Umělá inteligence I

4

Roman Barták, KTIML

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

Na úvod

- **Neinformované (slepé)** prohledávání umí najít (optimální) řešení problému, ale ve většině případů je hodně neefektivní.
- Naopak **informované** prohledávání, které používá problémově závislou informaci, může řešení nalézt mnohem rychleji.
  - **Jak využívat heuristiky v prohledávání?**
    - BFS, A*, IDA*, RBFS, SMA*
  - **Jak konstruovat heuristiky?**
    - relaxace, databáze vzorů

Informace v prohledávání

- Připomeňme, že hledáme (nejkratší) cestu z počátku do nějakého cílového stavu ve stavovém prostoru.
- Jaká informace může pomoci prohledávacímu algoritmu?
  - Například délka cesty do nějakého cílového stavu.
  - Tento údaj ale zpravidla nebývá známý (pokud ano, potom nemusíme nic prohledávat), proto se místo něj používá **evaluační funkce f(n)** ohodnocující uzel n na základě délky cesty do cíle.
  - **prohledávání dle ceny (best-first search)**
    - po expanzi pokračujeme s uzlem s nejmenší hodnotou f(n).
  - Existují různé prohledávací algoritmy lišící se funkcí f(n), její složkou ale skoro vždy bývá **heuristická funkce h(n)** odhadující délku nejkratší (nejlevnější) cesty do cíle.
    - Heuristická funkce je nejběžnější formou dodatečné informace pro prohledávání.
    - Nadále budeme uvažovat, že **h(n) = 0 ⇔ n je cílový stav**.

Hledání s cenou

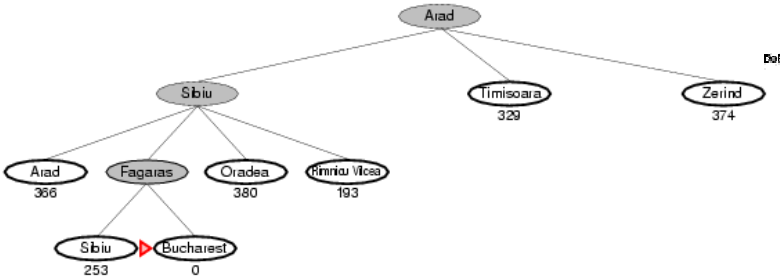
- Zkusme při prohledávání preferovat uzel, který je nejblíže cíli, tj. f(n) = h(n).
  - **hladový algoritmus prohledávání s cenou**

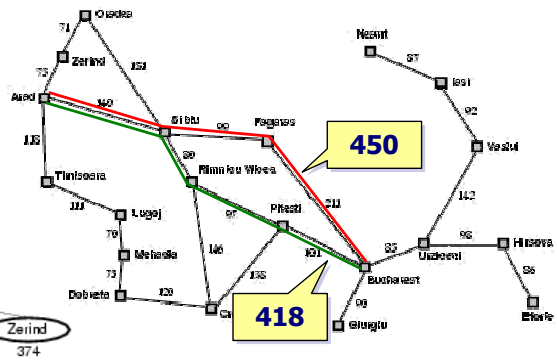
Příklad (cesta Arad → Bukurešť):

- Máme k dispozici tabulku přímých vzdáleností do Bukurešti.
- Pozn.: tato informace není součástí původního problému!

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244

Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



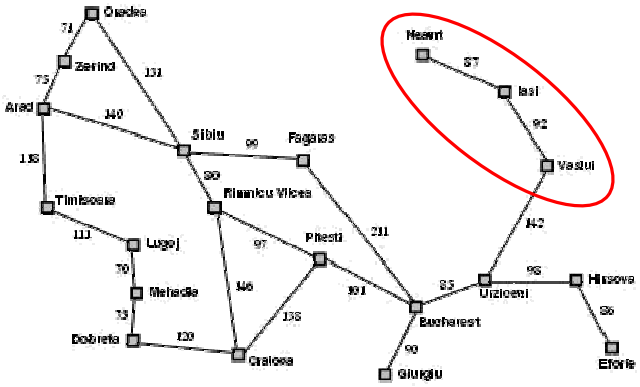


Nejkratší cesta?

Hledání s cenou
analýza

- Již víme, že hladový algoritmus s cenou **nemusí najít optimální řešení**.
- **Garantuje alespoň nalezení řešení?**
  - Pokud vždy bude pro expanzi brát uzel s nejmenší cenou, tak **řešení najít nemusí**.
    - **Příklad: cesta Iasi → Fagaras**
      - půjde do Neamt, potom opět Iasi, Neamt, …
      - je potřeba detekovat opakované stavy!

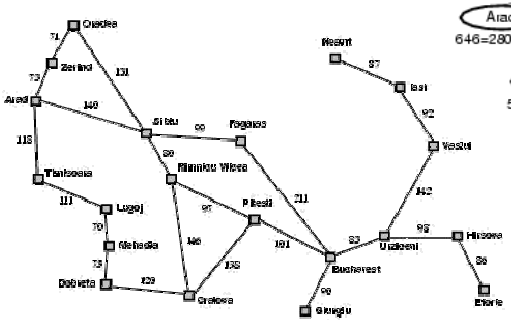
- **Čas $O(b^m)$,** kde m je maximální hloubka
- **Paměť $O(b^m)$**
- Dobrá heuristická funkce může složitost výrazně redukovat.

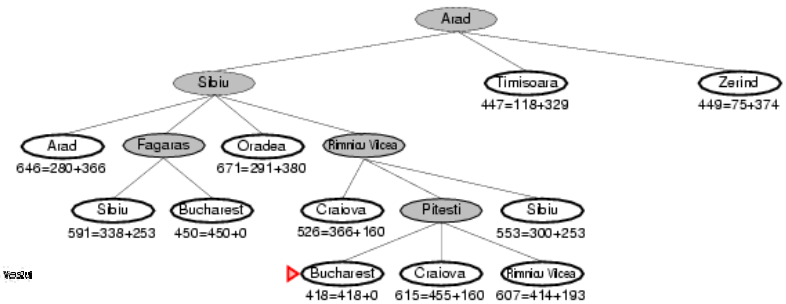


Algoritmus A*

- Zkusme nyní vzít **f(n) = g(n) + h(n)**
  - připomeňme, že g(n) je cena cesty z kořene do n
  - asi nejznámější algoritmus prohledávání s cenou
  - f(n) reprezentuje cenu cesty přes n
  - algoritmus tedy neprodlužuje cesty, které už jsou dlouhé

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



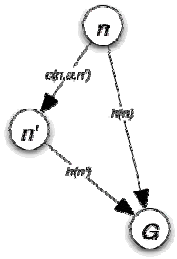


Vlastnosti heuristik

Jak je to s úplností a optimalitou A*?

Nejprve pár definic:

- **přípustná heuristika h(n)**
  - **h(n) ≤ „nejmenší cena cesty z n do cílového uzlu"**
  - optimistický pohled (uvažuje lepší cenu, než jaká skutečně je)
  - funkce f(n) u A* tedy také zdola odhaduje cenu řešící cesty přes n
- **monotónní (konzistentní) heuristika h(n)**
  - nechť n` je následník n přes akci a, c(n,a,n`) je cena přechodu
  - **h(n) ≤ c(n,a,n`) + h(n`)**
  - jedná se o formu trojúhelníkové nerovnosti



Monotónní heuristika je přípustná.

nechť n_1, n_2,..., n_k je optimální cesta z n_1 do n_k, potom
$h(n_i) - h(n_{i+1}) \leq c(n_i,a_i,n_{i+1})$, dle monotonie
$h(n_1) \leq \Sigma_{i=1,..,n-1}\ c(n_i,a_i,n_{i+1})$, po „sečtení"

Pro monotónní heuristiku jsou hodnoty f(n) podél libovolné cesty neklesající.

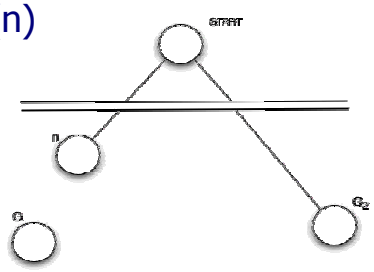
nechť n` je následník n, tj. g(n`) = g(n) + c(n,a,n`), potom
f(n`) = g(n`) + h(n`) = g(n) + c(n,a,n`) + h(n`) ≥ g(n) + h(n) = f(n)

Algoritmus A*

optimalita

- **Je-li h(n) přípustná heuristika, potom je algoritmus A* v rámci TREE-SEARCH optimální.**
  - jinými slovy, první expandovaný cílový uzel je optimální
  - nechť G_2 je sub-optimální cílový uzel z okraje a C* je optimální cena
    - $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$, protože $h(G_2) = 0$
  - nechť n je uzel z okraje, který je na optimální cestě
    - $f(n) = g(n) + h(n) \leq C^*$, dle přípustnosti h(n)
  - dohromady
    - $f(n) \leq C^* < f(G_2)$,

    tj. algoritmus musí vždy expandovat n dříve než G_2 a tím se dostane do optimálního cílového uzlu.



Algoritmus A*

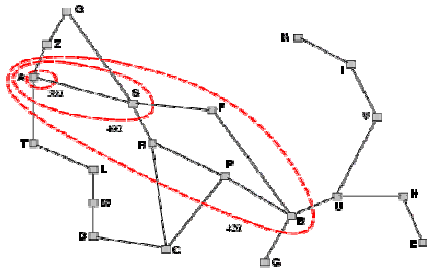
optimalita

- **Je-li h(n) monotónní heuristika, potom je algoritmus A* v rámci GRAPH-SEARCH optimální.**
  - Problém nastane, pokud se do stejného stavu podruhé dostaneme lepší cestou – standardní GRAPH-SEARCH tuto druhou cestu zahodí!
  - Řešením může být vybrání lepší z cest u již zavřeného uzlu (extra bookkeeping) nebo použití monotónní heuristiky.
    - pro monotónní heuristiku jsou hodnoty f(n) podél libovolné cesty neklesající
    - pro expanzi (zavření) vybírá A* vždy uzel s nejmenším f(n), tj. ostatní otevřené uzly m nemají menší f(m), tj. mezi „otevřenými" cestami nemůže být žádná cesta vedoucí do n lepší než ta právě nalezená (cesta se nezkrátí)
    - proto první zavřený cílový uzel musí být optimální

Algoritmus A*

vlastnosti

- Pro neklesající f(n) je možné ve stavovém grafu nakreslit **vrstevnice** tak, že v dané vrstvě jsou vždy všechny vrcholy mající f(n) do dané meze.
  - pro h(n) = 0 dostaneme soustředné okruhy
  - pro přesnější h(n) se budou okruhy „protahovat" podél optimální cesty k cíli.



  - A* expanduje všechny uzly, kde f(n) < C*, a to po vrstevnicích
  - A* může expandovat některé uzly, kde f(n) = C*
  - uzly, kde f(n) > C*, nejsou expandovány
  - algoritmus tedy **vždy najde optimální řešení**

Časová složitost:

A* může expandovat exponenciálně mnoho uzlů

- lze tomu zabránit, pokud $|h(n)-h^*(n)| \leq O(\log h^*(n))$, kde $h^*(n)$ je cena optimální cesty z n do cíle

Prostorová složitost:

A* drží v paměti všechny vygenerované uzly

A* zpravidla dříve vyčerpá paměť než nastanou problémy s časem

Omezení paměti

- Jednoduchý způsob zmenšení paměťových nároků je použití iterovaného prohlubování.
- **Algoritmus IDA***

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f-COST limit
          root, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f-COST(root)
  loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end

function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
  inputs: node, a node
          f-limit, the current f-COST limit
  static: next-f, the f-COST limit for the next contour, initially ∞

  if f-COST[node] > f-limit then return null, f-COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
  return null, next-f
```

- pro limit prohledávání se místo hloubky použije cena f(n)
- pro následující iteraci se použije limit odpovídající nejmenšímu f(n) uzlu n, který v předchozí iteraci překročil limit
- často používaný algoritmus

Rekurzivní BFS

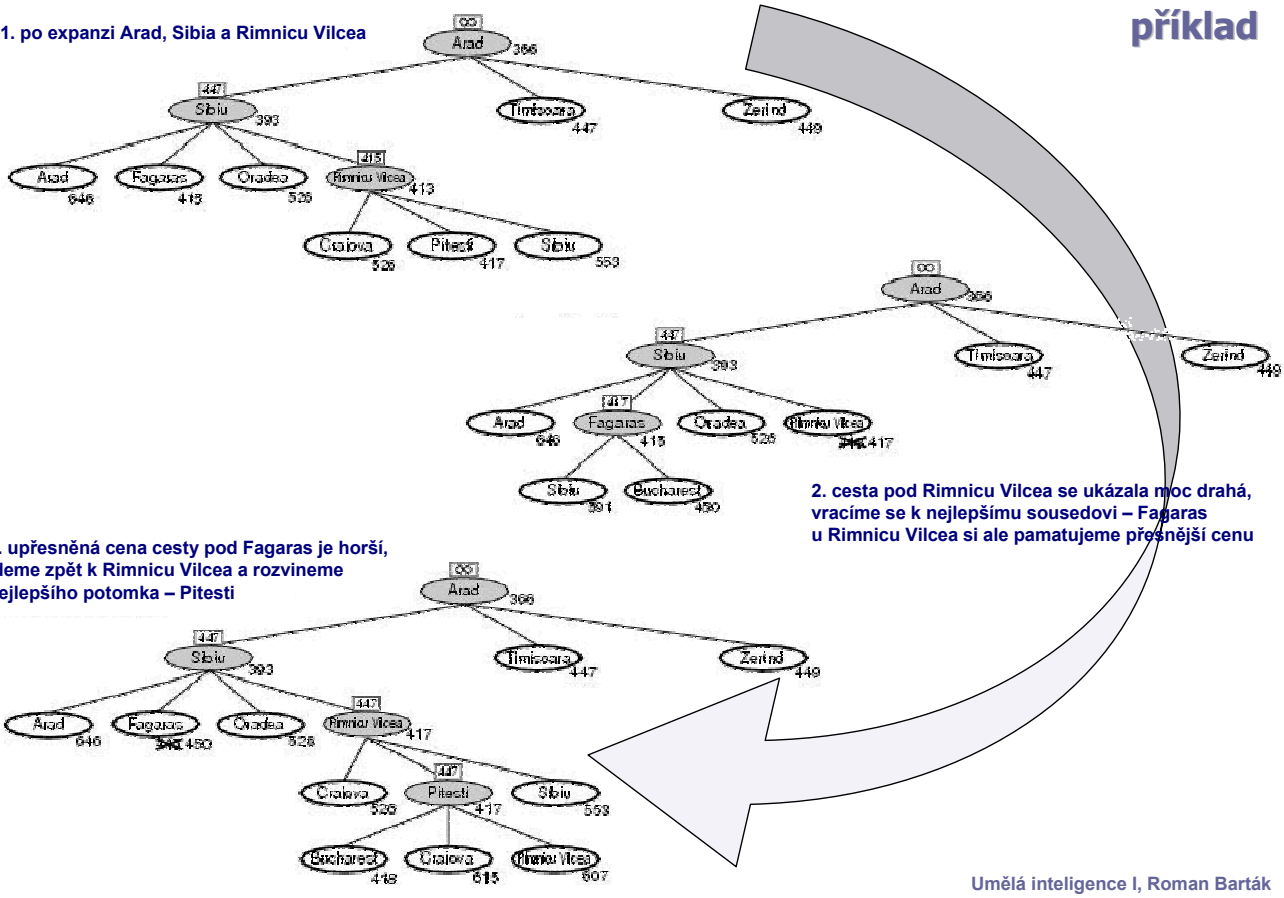
- Zkusme **rekurzivní verzi prohledávání s cenou** za použití lineárního prostoru
  - algoritmus přeruší prohledávání, pokud je v paralelní větvi lepší cena f(n)
  - pokud se algoritmus vrací do uzlu n, upřesní jeho cenu f(n) dle prozkoumaných potomků (paměť již prozkoumané části)
- **Je-li h(n) přípustná, je algoritmus optimální.**
- **Paměťová složitost O(bd)**
- **Časová složitost stále exponenciální** (řada uzlů se může procházet opakovaně)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        if f[best] > f_limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

Rekurzivní BFS

příklad

1. po expanzi Arad, Sibia a Rimnicu Vilcea



2. cesta pod Rimnicu Vilcea se ukázala moc drahá, vracíme se k nejlepšímu sousedovi – Fagaras u Rimnicu Vilcea si ale pamatujeme přesnější cenu

3. upřesněná cena cesty pod Fagaras je horší, jdeme zpět k Rimnicu Vilcea a rozvineme nejlepšího potomka – Pitesti

Zjednodušené MA*

- IDA* i RBFS nevyužívají dostupnou paměť!
- To je škoda, protože se dříve expandované uzly znova procházejí (ztráta času)
- Zkusme vzít standardní algoritmus A*

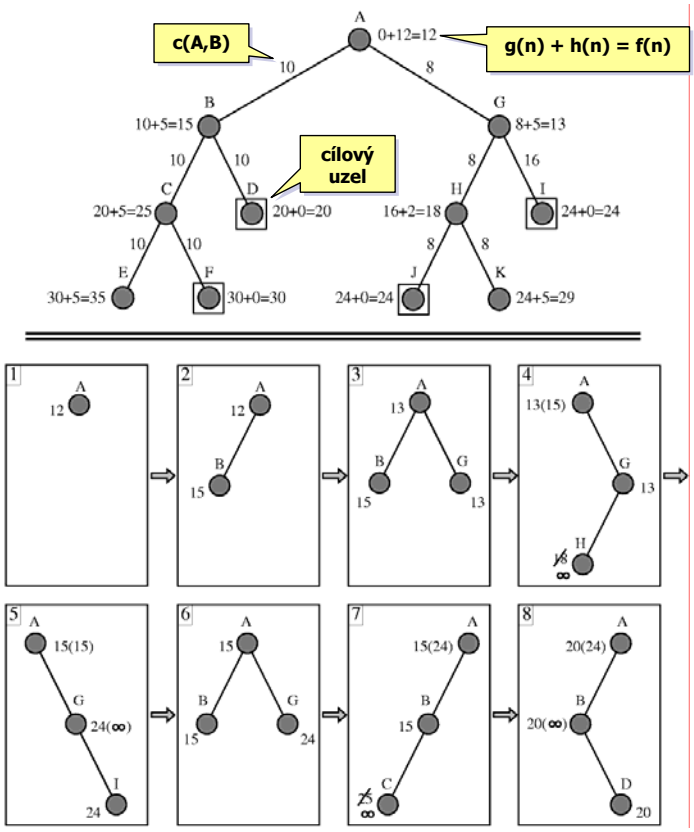
```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
    if Queue is empty then return failure
    n ← deepest least-f-cost node in Queue
    if GOAL-TEST(n) then return success
    s ← NEXT-SUCCESSOR(n)
    if s is not a goal and is at maximum depth then
      f(s) ← ∞
    else
      f(s) ← MAX(f(n), g(s)+h(s))
    if all of n's successors have been generated then
      update n's f-cost and those of its ancestors if necessary
    if SUCCESSORS(n) all in memory then remove n from Queue
    if memory is full then
      delete shallowest, highest-f-cost node in Queue
      remove it from its parent's successor list
      insert its parent on Queue if necessary
    insert s on Queue
  end
```

Cesta z kořene k tomuto necílovému listu se nevejde do paměti, a proto přes tento uzel nemůžeme najít optimální řešení!

- když vyčerpá paměť, vyhodíme nejméně slibný list, tj. ten s největší cenou f(n)
- podobně jako u RBFS si ale tuto cenu zapamatujeme u rodiče

Zjednodušené MA*

příklad

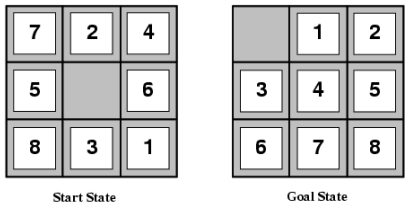


- Uvažujme paměť pouze pro **tři uzly**.
- Pokud je paměť dostatečná pro (optimální) cestu, SMA* najde (optimální) řešení.
- Jinak najde nejlepší možné řešení s dostupnou pamětí.
  - Pokud by cena J byla 19, tak to je optimální řešení, ale cesta k němu se nevejde do paměti!

Hledáme heuristiky

Jak hledat přípustné heuristiky?

Příklad: Loydova osmička



- průměrně 22 kroků k nalezení řešení
- větvící faktor je kolem 3
- (plný) prohledávací strom: $3^{22} \approx 3{,}1 \times 10^{10}$ uzlů
- počet stavů ale jen 9!/2 = 181 440
- pro Loydovu patnáctku už 10^{13} stavů
- potřebujeme heuristiku, pokud možno přípustnou
  - h_1 = „počet špatně umístěných kostiček"
    = 8
  - h_2 = „součet vzdáleností kostiček od cílových pozic"
    = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18
    tzv. Manhattanská heuristika
  - skutečné řešení potřebuje 26 kroků

Výkon heuristik

Jak měřit kvalitu heuristik?

Efektivní větvící faktor b*

- nechť algoritmus potřebuje N uzlů na nalezení řešení v hloubce d.
- b* je větvící faktor rovnoměrného stromu hloubky d, který obsahuje N+1 uzlů, tj.

$$N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

Příklad:

- Loydova patnáctka
- průměr ze 100 náhodných problémů pro každou délku řešení

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Dominance

- **Je h_2** (z Loydovy osmičky) **vždy lepší než h_1 a jak to snadno poznat?**
  - všimněme si, že $\forall n\ h_2(n) \geq h_1(n)$
  - říkáme, že **h_2 dominuje h_1**
  - A* s h_2 nikdy neexpanduje více uzlů než A* s h_1
    - A* expanduje všechny uzly, kde $f(n) < C^*$, tj. $h(n) < C^* - g(n)$
    - tedy, pokud expanduje uzel dle h_2, potom expanduje stejný uzel i dle h_1
- **Vždy je lepší použít heuristickou funkci s většími hodnotami.**
  - **pokud nepřekročí mez C* - g(n) (to by nebyla přípustná)**
  - **pokud se nepočítá příliš dlouho**

Relaxace

Může agent sám najít přípustné heuristiky pro obecný problém?

Ano, pomocí **relaxace problému!**

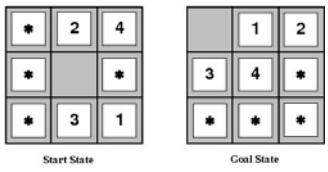
- relaxace je zjednodušení problému takové, aby řešení původního problému bylo řešením i relaxovaného problému (i když třeba neoptimální)
- relaxujeme do takové míry, že relaxovaný problém umíme rychle vyřešit
- cena optimálního řešení relaxovaného problému je potom dolním odhadem ceny originálního problému, tj. přípustnou (a také monotónní) heuristikou

- **Příklad (Loyd)**
  - kostičku lze posunou z místa A na místo B pokud:
    - A je horizontálně i vertikálně sousedem B
    - B je prázdné políčko
  - možné relaxace (vyřazení podmínek z pravidla pro posun):
    - posun A na B je možný, pokud A je sousedem B (Manhattanská heuristika)
    - posun A na B je možný, pokud B je prázdné
    - posun A na B je možný vždy (heuristika h_1)

Databáze vzorů

Jiný přístup k hledání přístupných heuristik je přes **databázi vzorů** (pattern database)

- založeno na **řešení podproblémů** (typových příkladů)
- prohledáním od cíle najdeme různá optimální řešení, ze kterých uděláme **vzory**



- heuristiku sestrojíme tak, že vezmeme nejhorší optimální řešení pro vzory, které najdeme v aktuálním stavu
- Pozor! Součet řešení vzorů nemusí být přípustný (v postupu pro řešení jednoho vzoru můžeme použít kroky z řešení jiných vzorů)

Máme-li **více heuristik**, můžeme z nich vzít **maximum** (dominuje každé z nich).