Programování
s omezujícími podmínkami

Roman Barták
Katedra teoretické informatiky a matematické logiky

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

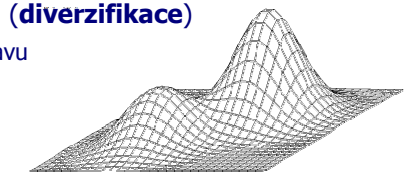
3

Lokální prohledávání

- **prochází úplná nekonzistentní ohodnocení dokud nenajde konzistentní ohodnocení**
  - oproti metodě „generuj a testuj" je další kandidát na řešení generován na základě výsledku testu splnění podmínek – cíleně se snažíme splnit více podmínek

Algoritmy lokálního prohledávání definují

- **okolí** současného stavu (ohodnocení proměnných) a metodu **výběru stavu** z okolí (**intenzifikace**)
  - HC heuristika – výběr nejlepšího ohodnocení lišícího se v jedné proměnné
    - někdy se bere první lepší ohodnocení
  - MC heuristika – výběr nejlepšího ohodnocení lišícího se ve vybrané konfliktní proměnné
- metodu **úniku z lokálního optima** (**diverzifikace**)
  - restart – začneme ve zcela jiném stavu
  - RW – vybereme stav náhodně
  - Tabu – zakážeme opakování kroků



SAT a lokální prohledávání

Řadu problémů lze formulovat v podobě SAT
= **splnitelnost logické formule** v konjunktivní normální formě
- **CNF** = konjunkce klauzulí
- **klauzule** = disjunkce literálů (podmínka)
- **literál** = atom nebo negace atomu

Příklad:

$(A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee \neg A)$

- Podobně jako CSP je SAT také NP-úplný problém, nelze tedy očekávat rychlý úplný algoritmus.
- Lokálním prohledáváním lze řešit poměrně velké formule.

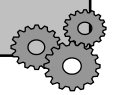
Poznámky:
- splnitelnost formule v disjunktivně normální formě lze řešit v lineárním čase
- SAT je speciálním případem CSP a naopak, CSP lze převést na SAT

Algoritmus GSAT

- GSAT řeší SAT problém postupným „překlápěním" proměnných.
- Cílem je maximalizovat (vážený) počet splněných klauzulí.

Algoritmus GSAT

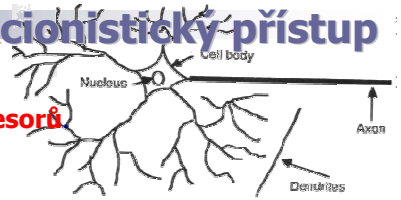
```
procedure GSAT(A,Max_Tries,Max_Moves)
    A: is a CNF formula
    for i:=1 to Max_Tries do
        S ← random assignment of variables
        for j:=1 to Max_Moves do
            if A satisfiable by S then return S
            V ← the variable whose flip yield the most important raise
                in the number of satisfied clauses
            S ← S with V flipped
        end for
    end for
    return the best assignment found
end GSAT
```



GSAT a heuristiky

- GSAT lze kombinovat s různými heuristikami, které zvyšují jeho efektivitu (zvláště při řešení strukturovaných problémů):
- **Random-Walk**
  - stejně jako u MCRW
- **Vážení klauzulí**
  - vychází z pozorování, že některé klauzule zůstávají po řadu iterací nesplněné, klauzule tedy nemají v problému stejnou důležitost
  - splnění „těžké" klauzule lze preferovat přidáním váhy
  - váhu může systém sám odvodit
    - na začátku mají všechny klauzule stejnou váhu
    - po každé iteraci je zvětšena váha u nesplněných klauzulí
- **Průměrování řešení**
  - standardně každý pokus začíná z náhodného ohodnocení, tj. „zapomene" se již získané řešení
  - lze zachovat společné části předchozích řešení
    - restartovací stav se vypočte ze dvou posledních výsledků bitovým srovnáním (stejné bity jsou zachovány, ostatní nastaveny náhodně)

Konekcionistický přístup



- Založen na myšlence reprezentovat problém jako **síť jednoduchých procesorů**
  - procesory nabývají různých stavů (typicky on a off).
  - následující stav procesoru je odvozen od stavu připojených procesorů (roli hraje i síla spojení).
- Cílem je najít stabilní stav celé sítě, tj. procesory nejsou nuceny změnit svůj stav.
- Tento stabilní stav reprezentuje řešení problému.

Vlastnosti:

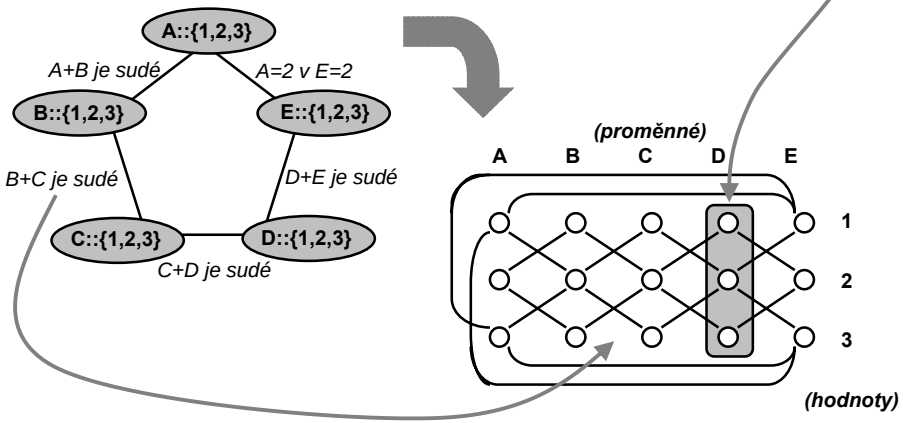
- možnost masivního paralelismu (tj. rychlé řešení problémů)
- černá skříňka (není zcela jasné, co se děje uvnitř)

- asi nejznámějším reprezentantem jsou umělé **neuronové sítě** (NN)
- ale na podobném principu pracují i **celulární automaty**

GENET - CSP jako NN

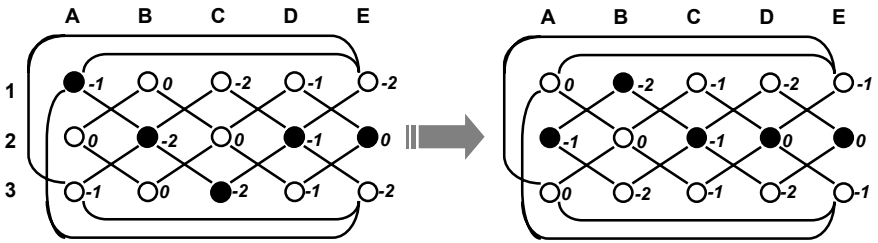
- Proměnné s doménou odpovídá cluster „neuronů" (pro každou hodnotu jeden neuron)
- Dva neurony jsou spojeny inhibiční hranou s negativní váhou pokud jsou příslušné hodnoty nekompatibilní (binární CSP).

Příklad:



Počítání s GENETem

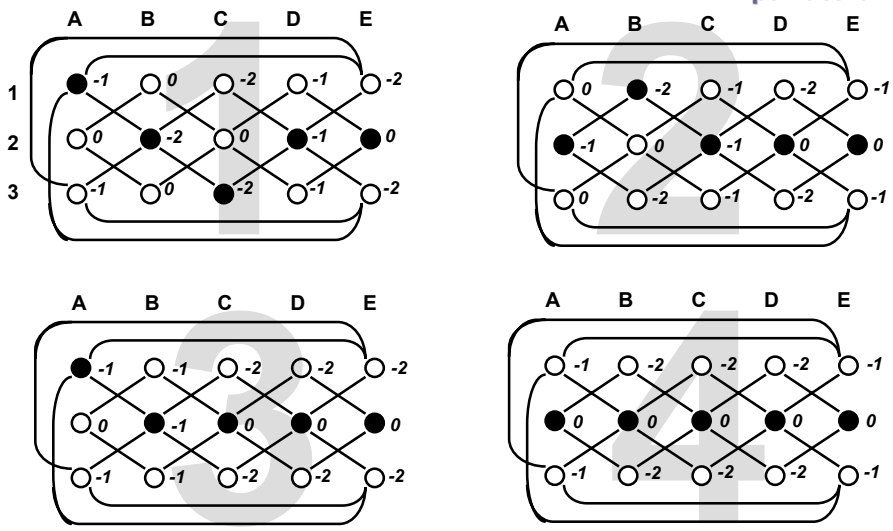
- Na začátku je v každém clusteru **vybrán jeden aktivní** neuron.
- Neurony mění své stavy **synchronně** (všechny najednou)
  - na základě svého vstupu (Σ w*s – vážený součet stavů z okolí)
  - v každém clusteru se vybere neuron s největším vstupem
- Výpočet končí ve **stabilním stavu**.



● = „zapnutý" neuron; čísla indikují vstup neuronu

Počítání s GENETem

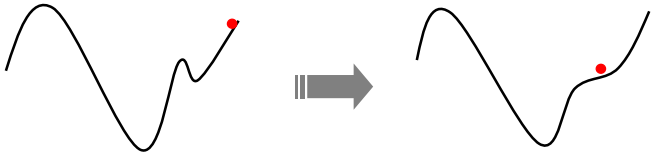
pokračování



● = „zapnutý“ neuron; čísla indikují vstup neuronu

Útěk z lokálního optima

- Co dělat ve stabilním stavu, který není koncový?
  - Dosud jsme používali restart nebo „šum“.
  - Můžeme ale také **změnit prostor ohodnocení**.
    - Jak? Změnou hodnotící funkce!
      - **dynamické lokální prohledávání**

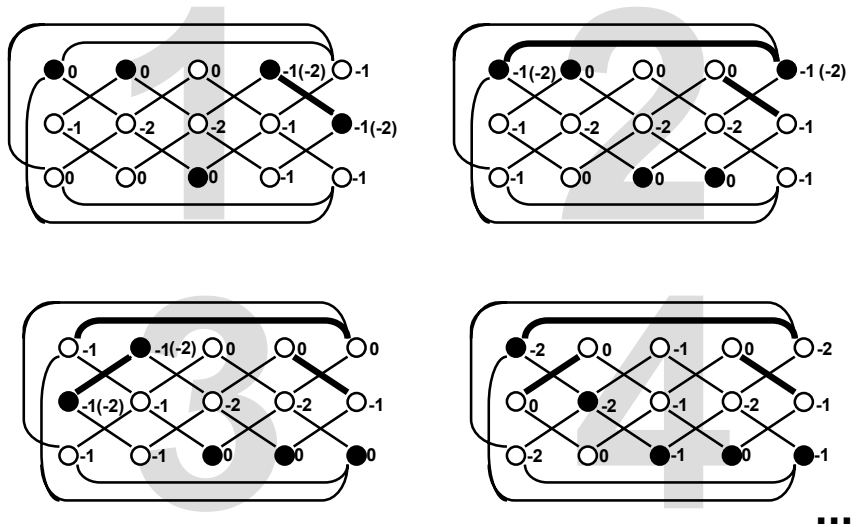


U GENETu můžeme **měnit váhy spojnic**!

- Pokud hrana spojuje dva aktivní neurony (= porušená podmínka), posílí svojí váhu.
  - nová váha$_{x,y}$ = stará váha$_{x,y}$ – s_x* s_y
- Tím vlastně měníme evaluační funkci (**Guided Local Search**).

Posilování vah u GENET

V lokálním minimu se **posílí váhy** vybraných hran (což poruší rovnováhu).



Algoritmus GENET

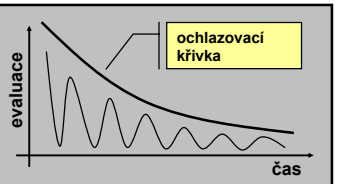
```
procedure GENET(connectionist network)
    one arbitrary node per cluster is switched on;
    repeat
        repeat  % network convergence
            modified <- false;
            for each cluster C do in parallel
                on_node <- currently switched on node in cluster C;
                label_set <- the set of nodes in C which input are maximum;
                if on_node is not in label_set then
                    switch off on_node;
                    modified <- true;
                    switch on arbitrary node in label_set;
                end if
            end for
        until not modified
        if sum of input to all switched-on nodes < 0 then
            for each connection c connecting nodes x & y do in parallel
                if both x & y are switched on then
                    decrease the weight of c by 1;
            end for
        end if
    until input to all switched-on nodes are 0
end GENET
```

Simulované žíhání (SA)

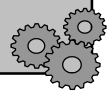
- Založeno na myšlence **simulace procesu ochlazování kovů**.
  - Při vyšší teplotě atomy více kmitají a pravděpodobnost změny krystalické mřížky je vyšší.
  - Postupným ohlazování se atomy usazují do „nejlepší" polohy – polohy s nejmenší energií.
- Stejný proces lze aplikovat v optimalizačních algoritmech
  - tzv. **simulované žíhání** (simulated annealing):
    - začneme s náhodným stavem
    - lokální krok je akceptován jen pokud
      - vede ke zlepšení stavu
      - vede ke zhoršení stavu (v tomto případě je krok přijat jen s určitou pravděpodobností odvozenou od aktuální „teploty")
    - „teplota" je postupně snižována, tj. snižuje se pravděpodobnost zhoršení stavu dle tzv. **ochlazovací křivky**

Algoritmus SA

```
procedure SA(InitT, MinT, MaxMoves)
   s ← random assignment of variables
   best ← s
   T ← InitT
   while MinT<T do
      num_errors ← 0
      while num_error<MaxMoves do
         next_s ← a random local change of s
         if eval(next_s) < eval(s) then
                  s ← next_s
                  if eval(s) < eval(best_s) then best ← s
         else
                  p ← random number in [0,1)
                  if p < e^(eval(s)-eval(next_s))/T then
                     s ← next_s
                  else
                     num_errors ← num_errors+1
      end while
      T ← 0.8 x T
   end while
   return best
end SA
```



Metropolisova heuristika

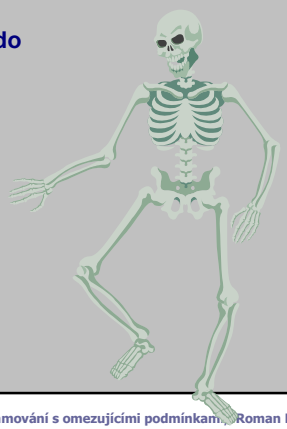


Localizer

- Prezentované algoritmy lokálního prohledávání mají společnou kostru, kterou lze doplnit procedurami realizujícími konkrétní algoritmus.

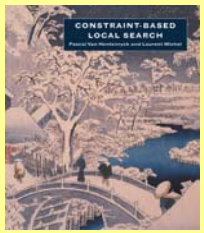
Algoritmus Local Search

```
procedure local-search(Max_Tries,Max_Moves)
      s ← random valuation of variables
      for i:=1 to Max_Tries while Gcondition do
            for j:=1 to Max_Moves while Lcondition do
                  if eval(s)=0 then
                              return s
                  end if
                  select n in neighbourhood(s)
                  if acceptable(n) then
                              s ← n
                  end if
            end for
            s ← restartState(s)
      end for
      return best s
end local-search
```



Shrnutí lokálního prohledávání

- **Algoritmy lokálního prohledávání začínají v nějakém stavu a přechodem do stavu z okolí se snaží najít cílový stav.** Obecně jsou definovány
  - volbou okolí a v něm přípustných stavů
  - heuristikou výběru lepšího stavu z okolí (intenzifikace)
  - meta-heuristikou pro vyskočení z lokálního optima (diverzifikace)



www.comet-online.org

Z Lokalizeru vychází systém **Comet** (MaxOS X, Linux, Win), který umožňuje popisovat algoritmy lokálního prohledávání deklarativním způsobem

Zpět ke GT

Pro připomenutí:

- metoda generuj a testuj otestuje splnění podmínek až po vygenerování kandidáta na řešení (úplného ohodnocení proměnných)
- kandidáty generujeme systematicky

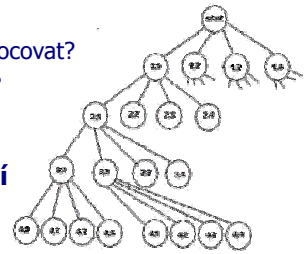
```
procedure generate_first(Variables)
    for each V in Variables do
        label V by the first value in D_V
    end for
end generate_first
procedure generate_next(Assignment)
    find first X in Assignment such that all following variables are labelled by the last value
    from their respective domains (name the set of these variables Vs)
    if X is labelled by the last value then return fail
    label X by next value in D_X
    for each Y in Vs do
        assign first value in D_Y to Y
    end for
end generate_next
```

- Splnění podmínky můžeme otestovat, jakmile známe hodnoty proměnných v podmínce!
  - fázi testování v GT děláme zároveň s generováním

Backtracking

- Asi nejběžnější algoritmus systematického prohledávání, který **testuje podmínky ihned, jak je to možné**.
  - při neúspěchu (nějaká podmínka nelze splnit) se vrátí k poslední ohodnocené proměnné a zkusí jinou hodnotu
  - prohledávání do hloubky s navracením
- Základní princip prohledávání s navracení při řešení CSP:
  1. postupně ohodnocuj proměnné
  2. po každém ohodnocení otestuj podmínky, jejichž všechny proměnné již mají hodnotu

  Otevřené otázky (které zodpovíme později):
  - v jakém pořadí se mají proměnné ohodnocovat?
  - v jakém pořadí se mají zkoušet hodnoty?
- **Algoritmus prohledávání s navracením postupně prochází částečná konzistentní ohodnocení, dokud nenajde úplné (konzistentní) ohodnocení.**

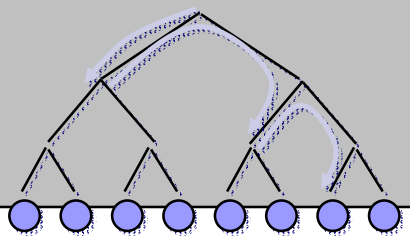


Chronologický backtracking

rekurzivně

```
procedure BT(X:variables, V:assignment, C:constraints)
    if X={} then return V
    x ← select a not-yet assigned variable from X
    for each value h from the domain of x do
        if constraints C are consistent with V ∪ {x/h} then
            R ← BT(X – {x}, V ∪ {x/h}, C)
            if R ≠ fail then return R
    end for
    return fail
end BT
```

volá se BT(X, {}, C)



Poznámka:

Pokud můžeme test provést nad částečně vygenerovaným kandidátem řešením, je BT vždy lepší než GT, protože nemusí procházet všechny kandidáty.

Chronologický backtracking

iterativně

```
procedure BT(X:variables, C:constraints)
    i ← 1, D'_i ← D_i
    while 1 ≤ i ≤ n do
        instantiate_and_check(i, C)
        if x_i = null then i ← i – 1 else i ← i + 1, D'_i ← D_i
    end while
    if i = 0 then return fail
    return {x_1,..., x_n}
end BT
```

```
procedure instantiate_and_check(i, C:constraints)
    while D'_i is not empty do
        select and delete some element b from D'_i
        x_i ← b
        if constraints C are consistent with {x_1,..., x_i} then return
    end while
    x_i ← null
end instantiate_and_check
```