Umělá inteligence I

Roman Barták, KTIML

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

5

Na úvod

- **Minule** jsme si řekli, jak využívat heuristiky v prohledávání a jak konstruovat heuristiky
  - BFS, A*, IDA*, RBFS, SMA*
  - relaxace problému, databáze vzorů
- **Dnes**:
  - **Co když na cestě nezáleží?**
    - lokální prohledávání: HC, SA, BS, GA
  - **Co když se svět mění?**
    - on-line prohledávání, LRTA*

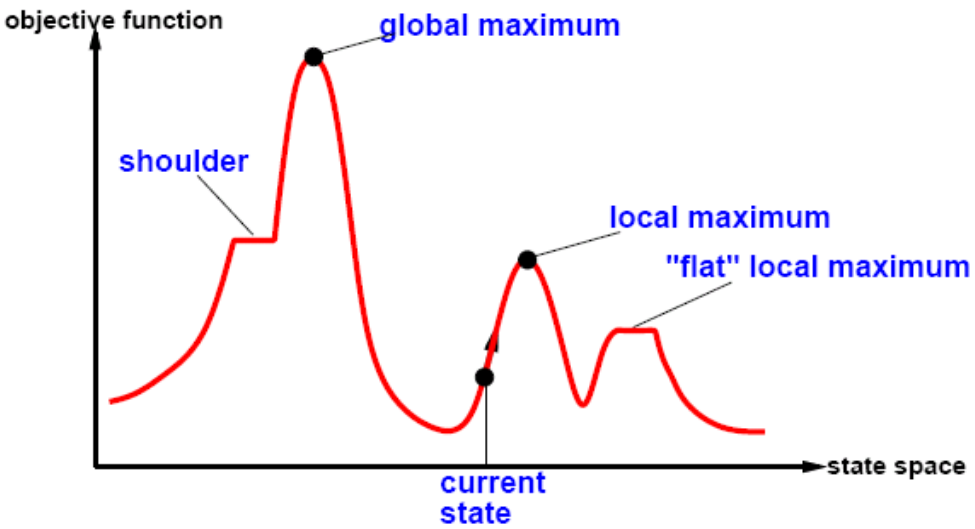
Lokální prohledávání

- Dosud jsme systematicky prohledávali cesty vedoucí do cílového stavu a nalezená cesta potom byla součástí řešení.
- U některých problémů ale není cesta relevantní, důležitý je nalezený cíl (např. 8-královen).
- V takovém případě můžeme zkusit tzv. **lokální prohledávání**.
  - pracuje s jedním stavem (konstantní paměť)
  - v každém kroku tento stav „trochu" změní na jiný stav
  - zpravidla si nepamatuje prošlou cestu
  - umí hledat i **optimální stav**, kde optimalita je definovaná objektivní funkcí (na stavech)
    - Např. u problému 8-královen může být objektivní funkcí (pro minimalizaci) počet konfliktních dvojic – informace o problému.

Lokální prohledávání

základní pojmy

- Lokální prohledávání je často definováno pohybem v **krajině**, kde souřadnice pozice určují stav a výška definuje hodnotu objektivní funkce v daném stavu.



Horolezecká metoda

- Z okolí daného stavu vždy vybereme stav, který má nejlepší objektivní funkci a do tohoto stavu přejdeme (**metoda největšího stoupání**).
  - vidí pouze okolí daného stavu
  - předchozí stav okamžitě zapomíná

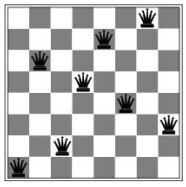
- **počet konfliktů = 17**
- **změna stavu = změna řádku jedné královny**
- **náhodný výběr mezi více „nejlepšími" soused**


```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node
   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```


Horolezecká metoda

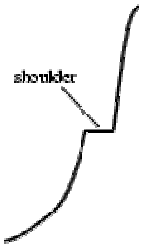
problémy

HC je **hladový algoritmus** – jde za nejlepším sousedem bez pohledu dále dopředu



- **lokální optimum** (stav, kde každý pohyb vede do horšího stavu)
  - HC z něho neumí uniknout

- **hřebeny** (posloupnost lokálních optim)
  - velmi komplikované pro hladové algoritmy



- **plošiny** (plateau - oblast se stejně dobrými stavy)
  - rameno – typ plošiny, ze které lze uniknout
  - HC se nemusí podařit najít cestu (cyklus)

Horolezecká metoda

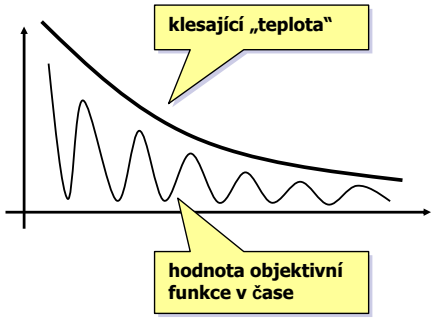
varianty

- **stochastický HC**
  - mezi zlepšujícími sousedy vybírá náhodně s pravděpodobností danou velikostí zlepšení
  - nemusí se vydat do nejlepšího souseda (pomalejší konvergence)
  - pro některé problémy dává lepší řešení
- **HC první volby**
  - náhodně generuje sousedy dokud nenajde lepší stav a do něj se vydá
  - vhodné, pokud je velké množství sousedů
- **HC s náhodnými restarty**
  - ocitne-li se v lokálním optimu, začne prohledávat znova z náhodného stavu (restart)
  - umožňuje únik z lokálního optima
  - při pravděpodobnosti p, že cesta vede k řešení, je očekávaný počet restartů 1/p
  - velmi efektivní metoda pro řešení N-královen (p ≈ 0,14 tj. 7 iterací)

Simulované žíhání

- **HC** se nikdy nevydá dolů (proti objektivní funkci), proto není úplný (nepřekoná lokální optima).
- **Náhodná procházka** (random walk), která vybírá stav z okolí zcela náhodně, je úplná, ale pomalá.
- **Simulované žíhání** kombinuje výhody obou metod
  - motivace v metalurgii – postupné ochlazování umožňuje lepší usazení atomů do krystalické mřížky
  - algoritmus náhodně volí stav z okolí, do které se vydá pokud:
    - je lepší než aktuální stav
    - je horší než aktuální stav, ale zhoršení je povoleno s určitou mírou pravděpodobnosti odvozenou od aktuální teploty; teplota se přitom postupně snižuje podle **„ochlazovacího" schématu**
  - stav se v krajině pohybuje podobně jako skákající míček

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps
  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```



Paprsky

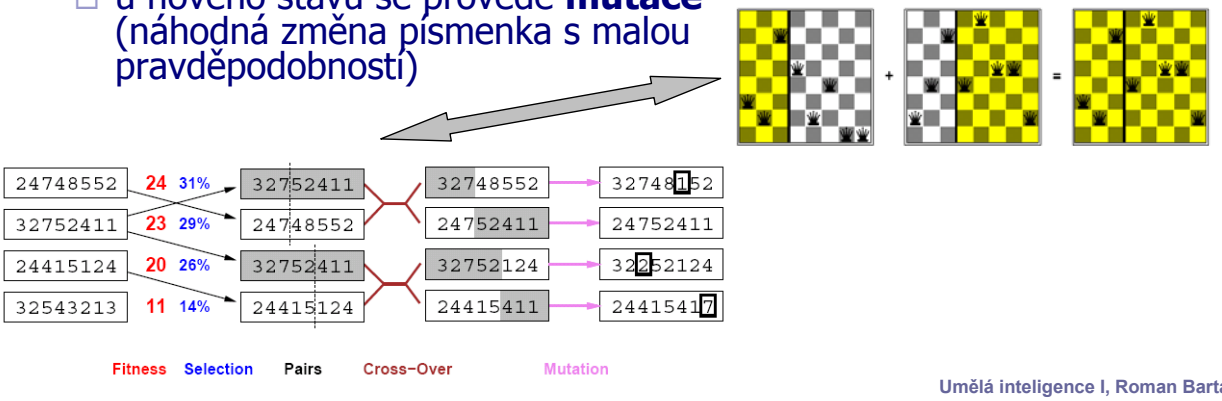
Představené algoritmy lokálního prohledávání mají extrémně malé paměťové nároky (jeden stav).

Nešlo by využít dostupnou paměť lépe?

- **Algoritmus lokálních paprsků**
  - začíná s k náhodnými stavy
  - v každém kroku najde všechny jejich sousedy
    - je-li mezi nimi cílový stav, potom končí
  - ze sousedů vybere k nejlepších pro další krok
- **Pozor! To není paralelní simulace k restartů HC!**
  - zde se předává informace mezi jednotlivými paprsky
  - vybraná k-tice stavů mohou být např. sousedé jediného stavu
  - algoritmus se tak soustředí na prozkoumání slibné oblasti
  - někdy tak ale může přijít o rozmanitost prozkoumaných stavů
    - lze řešit stochastickou verzí (náhodný výběr stavů s pravděpodobností danou kvalitou stavu)
    - trochu připomíná přirozený výběr druhů dle Darwina

Genetické algoritmy

- Varianta stochastického prohledávání s paprsky – kombinují se dva stavy (sexuální reprodukce)
  - začneme s k náhodnými stavy – **populace**
    - stav je reprezentován řetězcem symbolů v konečné abecedě (jako DNA)
    - **fitness** funkce určuje kvalitu stavu (dle objektivní funkce)
  - vyberou se dvojice stavů pro reprodukci (pravděpodobnost výběru dána fitness funkcí)
  - pro každý pár se určí bod přechodu (**crossover**), který rozdělí řetězce popisující stavy
  - komplementární řetězce se spojí do nového stavu
  - u nového stavu se provede **mutace** (náhodná změna písmenka s malou pravděpodobností)



Genetické algoritmy
struktura

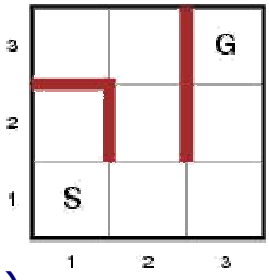
```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    loop for i from 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```


Offline vs. online

- Dosud probírané **offline prohledávání**
  - nejprve najde kompletní řešení
  - potom aplikuje řešení aniž by uvažovalo vjemy
- **Online prohledávání** naopak
  - prolíná hledání řešení a provádění akcí
    - vybere akci
    - provede akci
    - zjistí jak vypadá svět
    - vybere další akci
  - vhodné pro dynamické a semi-dynamické prostředí
  - vhodné pro neurčité výsledky akcí a pro neznámé akce

Online prohledávání

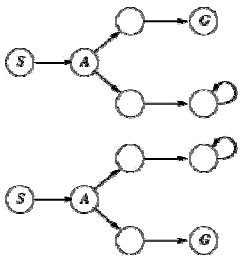
- Online prohledávání má smysl pro **agenty, kteří provádějí akce** (nemá smysl pro čistě „výpočtové" agenty).
- Agent má k dispozici následující **informace**
  - **Actions(s)** – seznam akcí aplikovatelných na stav S
  - **c(s,a,s`)** – cena provedení jednoho kroku (může být použita až když agent zná stav s`)
  - **Goal-Test(s)** – stav s je cílový
- Dále budeme předpokládat, že agent
  - umí **rozpoznat již navštívený stav**
  - (agent může budovat mapu světa)
  - používá **deterministické akce**
  - má k dispozici **přípustnou heuristiku** h(s)



Kvalita algoritmů

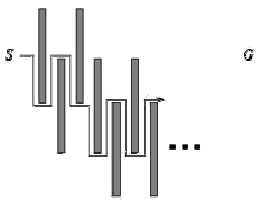
Kvalita online algoritmů se typicky měří **porovnáním s offline řešením**.

- **Kompetitivní poměr**
  = kvalita online řešení / kvalita nejlepšího řešení
  - Může být ∞, např. při vkročení do **slepé uličky** (pokud nemáme reverzní akce pro krok zpět).
  - **Tvrzení:** Žádný online algoritmus se umí vyhnout slepým uličkám ve všech problémech.
  - Důkaz (metodou protivníka)
    Agent, který navštívil S a A se musí v obou příkladech rozhodnout stejně a v jednom případě tedy skončí ve slepé uličce.



Budeme předpokládat **bezpečně prozkoumatelné světy** (z každého stavu vede cesta do cíle).

- Ani zde ale nelze garantovat kompetitivní poměr, pokud máme cesty neomezené ceny.
- Metodou protivníka můžeme stavět do cesty překážky, které cestu libovolně prodlouží.



Kvalita online algoritmů se proto často měří **vzhledem k velikosti celého stavového prostoru** (místo délky nejkratší cesty).

Online do hloubky

- Uvědomme si, že na rozdíl od offline algoritmů typu A* **nemohou on-line algoritmy přeskočit do zcela jiného stavu** aniž by absolvovaly příslušnou cestu.
- Je proto vhodné expandovat uzly v **lokálním** pořadí, jako to například dělá DFS.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          unexplored, a table that lists, for each visited state, the actions not yet tried
          unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state then unexplored[s'] ← ACTIONS(s')
  if s is not null then do
      result[a, s] ← s'
      add s to the front of unbacktracked[s']
  if unexplored[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
  else a ← POP(unexplored[s'])
  s ← s'
  return a
```

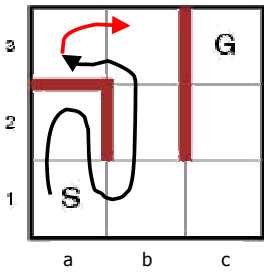
učíme se stavy po aplikaci akce

daným stavem můžeme i na jedné cestě projít několikrát (pokaždé odejdeme jinam), je proto nutné si pamatovat kam se vracet – jako **Ariadnina nit**

pro návrat potřebujeme reverzní akci k akci, která nás naposledy dovedla do stavu s`

Online do hloubky

příklad

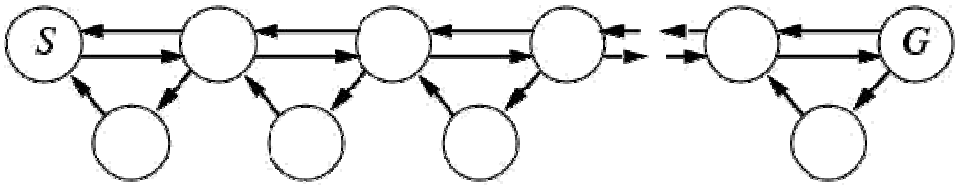


stav	unEX	unBT	rUP	rDN	rLF	rRG
(1,a)	{}	(2,a)	(2,a)	-	-	(1,b)
(2,a)	{}	(1,a)	-	(1,a)	-	-
(1,b)	LF,RG	(1,a)	(2,b)	-		
(2,b)	DW	(1,b)	(3,b)		-	-
(3,b)	DW	(3,a), (2,b)	-		(3,a)	-
(3,a)		(3,b)	-	-	-	(3,b)

- V nejhorším případě přejde po každém spoji dvakrát (tam a zpátky).
- To je optimální pro průzkum světa, ale cesta i do blízkého cíle může být dlouhá.
  - on-line verze iterování hloubky to řeší
- On-line prohledávání do hloubky lze uplatnit pouze tehdy, pokud jsou **akce oboustranné** (může se vrátit o krok zpět).

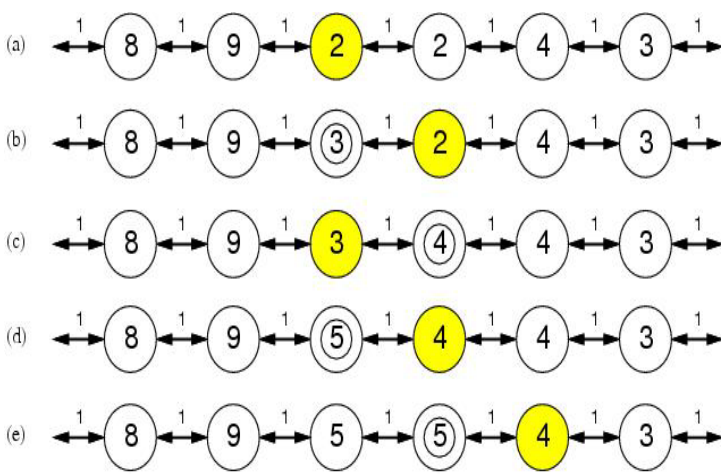
Online lokálně

- **Horolezecká metoda je on-line algoritmus.**
  - drží v paměti **jediný stav** (kde se nachází agent)
  - dělá pouze **lokální kroky** do „sousedních" stavů
  - v nejjednodušší podobě se ale **neumí dostat z lokálního optima**
    - Pozor! **Nelze vyřešit náhodným restartem!**
    - Můžeme ale použít **náhodnou procházku**.
      - Za předpokladu konečného stavového prostoru **nakonec najde řešení** nebo prozkoumá celý stavový prostor.
      - Obecně ale může **projít exponenciálně dlouhou cestu**.
      - V následujícím stavovém prostoru je pravděpodobnost kroku zpět dvakrát větší než pravděpodobnost kroku dopředu



Lokálně s učením

- Jiná (efektivnější) metoda cesty z lokálního optima je **využití paměti**.
  - **H(s)** – současný nejlepší odhad délky cesty ze stavu s do cíle (na začátku h(s))



Lokálně s učením

- Algoritmus LRTA* dělá lokální kroky a učí se, kam ho daná akce dovede (result) a jaká je vzdálenost do cíle (H).

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          H, a table of cost estimates indexed by state, initially empty
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  unless s is null
      result[a, s] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[b, s], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[b, s'], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

upřesníme odhad v předchozím stavu

vybereme další akci s nejlepší cenou do cíle (může vést i do předchozího stavu); dává přednost novým cestám

pokud jsme akci na daný stav ještě nepoužili, optimisticky předpokládáme, že nás dovede do cíle s nejmenší možnou cenou, tj. h(s)