Programování s omezujícími podmínkami

Roman Barták, KTIML

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

4

Co bylo minule

Pokročilejší prohledávací algoritmy

Backjumping
- odstranění thrashingu analýzou konfliktů

Dynamický backtracking
- uchování ohodnocení změnou pořadí proměnných

Backmarking
- odstranění redundance pomocí paměti konfliktů

A co na to heuristiky?
→ *Limited Discrepancy Search*

Nelze podmínky používat aktivněji?
→ *Konzistenční techniky*

Stromové prohledávání a heuristiky

Pozorování 1:
Praktické problémy mají tak velké prostory možných ohodnocení, že není možné je celé prohledat.

***Heuristiky* - rádce kudy se při prohledávání dát**
- doporučují hodnotu pro přiřazení
- často vedou přímo k řešení

Co dělat, když heuristika neuspěje?
BT se stará se hlavně o konec prohledávání (spodní část stromu)
- spíše tedy opravuje poslední použité heuristiky než první
- předpokládá, že prvně použité heuristiky ho navedly dobře

Pozorování 2:
Heuristiky jsou méně spolehlivé na začátku prohledávání než na jeho konci (na konci máme více informací).

Pozorování 3:
Počet porušení heuristiky na úspěšné cestě je malý.

Limited Discrepancy Search

Discrepance = porušení heuristiky
(vybereme jinou hodnotu, než doporučila heuristika)

Idea *Limited Discrepancy Search* (LDS):
- nejprve jdeme tak, jak doporučuje heuristika
- pokud neuspějeme, prozkoumáme cesty, kde je heuristika max. jednou porušena (nejprve ji porušíme u prvních vrcholů)
- pokud opět neuspějeme, prozkoumáme cesty, kde je heuristika porušena maximálně dvakrát ...

Příklad:
heuristika doporučuje vydat se levou větví

Algoritmus LDS

```
procedure LDS-PROBE(Unlabelled,Labelled,Constraints,D)
    if Unlabelled = {} then return Labelled
    select X in Unlabelled
    Values_X ← D_X - {values inconsistent with Labelled using Constraints}
    if Values_X = {} then return fail
    else select HV in Values_X using heuristic
        if D=0 then return LDS-PROBE(Unlabelled-{X}, Labelled∪{X/HV}, Constraints, 0)
        for each value V from Values_X -{HV} do
            R ← LDS-PROBE(Unlabelled-{X}, Labelled∪{X/V}, Constraints, D-1)
            if R≠ fail then return R
        end for
        return LDS-PROBE(Unlabelled-{X}, Labelled∪{X/HV}, Constraints, D)
    end if
end LDS-PROBE

procedure LDS(Variables,Constraints)
    for D=0 to |Variables| do          % D určuje max. počet povolených diskrepancí
        R ← LDS-PROBE(Variables,{},Constraints,D)
        if R≠ fail then return R
    end for
    return fail
end LDS
```


Úvod do konzistenčních technik

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 (NC)

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)
        for each value V in the domain D_X
            if unary constraint on X is inconsistent with V then
                        delete V from D_X
        end for
    end for
end NC
```


Hranová konzistence (AC)

Nadále se budeme zabývat jen binárními CSP

tedy podmínka odpovídá hraně v grafu podmínek.

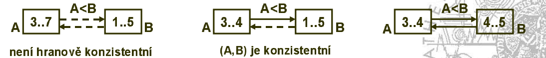
Definice:

- *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) je hranově konzistentní (v obou směrech).

Příklad:



Algoritmus 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 D_i do
        if there is no such Y in D_j such that (X,Y) is consistent, i.e.,
                (X,Y) satisfies all the constraints on V_i, V_j then
            delete X from D_i
            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
        end for
    until not(CHANGED)
end AC-1
```

Co je neefektivního na AC-1?

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

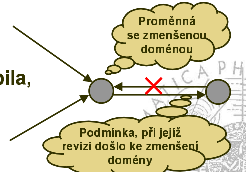
Jaké hrany se mají tedy zrevidovat po zmenšení domény?

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 je 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 }
            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
            Q ← Q ∪ {(i,k) | (i,k)∈arcs(G), i≠k, i≠m}
        end if
    end while
end AC-3
```

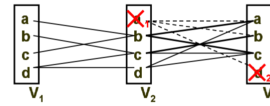
Technika AC-3 je dnes asi nejpoužívanější, ale pořád není optimální

Hledání (a pamatování si) podpor

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_2, zda některá z hodnot *a,b,c,d* neztratila svoji podporu ve V_1.

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> | $b \in D_j$, $(a,b) \in C_{i,j}$}

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

Algoritmus 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
```

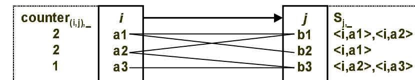
Sj,b - množina dvojic <i,a>, pro které je <j,b> podporou

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

Počítání podpor a jak je využít

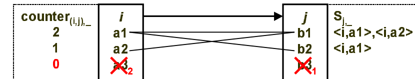
Situace:

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 Sj,b do
            counter[(i,j),a] ← counter[(i,j),a] - 1
            if counter[(i,j),a] = 0 & "a" is still in Di then
                delete "a" from Di
                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 (Bessiere, Cordier)

- zlepšuje paměťovou náročnost a průměrný čas AC-4
- drží si pouze jednu podporu, další podporu hledá až při ztrátě aktuální podpory

AC-7 (Bessiere, Freuder, Regin)

- založen na podporách (jako AC-4 a AC-6)
- využívá „dvousměrnosti“ podmínky