínky používali jen pasivně (jako test) …
… maximálně jsme analyzovali důvod jejich nesplnění.

Nešlo by podmínky používat aktivněji?

Příklad:

A in 3..7, B in 1..5 domény proměnných

A<B

- z domén lze řadu hodnot vyřadit bez ztráty řešení
- dostaneme A in 3..4, B in 4..5

Poznámka: ne všechny zbývající kombinace jsou konzistentní (například A=4, B=4)

- **Jak odstranit nekonzistentní hodnoty z domén proměnných v síti podmínek?**

Vrcholová konzistence

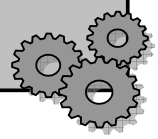
- Unární podmínky převedeme do domén proměnných.

Definice:

- **Vrchol** reprezentující proměnnou X je **vrcholově konzistentní (node consistent),** právě když každá hodnota z aktuální domény D_x splňuje všechny unární podmínky na X.
- **CSP** je **vrcholově konzistentní**, právě když je každý jeho vrchol vrcholově konzistentní.

Algoritmus NC

```
procedure NC(G)
    for each variable X in nodes(G) do
        for each value V in the domain D_X do
            if unary constraint on X is inconsistent with V then
                                delete V from D_X
        end for
    end for
end NC
```



Hranová konzistence

- Nadále se budeme zabývat jen binárními CSP
  tedy podmínka odpovídá hraně v grafu podmínek.

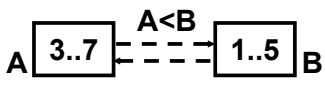
Definice:

- **Hrana (V_i,V_j) je hranově konzistentní (arc consistent),** právě když pro každou hodnotu x z aktuální domény D_i existuje hodnota y v aktuální doméně D_j tak, že ohodnocení V_i =x a V_j = y splňuje všechny binární podmínky nad V_i, V_j.

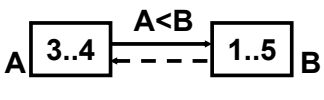
  Poznámka: Koncept hranové konzistence je směrový, tj. konzistence hrany (V_i,V_j) nezaručuje automaticky konzistenci hrany (V_j,V_i).

- **CSP je hranově konzistentní,** právě když je každá jeho hrana *(V_i,V_j)* hranově konzistentní (v obou směrech).

Příklad:

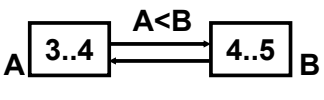


není hranově konzistentní



(A,B) je konzistentní

(A,B) i (B,A) jsou konzistentní



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 žádná hodnota *y* v D_j tak, aby ohodnocení V_i = x, V_j = y splňovalo všechny binární podmínky mezi V_i a V_j).

Algoritmus revize hrany

```
procedure REVISE((i,j))
    DELETED ← false
    for each X in Di do
        if there is no such Y in Dj such that (X,Y) is consistent, i.e.,
                    (X,Y) satisfies all the constraints on Vi, Vj then
            delete X from Di
            DELETED ← true
        end if
    end for
    return DELETED
end REVISE
```

Je vhodné také oznámit, pokud došlo k vymazání nějaké hodnoty.

Algoritmus AC-1

Jak udělat CSP hranově konzistentní?

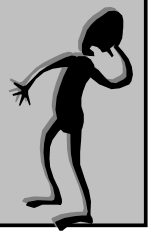
- Provedeme revizi každé hrany.
- Pozor, to nestačí! Pokud revize hrany zmenší doménu, mohou se již zrevidované hrany stát nekonzistentní.

A<B, B<C: (3..7,*1..5*,1..5) (*3..4*,1..5,1..5) (3..4,4..5,*1..5*) (3..4,*4*,1..5) (3..4,*4*,5) (3,4,5)

- Revize tedy budeme opakovat tak dlouho, dokud dochází ke zmenšení nějaké domény.

Algoritmus AC-1

```
procedure AC-1(G)
    repeat
        CHANGED ← false
        for each arc (i,j) in G do
            CHANGED ← REVISE((i,j)) or CHANGED
            if Di=∅ then stop with fail
        end for
    until not(CHANGED)
end AC-1
```



Neefektivita AC-1

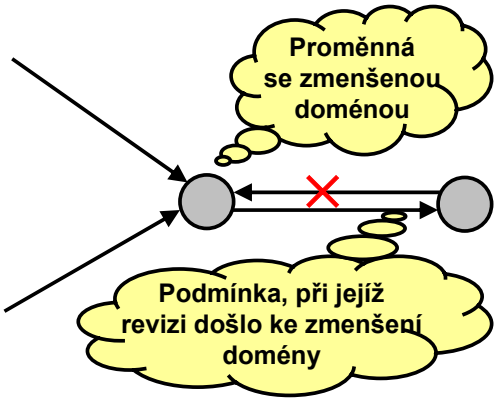
- Pokud zmenšíme jedinou doménu, provádí se stejně revize všech hran, i když změnou domény nejsou vůbec zasaženy.

Jaké hrany se mají tedy zrevidovat po zmenšení domény?

- Hrany, jejichž konzistence může být zmenšením domény proměnné narušena.
- Jedná se o hrany vedoucí do inkriminované proměnné.

Můžeme ještě ušetřit!

Hranu vedoucí z proměnné, která zmenšení domény způsobila, není potřeba revidovat (změna se jí nedotkne).



Algoritmus AC-2

Zobecněná verze Waltzova ohodnocovacího algoritmu

- V každém kroku vyřídí hrany vedoucí z daného vrcholu do již prošlých vrcholů (tj. podgraf z již navštívených vrcholů udržujeme AC).

Algoritmus AC-2

```
procedure AC-2(G)
    for i ← 1 to n do                          % n je počet proměnných
        Q ← {(i,j) | (i,j)∈arcs(G), j<i}       % hrany pro první revizi
        Q' ← {(j,i) | (i,j)∈arcs(G), j<i}      % hrany pro re-revizi
        while Q non empty do
            while Q non empty do
                select and delete (k,m) from Q
                if REVISE((k,m)) then
                    Q' ← Q' ∪ {(p,k) | (p,k)∈arcs(G), p≤i, p≠m }
                    if D_k=∅ then stop with fail
            end while
            Q ← Q'; Q' ← empty
        end while
    end for
end AC-2
```

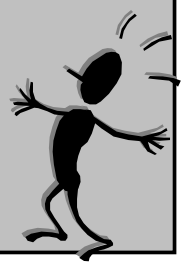
Algoritmus AC-3

Opakování revizí můžeme dělat elegantněji než AC-2

1. stačí jediná fronta hran, které je potřeba zrevidovat (znova zrevidovat)
2. do fronty přidáváme jen hrany, jejichž konzistence mohla být narušena zmenšením domény (jako AC-2)

Algoritmus AC-3

```
procedure AC-3(G)
    Q ← {(i,j) | (i,j)∈arcs(G), i≠j}        % seznam hran pro revizi
    while Q non empty do
        select and delete (k,m) from Q
        if REVISE((k,m)) then
            if D_k=∅ then stop with fail
            Q ← Q ∪ {(i,k) | (i,k)∈arcs(G), i≠k, i≠m}
        end if
    end while
end AC-3
```

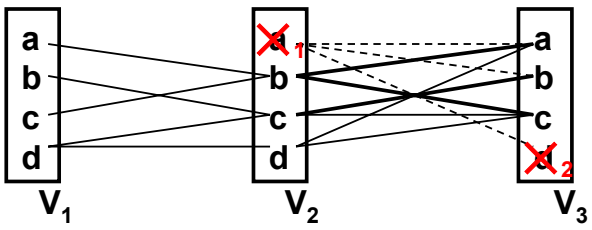


Technika AC-3 je dnes asi nejpoužívanější, ale pořád není optimální!

Podpory

Fakt (AC-3):

- Při každé revizi hrany testujeme množství dvojic hodnot na konzistenci vzhledem k podmínce.
- Tyto testy znova opakujeme při každé další revizi hrany.



1. Při revizi hrany V_2,V_1 vyřadíme hodnotu *a* z domény proměnné V_2.

2. Nyní musíme prozkoumat doménu proměnné V_3, zda některá z hodnot a, b, c, d neztratila svoji podporu ve V_2.

Pozorování:

Hodnoty a, b, c není potřeba znova kontrolovat, protože mají ve V_2 také jinou podporu než a.

Podpora pro $a \in D_i$ je $\{<j,b> \mid b \in D_j , (a,b) \in C_{i,j}\}$

Nemohli bychom podpory spočítat pouze jednou a při opakovaných revizích jich využívat?

Hledání podpor

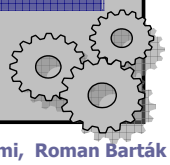
- Udržujeme seznam hodnot, které sami podporujeme (víme, komu říct, když zmizíme), a počet vlastních podpor (víme, co nám chybí)

Naplnění datových struktur s podporami

```
procedure INITIALIZE(G)
    Q ← {} , S ← {}                          % vyprázdnění datových struktur
    for each arc (Vi,Vj) in arcs(G) do
        for each a in Di do
            total ← 0
            for each b in Dj do
                if (a,b) is consistent according to the constraint Ci,j then
                    total ← total + 1
                    Sj,b ← Sj,b ∪ {<i,a>}
                end if
            end for
            counter[(i,j),a] ← total
            if counter[(i,j),a] = 0 then
                delete a from Di
                Q ← Q ∪ {<i,a>}
            end if
        end for
    end for
    return Q
end INITIALIZE
```

$S_{j,b}$ - množina dvojic <i,a>, pro které je <j,b> podporou

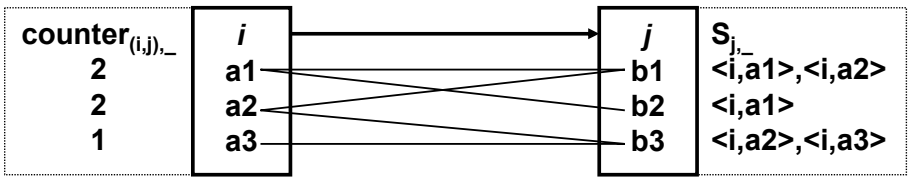
***counter[(i,j),a]* - počet podpor, které má hodnota *a* z D_i u proměnné V_j**



Hledání a využití podpor

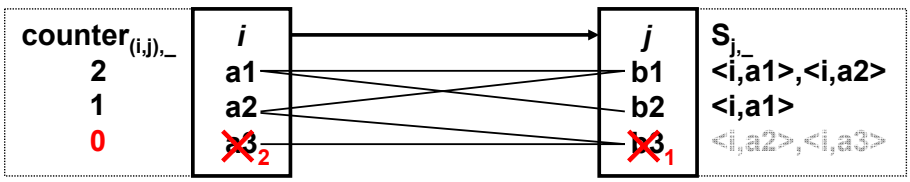
Situace:

v algoritmu INITIALIAZE jsme právě zpracovali hranu (i,j)



Využití struktur s podporami:

1. Nechť z nějakého důvodu vyřadíme b3.
2. Podíváme se do $S_{j,b3}$ na hodnoty, pro které je b3 podporou (tj. <i,a2>,<i,a3>).
3. U těchto hodnot snížíme counter (tj. odstraníme jim jednu podporu).
4. Pokud se některý counter vynuloval (a3), potom pokračujeme s příslušnou hodnotou od kroku 1.



Algoritmus AC-4

- Algoritmus AC-4 je optimální v nejhorším případě!

Algoritmus AC-4

```
procedure AC-4(G)
    Q ← INITIALIZE(G)
    while Q non empty do
        select and delete any pair <j,b> from Q
        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
                if D_i=∅ then stop with fail
                Q ← Q ∪ {<i,a>}
            end if
        end for
    end while
end AC-4
```

Bohužel v průměrném případě si tak dobře nevede
… navíc tady máme velkou paměťovou náročnost!

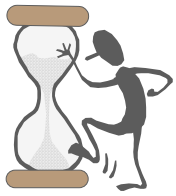
Další AC algoritmy

- **AC-5 (Hentenryck, Deville, Teng)**
  - generický algoritmus pro hranovou konzistenci
  - lze ho redukovat jak na AC-3 tak na AC-4
  - může využít sémantiku podmínek
    - funkcionální, anti-funkcionální a monotónní podmínky
- **AC-6 (Bessière, Cordier)**
  - zlepšuje paměťovou náročnost a průměrný čas AC-4
  - drží si pouze jednu podporu, další podporu hledá až při ztrátě aktuální podpory
- **AC-7 (Bessière, Freuder, Régin)**
  - založen na podporách (jako AC-4 a AC-6)
  - využívá „dvousměrnosti" podmínky

…

Algoritmus AC-3.1

optimální algoritmus AC-3



Pozorování:

- AC-3 není (teoreticky) optimální
- AC-4 je (teoreticky) optimální, ale (prakticky) pomalý
- AC-6/7 jsou (prakticky) rychlejší než AC-4, ale složité

Co je na AC-3 neefektivní?

- Hledání podpor v REVISE, které začíná vždy od nuly!

if „there is no such Y in D_j such that (X,Y) is consistent" then

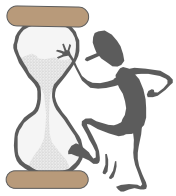
AC-3.1 (Zhang, Yap)

- běh stejný jako u AC-3
- pro každou hodnotu si pamatuje poslední nalezenou podporu (last) v každém směru a hledání začíná u ní

```
procedure EXIST((i,x),j)
    y ← last((i,x),j)
    if y∈Dj then return true
    while y←next(y,Dj) & y≠nil do
        if (x,y)∈C(i,j) then
            last((i,x),j) ← y
            return true
    end while
    return false
end EXIST
```


Algoritmus AC-2001

optimální algoritmus AC-3



Verze AC-3 s frontou proměnných (AC-8)

```
procedure AC-2001(G)
    Q ← {i | i∈nodes(G)} % seznam vrcholů pro revizi
    while Q non empty do
        select and delete j from Q
        for each i∈nodes(G) such that (i,j)∈arcs(G) do
            if REVISE2001(i,j) then
                if Di=∅ then return fail
                Q ← Q ∪ {i}
        end for
    end while
    return true
end AC-2001
```

```
procedure REVISE2001(i,j)
    DELETED ← false
    for each x in Di do
        if last((i,x),j)∉Dj then
            if ∃y∈Dj y>last((i,x),j)
            & (x,y)∈C(i,j) then
                last((i,x),j) ← y
            else
                delete x from Di
                DELETED ← true
            end if
    end for
    return DELETED
end REVISE2001
```

Poznámka:

- Algoritmus fakticky pracuje s rozdílovými množinami tj. pro každou proměnou si pamatuje, jaké hodnoty byly smazány z domény od poslední revize.

Směrová hranová konzistence

- **Pozorování 1:**
  AC má směrový charakter, ale CSP není směrové.
- **Pozorování 2:**
  AC musí opakovat revize hran a počet opakování závisí nejen na počtu hran, ale i na velikosti domén (while cyklus).
- **Můžeme AC nějak oslabit, abychom každou hranu revidovali pouze jedenkrát?**

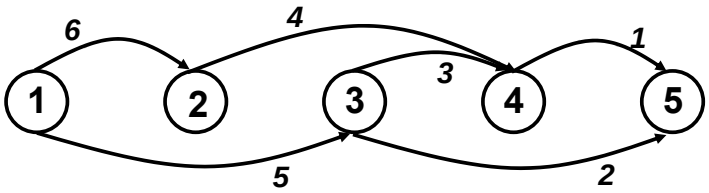
Definice:

CSP je **směrově hranově konzistentní (directional arc consistent)** při daném uspořádání proměnných, právě když každá hrana (i,j), kde i<j, je hranově konzistentní.

- Opět kontrolujeme každou hranu, tentokrát v jednom směru

Algoritmus DAC-1

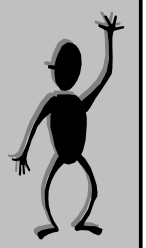
1. Konzistenci hrany požadujeme jen v jednom směru
2. Proměnné jsou uspořádané
   ↳ žádný orientovaný cyklus hran!



Pokud hrany projdeme v dobrém pořadí, nemusíme revize opakovat!

Algoritmus DAC-1

```
procedure DAC-1(G)
    for j = |nodes(G)| to 1 by -1 do
        for each arc (i,j) in G such that i<j do
            REVISE((i,j))
            if Di=∅ then stop with fail
        end for
    end for
end DAC-1
```



Použití DAC

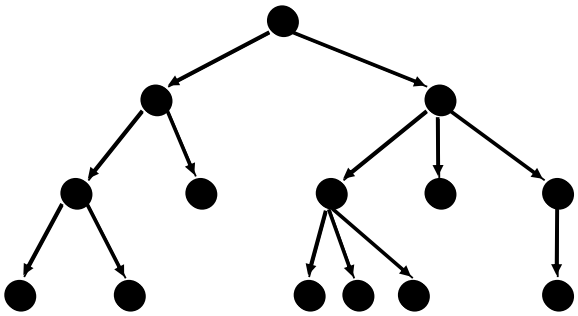
AC zřejmě zahrnuje DAC (je-li CSP AC, pak je i DAC)

Je DAC k něčemu dobré?

- DAC-1 je nepochybně efektivnější než AC-x
- existují problémy, kde DAC stačí

Příklad: Pokud graf CSP tvoří strom, potom stačí aplikovat DAC, abychom problém vyřešili bez navracení.

- **Jak uspořádáme vrcholy pro DAC?**
- **A jak uspořádáme vrcholy pro ohodnocování?**



1. Aplikujeme DAC v uspořádání vrcholů od kořene.

2. Ohodnocujeme vrcholy v pořadí od kořene.

DAC garantuje, že lze vždy najít hodnotu potomka kompatibilní s rodičem.

Vztah DAC k AC

Pozorování:

CSP je hranově konzistentní, jestliže pro dané uspořádání proměnných je směrově hranově konzistentní v obou směrech.

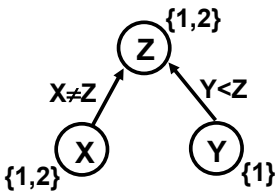
Můžeme AC dosáhnout tak, že aplikujeme DAC v obou směrech?

Obecně NE, ale…

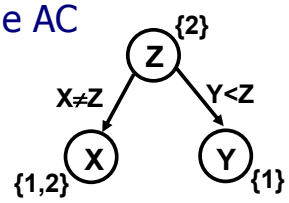
Příklad:

X in {1,2}, Y in {1}, Z in {1,2}, $X \neq Z, Y < Z$

při uspořádání X,Y,Z se domény nezmění



při uspořádání Z,Y,X se změní pouze Z, ale nedostane AC



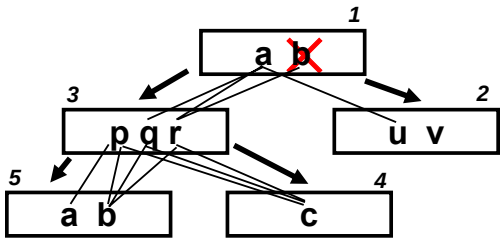
Pokud ale jako první zvolíme uspořádání Z,Y,X, dostaneme AC.

Od DAC k AC pro stromové CSP

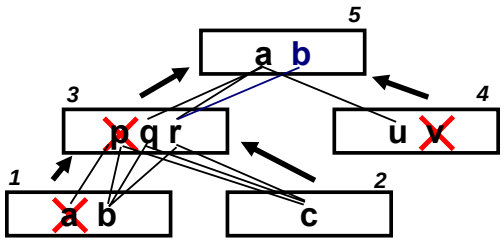
Pokud je DAC aplikováno na stromové CSP nejprve pro uspořádání vrcholů od kořene a po té v obráceném pořadí, potom získáme (plnou) hranovou konzistenci.

Důkaz:

po prvním běhu DAC zajistíme, že pro každou hodnotu rodičovského vrcholu najdeme podporu (konzistentní hodnotu) u všech potomků



pokud je **při druhém běhu DAC** (v opačném pořadí) vyřazena nějaká hodnota, potom tato hodnota nebyla podporou žádné hodnoty rodičovského uzlu (tj. hodnoty v rodičovském uzlu neztratily své podpory)



dohromady: každá hodnota má podporu ve všech potomcích (první běh) i v rodiči (druhý běh), tj. jedná se o AC

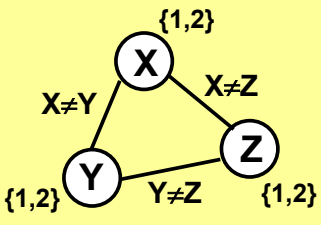
Je AC dostatečné?

Použitím AC odstraníme mnoho nekompatibilních hodnot.

- Dostaneme potom řešení problému?
- Víme alespoň zda řešení existuje?

  NE a NE!

Příklad:



CSP je hranově konzistentní a přesto nemá žádné řešení

K čemu tedy AC je?

- někdy **dá řešení přímo**
  - nějaká doména se vyprázdní → řešení neexistuje
  - všechny domény jsou jednoprvkové → máme řešení
- v obecném případě alespoň **zmenší prohledávaný prostor**