# Informovaný Agent "Hledač"
# (4. přednáška)

# Opakování: strom akcí

**Idea**

Procházet všechny možné posloupnosti akcí a zjistit, která vede k cíli

Postupným procházením vytváříme strom

- uzly ve stromu jsou posloupnosti akcí
- každému uzlu odpovídá stav světa po provedení dané posloupnosti akcí, dvěma různým uzlům mohou odpovídat stejné stavy světa
- hledáme uzel, jehož stav je cílovým stavem

Postupně budujeme strom akcí, dokud nenarazíme na cílový stav.

- v každém kroku dle nějaké strategie vybereme uzel ve stromu akcí
- v tomto uzlu strom rozšíříme
  - za každou akci, kterou lze v daném uzlu (resp. stavu) provést, přidáme do stromu akcí nový uzel

# Opakování: Stromové prohledávání dle strategie

```
def treeSearch( problem, strategy ):
  # Inicializace prohledavani
  fringe = node(None, problem.initialState())
  while nonempty( fringe ):
    # Zvol kandidata k expanzi dle strategie
    leafNode = strategy.choose(fringe, problem)
    state = leafNode.state()
    path = leafNode.path()
    # Mas-li reseni, vrat ho
    if problem.is_goal( state ):
      return leafNode
    # Expanduj kandidata: pridej jeho sousedy
    for a in state.possibleActions():
      child = node(path.append(a),state.act(a))
      fringe.append(child)
  # Dojdou-li kandidati, priznej porazku
  return failure
```

# Opakování: uninformed search

- úplnost: nalezne cílový stav, pokud existuje
- optimalita: nejkratší posloupnost vedoucí k cílovému stavu
- časová složitost: kolik uzlů je třeba vygenerovat
- paměťová náročnost: kolik uzlů je třeba držet v paměti: potenciální kandidáti a údaje potřebné k rekonstrukci optimální cesty po nalezení cíle

b — faktor větvení (branching factor)

d — hloubka optimálního řešení

m — hloubka prohledávaného stromu

| | | | složitost | |
|---|---|---|---|---|
| | úplnost | optimalita | časová | paměťová |
| do šířky (BFS) | ANO | ANO | $O(b^d)$ | $O(b^d)$ |
| do hloubky (DFS) | NE | NE | $O(m)$ | $O(d)$ |
| omezená hloubka (DLS) | ANO | ANO | $O(b^d)$ | $O(d)$ |
| iterování hloubky (IDS) | ANO | ANO | $O(b^d)$ | $O(d)$ |

# Grafové prohledávání dle strategie

```
def treeSearch( problem, strategy ):
  fringe = node(None, problem.initialState())
  while nonempty ( fringe ):
    # Zvol kandidata k expanzi dle strategie
    leafNode = strategy.choose(fringe, problem)
    state = leafNode.state()
    path = leafNode.path()
    if problem.is_goal( state ):
      return leafNode
    # GraphSearch - uloz si navstivene stavy
    visited.append( state )
    for a in state.possibleActions():
      resultState = state.act(a)
      if resultState in visited:
        continue
      child = node(path.append(a),resultState)
      fringe.append(child)
  # Dojdou-li kandidati, priznej porazku
  return failure
```

# Strategie — evaluační funkce

```python
def strategy( fringe, problem ):
    candidate = fringe[0]
    fitness = evalfunc(candidate)
    for i in range(len(fringe)):
      leaf = fringe[i]
      if evalfunc(i,leaf) <  fitness:
          candidate = leaf
          fitness = evalfunc(leaf)
    fringe.remove(candidate)
    return candidate
```

Vhodnou volbou evaluační funkce `evalfunc` dostáváme jednotlivé algoritmy:

- BFS — `evalfunc(i,leaf) = depth(leaf)`
- DFS — `evalfunc(i,leaf) = len(fringe)-i`

# Problém — expanze neperspektivních uzlů

Algoritmus zbytečně expanduje neperspektivní uzly.

- Hloubka uzlu (resp. jeho pořadí) nijak nesouvisí s "perspektivností".
- V konkrétních případech lze často perspektivnost uzlu odhadnout:
  - routing (hledání cesty na mapě) — perspektivní jsou ty uzly, které jsou blíž cíli
  - loydova osmička — perspektivní jsou ty uzly, kde je osmička blíže cílovému uspořádání

Na základě znalosti jednotlivých problémů lze nalézt vhodnout heuristickou funkci $h(node)$ a položit např.

```
evalfunc(i,leaf) = h(leaf)
```

# Hladové algoritmy (greedy best-first search)

Heuristickou funkci $h(node)$ lze interpretovat jako "cenu" cesty z daného uzlu do cíle. Hladový algoritmus volí uzel s nejmenší cenou:

```
evalfunc(i,leaf) = h(leaf)
```

- úplnost: NE (treesearch), ANO (graph search pro konečné grafy)
- časová složitost: obecně $O(b^m)$, dobrá heuristika může dramaticky zlepšit
- paměťová náročnost: obecně $O(b^m)$, dobrá heuristika může dramaticky zlepšit

Momentálně nejlevnější uzel nemusí být nejvhodnější.
(Nejsme tak bohatí, abychom si kupovali levné věci)

# $A^*$ search — minimalizace celkových nákladů

$A^*$ algoritmus bere v potaz nejen odhadovanou "cenu" cesty k cíli $h(node)$, ale i cenu cesty z počátku $g(node)$.

$$\text{evalfunc(i,leaf)} = \text{g(leaf)} + \text{h(leaf)}$$

- Algoritmus tedy neprodlužuje cesty, které už jsou dlouhé.
- Evaluační funkce udává odhadovanou cenu nejlevnější cesty přes daný uzel.
- Pro vhodnou volbu $h(node)$ (přípustná, konzistentní) je algoritmus optimální!

# Přípustné a monotónní heuristiky

Heuristika je **přípustná** (admisibile), pokud je vždy nižší, než minimální celková cena cesty z daného uzlu do cíle. Heuristika je **monotónní** (příp. konzistentní), pokud splňuje variantu tzv. trojúhelníkové nerovnosti:

$$h(node) \leq cost(node, action, successor) + h(successor),$$

kde $cost(node, action, successor)$ je reálná cena cesty z uzlu $node$ do následnického uzlu $successor$ pomocí akce $action$.

**Věta**: Monotónní heuristika je přípustná.

# Optimalita $A^*$

**Věta**: $A^*$ (tree search) je optimální pro přípustné heuristiky.

**Věta**: $A^*$ (graph search) je optimální pro monotónní heuristiky.

Idea:

- evaluační funkce je neklesající podél libovolné cesty
- kdykoliv je zvolen uzel k expanzi, nejkratší cesta do daného uzlu je již známa

# Vlastnosti $A^*$

**Absolutní chyba** heuristiky ($\Delta$) je rozdíl mezi odhadovanou cenou optimálního řešení $h(root)$ a reálnou cenou optimálního řešení $h^*(root)$. **Relativní chyba** ($\varepsilon$) je $\Delta/h^*(root)$.

- úplnost: ANO (pro konečné grafy)
- časová složitost: obecně $O(b^d)$, resp. $O(b^{\varepsilon d})$
- paměťová náročnost: obecně $O(b^d)$, resp. $O(b^{\varepsilon d})$

Největším problémem je stále paměť.

# $IDA^*, RBFS, SMA$

- iterative deepening $A*$ ($IDA^*$) — podobné jako IDS, limitem není hloubka ale hodnota evaluační funkce
- rekurzivní best-first search ($RBFS$) — prohledává nejslibnější větev; pamatuje si nejslibnější již prohledanou alternativu; pokud cena aktuální větve přesáhne cenu alternativy, aktuální větev zahodí a pokračuje v alternativě; má lineární paměťovou složitost, ale často znovu generuje větve, které dřív zahodil
- (simplified) memory-bounded $A*$ ($MA*$ a $SMA*$) — lépe využívá dostupnou paměť, uzly zahazuje až ve chvíli, kdy dojde paměť

# Heuristiky — kvalita, přesnost

Kvalitu heuristiky $h$ lze měřit pomocí tzv. **efektivního faktoru větvení**. Efektivní faktor větvení $b(h)$ je nejmenší $b$ takové, že algoritmus $A^*$ s danou heuristikou expanduje tolik uzlů, kolik má plný $b$-ární strom výšky $d$, kde $d$ je hloubka optimálního řešení .

- $h(b)$ se bude lišit problém od problému, pro dostatečně těžkou třídu problémů je relativně konstantní
- v důsledku předchozího bodu lze meřit experimentálně
- je-li $h_1 \geq h_2$ (t.j. pro každý uzel $n$ platí $h_1(n) \geq h_2(n)$), pak $b(h_1) \leq b(h_2)$

Máme-li dvě (přípustné, resp. konzistentní) heuristiky $h_1, h_2$, lze získat novou (přípustnou, resp. konzistentní) heuristiku pomocí

$$h(node) = \max\{h_1(node), h_2(node)\}$$

# Generování Heuristik — relaxace

- řešení problému musí splňovat nějaké podmínky
- uvolněním podmínek získáme typicky jednodušší problém
- řešení jednoduššího problému může být triviální
- takto lze generovat heuristiku — cena řešení "uvolněného" problému

**Příklad.** Loydova osmička: kostku lze přemístit z políčka $A$ na políčko $B$ pokud

- políčka spolu sousedí a
- políčko $B$ je prázdné

Relaxací těchto podmínek získáme tři různé problémy (každý jedoduše řešitelný): Kostku lze přemístit z políčka $A$ na políčko $B$ pokud

- políčka spolu sousedí (manhattan metric) nebo
- políčko $B$ je prázdné nebo
- kdykoliv (počet špatně umístěných políček)

# Automatické Generování Heuristik pomocí relaxace

- popsaný postup lze zformalizovat a automatizovat
- program ABSOLVER (první rozumná heuristika pro Rubikovu kostku, nejlepší heuristika pro Loydovu osmičku)

# Generování Heuristik — databáze vzorů

- nalezneme cenu optimálního řešení pro všechny malé “podproblémy”
- cena daného problému je maximum z cen těch podproblémů, které jsou v něm obsaženy
- obecně cenu nelze sčítat v pokud volíme podproblémy opatrně, může se to podařit

# Generování Heuristik — machine learning

- zvolíme charakteristiky problému $x_1(p), \ldots, x_n(p)$ (angl. features)
- vyřešíme mnoho náhodně generovaných problémů $p_1, \ldots, p_N, \; N \approx 10000$ získáme tím přesnou cenu $c(p_1), \ldots, c(p_N)$
- snažíme se najít nejlepší lineární (případně polynomiální) funkci "interpolující" získaná data t.j.

hledáme $c_1, k_1, \ldots, c_n, k_n$ tak,abychom minimalizovali

$$\sum_{i=0}^{N} c(p_i) - \left( \sum_{j=0}^{n} c_j x_j(p)^{k_j} \right).$$

Získáme tak heuristiku:

$$h(p) = \sum_{j=0}^{n} c_j x_j(p)^{k_j}.$$