Programování 9 omezujícími podmínkami

Roman Barták, KTIML

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

Co bylo minule

Konzistenční techniky v praxi
- událostmi řízené programování
- návrh propagace přes podmínky

Globální podmínky
- podmínka all-different

Prohledávání a konzistenční techniky
- Look Back schéma
  - backtracking, backjumping, backmarking
- Look Ahead schéma
  - forward checking
  - partial look ahead
  - (full) look ahead

Uspořádání proměnných

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

Jak volit pořadí proměnných obecně?

Princip prvního neúspěchu (FIRST-FAIL)

„vyber proměnnou, jejíž ohodnocení nejspíše povede k neúspěchu“

lepší je vypořádat se s neúspěchem dříve, později to bude těžší

- proměnné s menší doménou dříve (dynamické uspořádání)
  - větší pravděpodobnost nemožnosti vybrat správnou hodnotu
  - dynamické uspořádání je vhodné, jen pokud při řešení získáváme nějaké další informace (algoritmy pohledu dopředu)

„nejdříve řeš těžké případy, pokud je odložíš, budou ještě těžší“

- proměnné s více podmínkami dříve
  - tyto proměnné je složitější ohodnotit (je možné zohlednit i složitost podmínky)
  - tato heuristika se používá při stejně velkých doménách
- 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

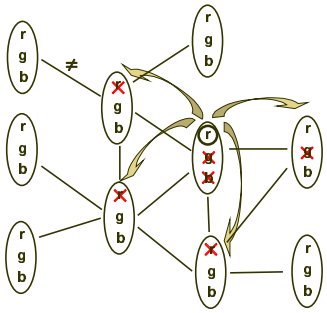
SUCCEED FIRST nejde proti FIRST-FAIL výběru proměnné!

- hodnota, která má nejvíc podpor dříve
  - tuto informaci lze získat například z AC-4
- hodnota, jejíž výběr nejméně omezí ostatní proměnné
  - tuto informaci lze získat například ze singleton konzistence
- hodnota, která vede k zjednodušení problému
  - vyřeš jednoduší variantu (např. strom) zbytku problému

Obecné heuristiky výběru hodnoty se zpravidla nevyplatí.

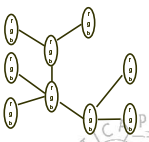
Nejlepší je pokud konkrétní problém dává preference pro výběr hodnoty!

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!

Algoritmus eliminace cyklů (CC)

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

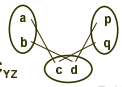
Problémy metody eliminace cyklů

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, dojte 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, dojte k chybě
předem provedená konzistence to neodhalí
potřebuje udržování konzistence v průběhu ohodnocování

Co třeba použít MAC ve spojení s technikou CC?

MAC Extended

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 úplné 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
```


Lokální prohledávání

Dosud: algoritmy rozšiřují částečné konzistentní ohodnocení na ohodnocení úplné (a konzistentní).

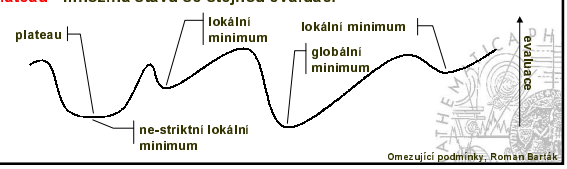
Metoda generuj a testuj: prochází úplná nekonzistentní ohodnocení dokud nenajde konzistentní ohodnocení.

Problém GT - generátor nepoužívá výsledku testu
Následující ohodnocení můžeme generovat na základě výsledku testu.

- budeme dělat jen lokální změny ohodnocení
- další ohodnocení je „lepší“ než předchozí
  lepší = splňuje více podmínek
- ohodnocení nejsou generována systematicky
  ztrácíme úplnost algoritmu
  výměnou za rychlost

Terminologie lokálního prohledávání

stav - ohodnocení všech proměnných
evaluace - hodnota objektivní funkce (počet nesplněných podmínek)
okolí - množina stavů lokálně se lišících od daného stavu (liší se v hodnotě jedné proměnné)
lokální optimum - stav, v jehož okolí není „lepší“ stav a zároveň není optimem
striktní lokální optimum - stav, který není optimem a v jehož okolí jsou pouze „horší“ stavy
ne-striktní lokální optimum - lokální optimum, které není striktní
globální optimum - stav, dosahující nejlepší evaluace
plateau - množina stavů se stejnou evaluací



Metoda největšího stoupání (HC)

Asi nejznámější metoda lokální prohledávání (hill climbing)
začíná v náhodně vybraném stavu
hledá vždy nejlepší stav v okolí
okolí = hodnota libovolné proměnné je změněna
velikost okolí = $\Sigma_{i=1..n}(D_i-1)$ (= n*(d-1))
„útěk“ ze striktního lokálního minima pomocí restartu

Algoritmus Hill Climbing

```
procedure hill-climbing(Max_Flips)
     restart: s ← random valuation of variables;
     for j:=1 to Max_Flips do                    % omezený počet kroků
          if eval(s)=0 then return s
          if s is a strict local minimum then
               go to restart
          else
               s ← neighbourhood with the smallest evaluation value
          end if
     end for
     go to restart
end hill-climbing
```

Metoda minimalizace konfliktů (MC)

Pozorování:

- okolí u hill-climbing je poměrně velké (n*(d-1))
- pouze změna konfliktní proměnné může přinést zlepšení

Metoda minimalizace konfliktů (min-conflicts)

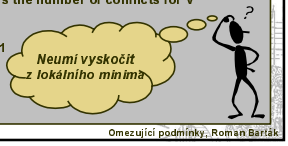
vybere náhodně konfliktní proměnnou a zkusí ji změnit k lepšímu

okolí = hodnota zvolené proměnné *i* je změněna

velikost okolí = (D_i-1) (= (d-1))

Algoritmus Min-Conflicts

```
procedure MC(Max_Moves)
    s ← random valuation of variables
    nb_moves ← 0
    while eval(s)>0 & nb_moves<Max_Moves do
        choose randomly a variable V in conflict
        choose a value v' that minimises the number of conflicts for V
        if v' ≠ current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end MC
```



Náhodná procházka (RW)

Jak se dostat z lokálního optima bez restartu (tj. lokálním krokem)?

Přidání „šumu“ do algoritmu!

Náhodná procházka (random walk)

stav z okolí je volen náhodně (resp. hodnota je volena náhodně)

tato metoda samostatně těžko povede k řešení

potřebujeme nějaké zacílení

Náhodná procházka je kombinována z heuristikou určující další tah pomocí pravděpodobnostního rozložení:

p - pravděpodobnost náhodného kroku

(1-p) - pravděpodobnost použití směrové heuristiky

Náhodná procházka a minimalizace konfliktů (MCRW)

MC vede algoritmus k cíli (splnění všech podmínek) a RW umožňuje vyskočení z lokálního minima.

Algoritmus Min-Conflicts-Random-Walk

```
procedure MCRW(Max_Moves,p)
    s ← random valuation of variables
    nb_moves ← 0
    while eval(s)>0 & nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose randomly a variable V in conflict
            choose a value v' that minimises the number of conflicts for V
        end if
        if v' ≠ current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end MCRW
```

$0.02 \le p \le 0.1$

Náhodná procházka a největší stoupání (SDRW)

V kombinaci s RW můžeme použít také HC heuristiku.

Potom není potřeba restart.

Algoritmus Steepest-Descent-Random-Walk

```
procedure SDRW(Max_Moves,p)
    s ← random valuation of variables
    nb_moves ← 0
    while eval(s)>0 & nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose a move <V,v'> with the best performance
        end if
        if v' ≠ current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end SDRW
```