Programování s omezujícími podmínkami

Roman Barták
Katedra teoretické informatiky a matematické logiky

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

9

Jak splňovat podmínky?

- **Dosud dvě metody:**
  - **prohledávání prostoru řešení**
    - úplné (najde řešení nebo dokáže jeho neexistenci)
    - zbytečně pomalé (exponenciální)
      - prochází i „evidentně" špatná ohodnocení
  - **konzistenční techniky**
    - většinou neúplné (zůstávají nekonzistentní hodnoty)
    - relativně rychlé (polynomiální)
- Můžeme **využít výhod obou metod** - stačí je **kombinovat**.
  - postupně ohodnocujeme proměnné (backtracking)
  - po přiřazení hodnoty zajistíme konzistenci
- Nezapomínejme na tradiční techniky!
  - řešení soustav lineárních rovností, simplex …
  - můžeme integrovat v podobě globálních podmínek!

Backtracking

Golomb&Baumert (1965), Bitner&Reingold (1975)

Základní technika pro splňování omezujících podmínek:

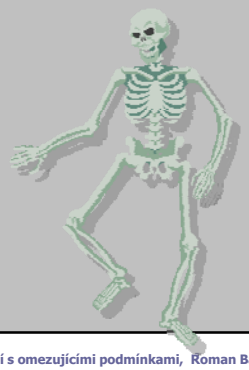
- postupně **ohodnocujeme proměnné**
  - proměnné očíslujeme a ohodnocujeme je v daném pořadí
- po vybrání hodnoty **testujeme konzistenci**

Kostra prohledávacího algoritmu

```
procedure Labelling(G)
    return LBL(G,1)
end Labelling

procedure LBL(G,cv)
    if cv>|nodes(G)| then return nodes(G)
    for each value V from D_cv do
        if consistent(G,cv) then
            R ← LBL(G,cv+1)
            if R ≠ fail then return R
        end if
    end for
    return fail
end LBL
```

„Hák" pro navěšení konzistenční procedury



Pohled zpět

look back

- Zajišťujeme **konzistenci mezi již ohodnocenými proměnnými**.
  - „zpět" = již ohodnocené proměnné
- **Co zjistí konzistence mezi již ohodnocenými proměnnými?**
  - **konflikt** (a případně jeho zdroj - nesplněnou podmínku)
- Backtracking je základní metoda pohledu zpět.

Test konzistence při backtrackingu

```
procedure AC-BT(G,cv)
    Q ← {(V_i,V_cv) in arcs(G),i<cv}      % hrany vedoucí z minulých prom.
    consistent ← true
    while not Q empty & consistent do
        select and delete any arc (V_k,V_m) from Q
        consistent ← not REVISE(V_k,V_m)
    end while
    return consistent
end AC-BT
```

Pokud vyřadíme prvek, bude doména prázdná

Backjumping a spol. využívají více informací z testu konzistence.

McGregor (1975),Van Hentenryck (1989)

Kontrola dopředu

forward checking

- **Lepší než odhalovat chyby je chybám předcházet!**
- Konzistenční techniky umožňují vyřazovat nekompatibilní hodnoty budoucích (=dosud neohodnocených) proměnných.
- Kontrola dopředu zajišťuje konzistenci mezi právě ohodnocenou proměnnou a proměnnými s ní spojenými podmínkou.

Algoritmus kontroly dopředu

```
procedure AC-FC(G,cv)
    Q ← {(V_i,V_cv) in arcs(G),i>cv}      % hrany vedoucí z budoucích prom.
    consistent ← true
    while not Q empty & consistent do
        select and delete any arc (V_k,V_m) from Q
        if REVISE(V_k,V_m) then
            consistent ← not empty D_k
        end if
    end while
    return consistent
end AC-FC
```

Vyprázdnění domény znamená nekonzistenci

Haralick&Elliot (1980)

(Částečný) pohled dopředu

partial look ahead

- **Proč kontrolovat jen přímé následníky, pojďme ještě dál!**
- Vybranou hodnotu proměnné můžeme propagovat do všech budoucích proměnných.

Algoritmus částečného pohledu dopředu

```
procedure DAC-LA(G,cv)
    for i=cv+1 to n do
        for each arc (V_i,V_j) in arcs(G) such that i > j & j ≥ cv do
            if REVISE(V_i,V_j) then
                if empty D_i then return fail
        end for
    end for
    return true
end DAC-LA
```

Poznámky:

- Vlastně děláme DAC (při obráceném uspořádání proměnných).
  - Partial Look Ahead neboli DAC - Look Ahead
- Není potřeba testovat konzistenci hran z budoucích do minulých proměnných jiných než aktuální proměnná!

Gaschnig (1974), Sabin&Freuder (1994)

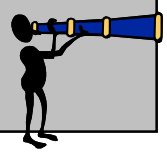
(Úplný) pohled dopředu

full look ahead

- **Kdo vidí dále do budoucnosti, je úspěšnější!**
- Proč dělat pouze DAC, když můžeme použít téměř plnou AC (např. AC3).

Algoritmus úplného pohledu dopředu

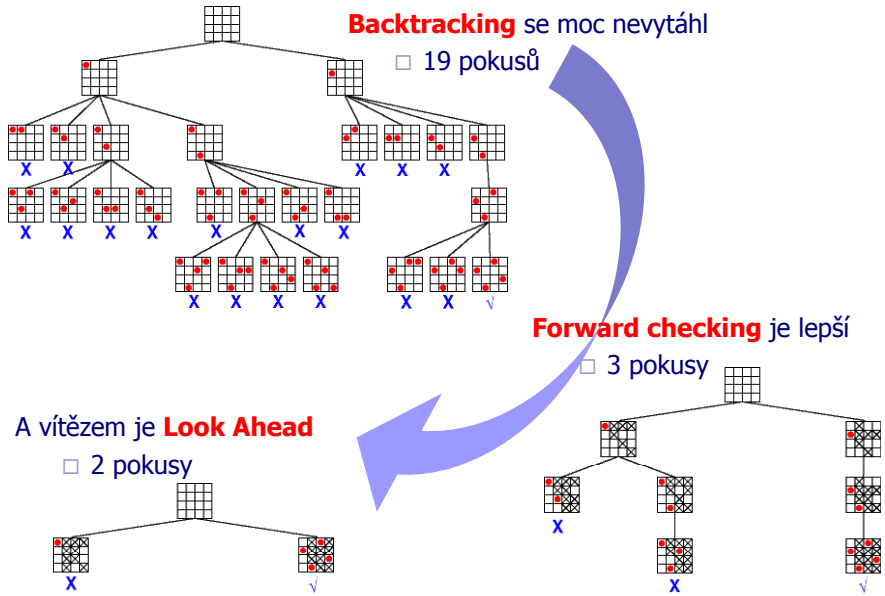
```
procedure AC3-LA(G,cv)
    Q ← {(V_i,V_cv) in arcs(G),i>cv}          % začínáme s hranami do cv
    consistent ← true
    while not Q empty & consistent do
        select and delete any arc (V_k,V_m) from Q
        if REVISE(V_k,V_m) then
            Q ← Q ∪ {(V_i,V_k) | (V_i,V_k) in arcs(G),i≠k,i≠m,i>cv}
            consistent ← not empty D_k
        end if
    end while
    return consistent
end AC3-LA
```



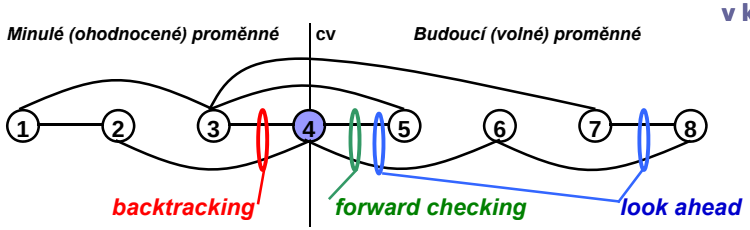
Poznámky:

- Hrany vedoucí do aktuální proměnné testujeme právě jednou.
- Hrany do jiných minulých proměnných už netestujeme.
- Můžeme použít jiné AC algoritmy (např. AC-4)

Co na to čtyři dámy?



Propagační algoritmy
v kostce



- Propagace přes více podmínek vyřadí více nekonzistencí (BT < FC < PLA < LA), za cenu větší složitosti každého kroku.
- **Forward Checking složitost backtrackingu příliš nezvýší** podmínka se při FC testuje dopředu (u BT se naopak testuje zpět).
- Použijeme-li AC-4 v LA, stačí provést jeho inicializaci pouze jednou.
- **Konzistenci můžeme zajistit ještě před startem prohledávání**
  - algoritmus MAC (Maintaining Arc Consistency)
    - provede AC před spuštěním prohledávání a potom po každém kroku
- Je možné používat i jiné než AC algoritmy (například na startu).

Uspořádání proměnných

Pořadí proměnných při ohodnocování výrazně ovlivňuje efektivitu výpočtu (viz stromové struktury).

Jak volit pořadí proměnných obecně?

Princip prvního neúspěchu (FAIL FIRST)

„vyber proměnnou, jejíž ohodnocení povede k neúspěchu"
- lepší je vypořádat se s neúspěchem dříve, později to bude těžší
- proměnné s menší doménou dříve (dynamické uspořádání) [**dom**]
  - větší pravděpodobnost nemožnosti vybrat správnou hodnotu
  - dynamické uspořádání je vhodné, jen pokud při řešení získáváme nějaké další informace (algoritmy pohledu dopředu)

„nejdříve vyřeš těžké případy, při odložení budou ještě těžší"
- proměnné s více podmínkami dříve [**deg**]
  - tyto proměnné je složitější ohodnotit (je možné zohlednit i složitost podmínky)
  - tato heuristika se používá při stejně velkých doménách [**dom+deg**]
- proměnné s více podmínkami s minulými proměnnými dříve
  - statická heuristika vhodná i pro prostý backtracking

Uspořádání hodnot

Pořadí, v jakém pro proměnnou vybíráme hodnotu, také ovlivňuje efektivitu (pokud vždy volíme správně, nemusíme se vracet).

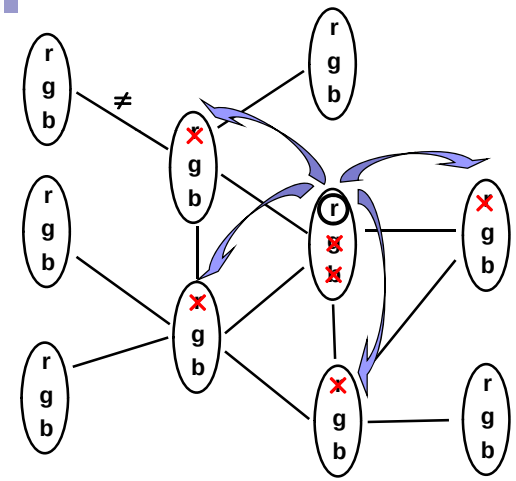
Jak volit pořadí hodnot pro proměnnou obecně?

Princip prvního úspěchu (SUCCEED FIRST)

„vyber hodnotu, která nejspíše patří do řešení"
- pokud to není žádná hodnota, na pořadí nezáleží (kontrolujeme vše)
- pokud taková hodnota existuje, je dobré ji najít dříve
- SUCCEED FIRST nejde proti FAIL FIRST výběru proměnné!
- **hodnota, která má nejvíce podpor**
  - tuto informaci lze získat například z AC-4
- **hodnota, jejíž výběr nejméně omezí ostatní proměnné**
  - lze získat například z jednotkové konzistence (součet/součin domén)
- **hodnota, která má nejvíce řešení relaxovaného problému**
  - počet řešení jednodušší varianty zbytku problému (např. strom)

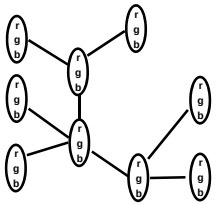
- Obecné heuristiky výběru hodnoty se zpravidla nevyplatí.
- **Nejlepší je pokud konkrétní problém dává preference pro výběr hodnoty!**

Trochu motivace



1) ohodnotíme proměnnou
2) zajistíme konzistenci
3) zbytek problému vyřešíme bez navracení

Jak to?



- Je-li CSP graf acyklický, umíme najít řešení bez navracení (stačí AC)!
- CSP grafy ale většinou nejsou acyklické!

Eliminace cyklů

Dechter&Pearl (1988)

Uděláme-li CSP graf acyklický, můžeme problém řešit bez navracení.

Jak se zbavit cyklů?

- stačí **ohodnotit proměnné na cyklech** (ohodnocení = vyřazení z grafu)
- dokonce stačí ohodnotit jen kružnicový řez (**cycle-cutset**)
- kružnicový řez = množina vrcholů, jejichž odstranění zruší všechny cykly

Algoritmus Cycle Cutset

```
procedure cycle-cutset(G)
    C ← find-cycle-cutset(G)
    while ex. assignment of variables in C satisfying all constraints do
        LC ← a(nother) assignment of variables in C satisfying all constraints
        enforce DAC from C to the remaining variables
        if all domains are non-empty then
            LR ← assignment of remaining variables (out of C) using backtrack-free
            search
            return LC+LR
        end if
    end while
    return fail
end cycle-cutset
```


Problémy eliminace cyklů

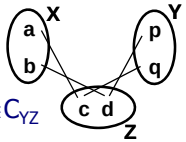
Při ohodnocování množiny cycle-cutset dochází k thrashingu.

Příklad 1:

- X,Y,Z - pořadí ohodnocování
- předpokládejme, že pro X=a není v Z kompatibilní hodnota
- pokaždé, když položíme X=a, dojde k chybě

lze řešit zajištěním konzistence před startem algoritmu

Příklad 2:



- X,Y,Z - pořadí ohodnocování
- (a,c), (b,d) $\in C_{XZ}$, (a,d) $\notin C_{XZ}$, (p,c) $\notin C_{YZ}$, (p,d), (q,c) $\in C_{YZ}$
- pokaždé, když položíme X=a, Y=p, dojde k chybě
- předem provedená hranová konzistence to neodhalí

potřebuje udržování konzistence v průběhu ohodnocování

Co třeba použít MAC ve spojení s technikou CC?

MAC Extended

Sabin&Freuder (1997)

- **méně ohodnocování**
  - ohodnocujeme (při prohledávání) jen malou množinu proměnných
- **méně propagace**
  - udržujeme pouze částečnou konzistenci (kterou lze ovšem rozšířit na plnou hranovou konzistenci)

Algoritmus MACE

```
1) zajisti hranovou konzistenci (pokud nelze, vrať neúspěch)
2) rozděl proměnné do dvou množin:
      množina C cycle-cutset
      množina U proměnných, které nejsou v žádném cyklu
            (poznamenejme, že U nemusí být doplňkem C)
3) odpoj U z grafu (v U nebudeme nadále zajišťovat konzistenci)
4) while C ≠ 0 do
      4a) zvol hodnotu pro proměnnou z C
      4b) zajisti hranovou konzistenci
            pokud nelze vrať se na 4a (nebo k předchozí proměnné)
      4c) odpoj proměnné s jednoprvkovou doménou (přidej je do U)
      4d) odpoj proměnné, které nejsou v žádném cyklu (přidej je do U)
5) znovu připoj proměnné z U
      a zajisti směrovou hranovou konzistenci do U
6) najdi úplně řešení prohledáváním bez navracení
```


Hledání kružnicového řezu

- CC i MACE potřebují množinu cycle-cutset (menší množina CC je lepší)
- Bohužel není znám polynomiální algoritmus pro hledání minimální CC.

Heuristiky:

- do CC dávej proměnné podle stupně (vyšší stupeň dříve)
- " + přidej pouze proměnné, které jsou v cyklu
- do CC dávej proměnné podle počtu cyklů, ve kterých se nachází

Algoritmus hledání cycle-cutset

```
procedure find-cycle-cutset(G)
    (V,E) = G
    Q ← order elements in V by descending order of their degrees in G
    CC ← {}
    while the graph G is cyclic do
        V ← first element in Q
        CC ← CC ∪ {V}
        Q ← Q - {V}
        remove V and edges involving it from the constraint graph G
    end while
    return CC
end find-cycle-cutset
```