Programování 8 omezujícími podmínkami

Roman Barták, KTIML

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

Co bylo minule

Zobecněné konzistenční techniky

- *k-konzistence*
  - rozšiřujeme (k-1) proměnných o další proměnnou
- *směrová k-konzistence*
- *konzistence a řešení bez navracení*
  - šířka grafu vs. stupeň konzistence
- *adaptivní konzistence*
  - různý stupeň konzistence v různých částech grafu
- *(i,j)-konzistence*
  - rozšiřujeme *i* proměnných o dalších *j* proměnných
- *inverzní konzistence*
  - NIC - hledáme řešení v okolí vrcholu
- *bodové konzistence*

Konzistenční (filtrační) techniky tvoří jádro CP!

Konzistenční techniky v praxi

S n-árními podmínkami se pracuje přímo!

Říkáme, že podmínka C_Y je hranově konzistentní, právě když pro každou proměnnou *i* z této podmínky a každou hodnotu $v \in D_i$ existuje ohodnocení zbylých proměnných v podmínce tak, že podmínka platí.

Příklad: A+B=C, A in 1..3, B in 2..4, C in 3..7 je AC

Využívá se sémantika podmínek!

intervalová konzistence
- nepracujeme s individuálními hodnotami, ale s intervaly hodnot
- intervalová aritmetika
- *Příklad:* po změně A počítáme A+B → C, C-A → B

konzistence okrajů/mezí (bounded consistency)
- pracujeme pouze s dolní a horní mezí domény

tyto techniky obecně nedávají plnou konzistenci

Pro různé podmínky lze použít různý druh konzistence!

Základní konzistenční algoritmus

Základem je obecná verze AC-3.

Opakovaně se provádí revize podmínek, dokud se mění domény.

```
procedure AC-3(C)
    Q ← C                    % seznam podmínek pro revizi
    while Q non empty do
        select and delete c from Q
        REVISE(c,Q)
    end while
end AC-3
```

Procedury REVISE „šité na míru“ jednotlivým podmínkám.

různé konzistenční algoritmy

Plánování podmínek

Jak určit pořadí podmínek pro revize (fronta Q)?

Programování řízené událostmi
- událost = změna domény
- REVIZE přidá další události a ty vyvolají zasažené podmínky

Návrh konzistenčních algoritmů

Uživatel má často možnost definovat vlastní REVISE kód

Jak se to dělá?

1) Je třeba určit událost, která kód vyvolá při změně domény proměnné (tzv. suspensions)
- kdykoliv se změní doména
- změna maxima či minima (nebo obecně okraje)
- navázání proměnné (vybrání hodnoty)

lze použít různé suspensions pro různé proměnné

Příklad: A<B propagace se spouští pro min(A) a max(B)
- směrová konzistence

2) Je třeba navrhnout propagaci přes podmínku

výsledkem propagace je omezení domén proměnných

pro jednu podmínku lze mít více propagačních kódů

Příklad: A<B

min(A): B in min(A)+1..sup, max(B): A in inf..max(B)-1

Definice podmínky (SICStus Prolog)

Jak se definuje propagace u podmínky A<B?

pro plnou konzistenci stačí propagace mezí!

```
less_then(A,B):-
  fd_global(a2b(A,B),no_state,[min(A)]),
  fd_global(b2a(A,B),no_state,[max(B)]).

dispatch_global(a2b(A,B),S,S,Actions):-
  fd_min(A,MinA), fd_max(A,MaxA), fd_min(B,MinB),
  (MaxA<MinB ->
      Actions = [exit]
  ;   LowerBoundB is MinA+1,
      Actions = [B in LowerBoundB..sup]).

dispatch_global(b2a(A,B),S,S,Actions):-
  fd_max(A,MaxA), fd_min(B,MinB), fd_max(B,MaxB),
   (MaxA<MinB ->
      Actions = [exit]
  ;   UpperBoundA is MinB-1,
      Actions = [B in inf..UpperBoundA]).
```

Globální podmínky

Propagace je lokální
- pracuje se s jednotlivými podmínkami
- interakce mezi podmínkami je pouze přes domény proměnných

Můžeme 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.

Příklad:

all_different([A,B,C,D])

A≠B, A≠C, A≠D, B≠C, B≠D, C≠D

exactly(X,[A,B,C,D]),Y)

A=Y⇔ A1=1, B=Y⇔ B1=1, C=Y⇔ C1=1, D=Y⇔ D1=1

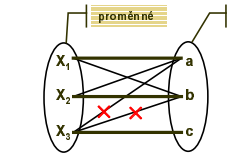
A1,B1,C1,D1 in 0..1

A1+B1+C1+D1=X

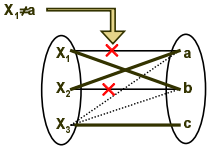
Podmínka alldifferent

all_different(X_1,…, X_k) = {(d_1,…, d_k) | ∀i $d_i \in D_i$ & ∀i≠j $d_i \neq d_j$}

Efektivní propagaci lze založit na párování v bipartitních grafech hodnot (Regin).



Inicializace:
1) vypočti maximální párování
2) odstraň všechny hrany, které nepatří do žádného maximálního párování



Propagace zmenšené domény:
1) odstraň odpovídající hrany
2) vypočti nové maximální párování
3) odstraň všechny hrany, které nepatří do žádného maximálního párování

Jak řešit omezující 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!

Základem je prohledávání s navracením

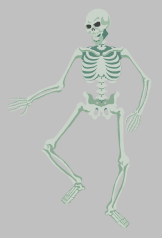
Základní technika pro splňování omezujících podmínek:
- postupně ohodnocujeme proměnné
  - pro jednoduchost proměnné očíslujeme a ohodnocujeme je v daném pořadí
- po vybrání hodnoty testujeme konzistenci

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 do 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.

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 do 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

(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 (Vi,Vj) in arcs(G) such that i>j & j≥cv do
            if REVISE(Vi,Vj) then
                if empty Di 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!

(Úplný) pohled dopředu (full look ahead)

Kdo vidí dále do budoucnosti, je úspěšnější!

Proč dělat pouze DAC, když můžeme použít plnou AC (např. AC3).

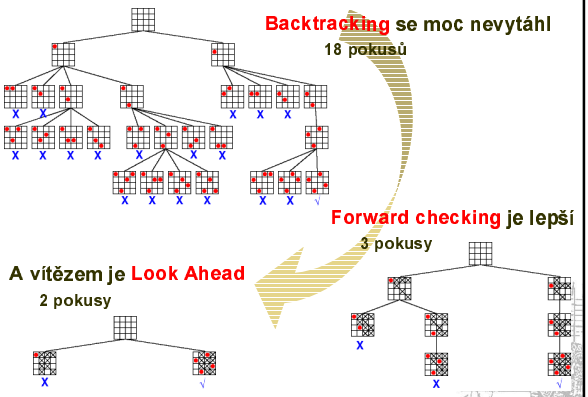
Algoritmus úplného pohledu dopředu

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

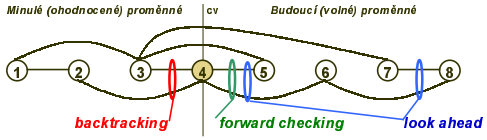
Poznámky:

- 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), samozřejmě to znamená větší složitost jednoho 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 (Maintaing 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 (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!