Prohledávání stavového prostoru

Aleš Horák

E-mail: hales@fi.muni.cz
http://nlp.fi.muni.cz/uui/

Obsah:

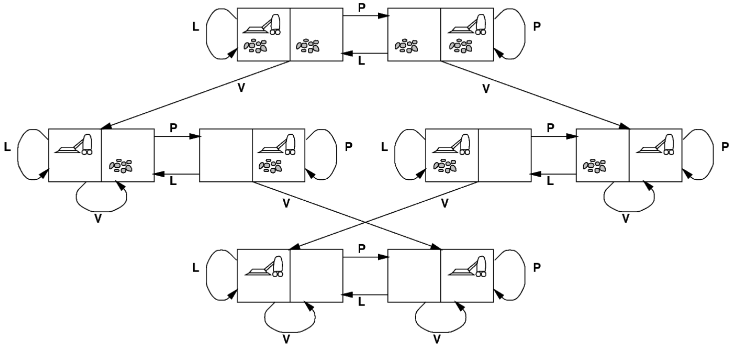
- Prohledávání stavového prostoru
- Neinformované prohledávání
- Informované prohledávání stavového prostoru
- Jak najít dobrou heuristiku?

Prohledávání stavového prostoru

Řešení problému prohledáváním stavového prostoru:

- stavový prostor, předpoklady – statické a deterministické prostředí, diskrétní stavy
- *počáteční stav* **problem.init_state**
- *cílová podmínka* **problem.is_goal(State)**
- *přechodové akce* **problem.moves(State)** $\rightarrow$ **NewStates**

Problém agenta Vysavače

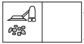
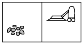


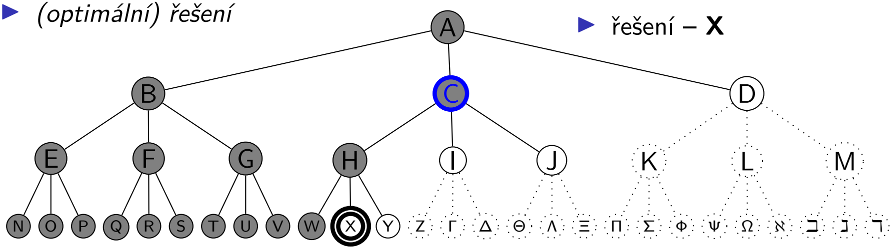
- máme dvě místnosti (L, P)
- jeden vysavač (v L nebo P)
- v každé místnosti je/není špína
- počet stavů je $2 \times 2^2 = 8$
- akce $=\{doLeva, doPrava, Vysávej\}$

Problém agenta Vysavače

Prohledávací strategie – prohledávací strom:

- *kořenový uzel*
- *uzel* prohledávacího stromu:
  - (odkaz na) *stav*
  - *rodičovský uzel*
  - *přechodová akce*
  - *hloubka uzlu*
  - *cena* – $g(n)$ cesty, $c(x, a, y)$ přechodu
- *(optimální) řešení*

- **A** (stav )
- např. uzel **C**:
  - stav – 
  - rodič – **A**
  - akce – **doPrava**
  - hloubka – **1**
  - cena – **1**
- řešení – **X**



Řešení problému prohledáváním

Kostra algoritmu:

```
function SEARCH(problem)
    process ← collection(problem. init_state )  # stavy ke zpracování
    while length(process) > 0  do
        current_node ← remove_current_node(process)
        if  problem. is_goal ( current_node)  then print  current_node  # řešení
        foreach  child  in  problem.moves(current_node) do
            process . add_node(child)
```

rekurzivně včetně cesty k řešení:

```
function RECURSIVESEARCH(problem,  path = collection())
    if  length(path)  = 0 then
        return RecursiveSearch(problem,  collection (problem. init_state ))
    current_node  = get_current_node(path)
    if  problem. is_goal ( current_node)  then print  path  # cesta k řešení
    foreach  child  in  problem.moves(current_node) do
        RecursiveSearch(problem,  path.with_node(child))
```

Prohledávací strategie

problem.moves(State) $\rightarrow$ **NewStates** – definuje prohledávací strategii

Porovnání strategií:

- úplnost
- optimálnost
- časová složitost
- prostorová složitost

složitost závisí na:

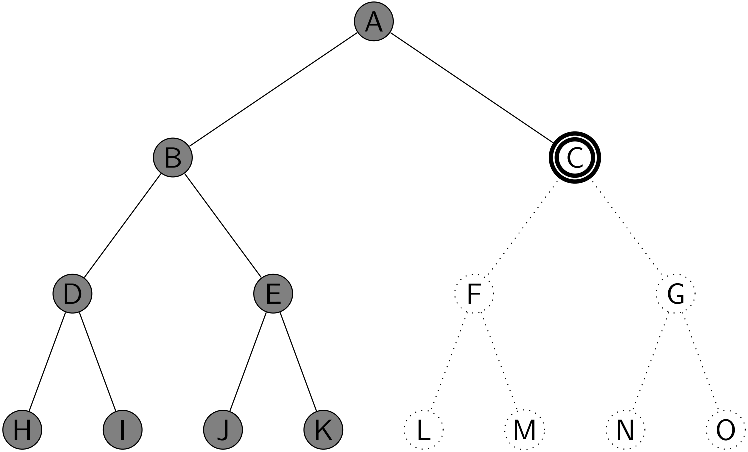
- b – faktor větvení (branching factor)
- d – hloubka cíle (goal depth)
- m – maximální hloubka větve/délka cesty (maximum depth/path, může být ∞?)

Neinformované prohledávání

- prohledávání do hloubky
- prohledávání do hloubky s limitem
- prohledávání do šířky
- prohledávání podle ceny
- prohledávání s postupným prohlubováním

Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



Prohledávání do hloubky

```
function DepthFirstSearch(problem, path ← [])
    if length(path) = 0 then
        return DepthFirstSearch(problem, [problem.init_state])
    current_node ← path.last()   # poslední prvek cesty
    if problem.is_goal(current_node) then
        print path
    foreach child in problem.moves(current_node) do
        if child ∉ path then
            DepthFirstSearch(problem, path + [child])
```

úplnost	není úplný (nekonečná větev, cykly)
optimálnost	není optimální
časová složitost	$O(b^m)$
prostorová složitost	$O(bm)$, lineární

Největší problém – nekonečná větev = nenajde se cíl, program neskončí!

Prohledávání do hloubky s limitem

Řešení nekonečné větve – použití “zarážky” = limit hloubky ℓ

```
function DepthFirstSearchLimited(problem, limit , path ← [])
    if length(path) = 0 then
        return DepthFirstSearchLimited(problem, limit , [problem.init_state ])
    current_node ← path.last()  # poslední prvek cesty
    if problem.is_goal (current_node) then print path  # cesta k řešení
    if limit = 0 then return "cutoff"
    cutoff_occured ← False
    foreach child in problem.moves(current_node) do
        if child ∉ path then
            result ← DepthFirstSearchLimited(problem, limit-1, path + [child])
            if result = "cutoff" then cutoff_occurred ← True
    if cutoff_occurred then return "cutoff" else return "exhausted"
```

Prohledávání do hloubky s limitem

konec má dvě možné interpretace – vyčerpání limitu nebo neexistenci (dalšího) řešení

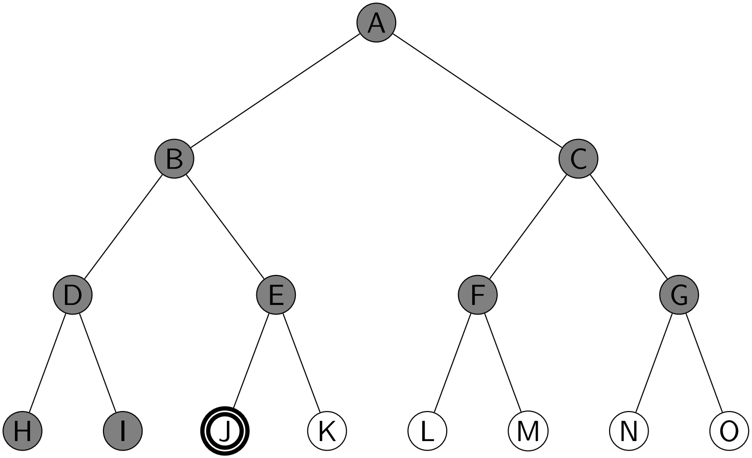
Vlastnosti:

úplnost	není úplný (pro $\ell < d$)
optimálnost	není optimální (pro $\ell > d$)
časová složitost	$O(b^\ell)$
prostorová složitost	$O(b\ell)$

dobrá volba limitu ℓ – podle znalosti problému

Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou.
(*Breadth-first Search, BFS*)



Prohledávání do šířky

ve frontě (FIFO) udržuje seznam cest

```
function BreadthFirstSearch(problem)
    process ← [[problem.init_state]]    # seznam cest k aktuálnímu uzlu
    while length(process) > 0 do
        current_path ← process.first()
        current_node ← current_path.last()
        if problem.is_goal(current_node) then
            print current_path    # vypíše cestu k řešení
        foreach child in problem.moves(current_node) do
            if child ∉ current_path then
                process ← process + [current_path + [child]]
```

Prohledávání do šířky – vlastnosti

úplnost	je úplný (pro konečné b)
optimálnost	je optimální podle délky cesty/není optimální podle obecné ceny
časová složitost	$1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, exponenciální v d
prostorová složitost	$O(b^{d+1})$ (každý uzel v paměti)

Největší problém – paměť:

Hloubka	Uzlů	Čas	Paměť
2	1100	0.11 sek	1 MB
4	111 100	11 sek	106 MB
6	10^7	19 min	10 GB
8	10^9	31 hod	1 TB
10	10^{11}	129 dnů	101 TB
12	10^{13}	35 let	10 PB
14	10^{15}	3 523 let	1 EB

Ani čas není dobrý → potřebujeme informované strategie prohledávání.

Prohledávání podle ceny

- BFS je optimální pro rovnoměrně ohodnocené stromy $\times$ prohledávání podle ceny (Uniform-cost Search) je optimální pro obecné ohodnocení
- fronta uzlů se udržuje uspořádaná podle ceny cesty

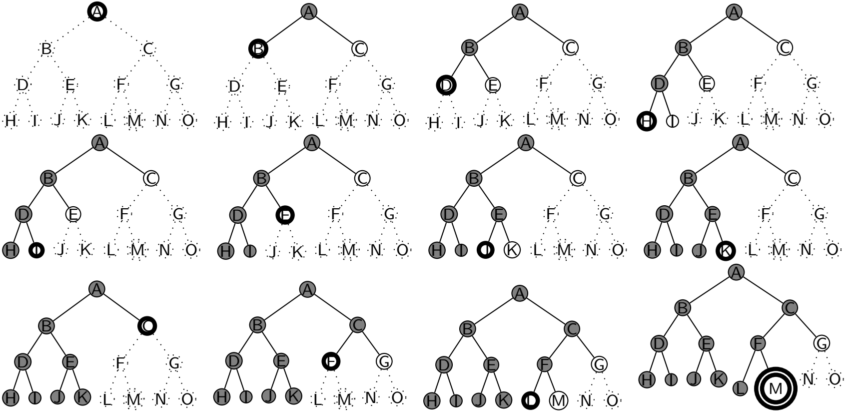
Vlastnosti:

úplnost	je úplný (pro cena $\geq \epsilon$ a b konečné)
optimálnost	je optimální (pro cena $\geq \epsilon$, $g(n)$ roste)
časová složitost	počet uzlů s $g \leq C^*$, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, kde $C^*\ldots$ cena optimálního řešení
prostorová složitost	počet uzlů s $g \leq C^*$, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Prohledávání s postupným prohlubováním

prohledávání do hloubky s postupně se zvyšujícím limitem (Iterative deepening DFS, IDS)

limit=3



Prohledávání s postupným prohlubováním – vlastnosti

úplnost	je úplný (pro konečné b)
optimálnost	je optimální (pro $g(n)$ rovnoměrně neklesající funkce hloubky)
časová složitost	$d(b) + (d-1)b^2 + \ldots + 1(b^d) = O(b^d)$
prostorová složitost	$O(bd)$

- kombinuje výhody BFS a DFS:
  - nízké paměťové nároky – lineární
  - optimálnost, úplnost
- zdánlivé plýtvání opakovaným generováním
  ALE generuje o jednu úroveň míň, např. pro $b = 10, d = 5$:

$$\begin{aligned} N(\mathrm{IDS}) &= 50 + 400 + 3\,000 + 20\,000 + 100\,000 &&= \quad 123\,450 \\ N(\mathrm{BFS}) &= 10 + 100 + 1\,000 + 10\,000 + 100\,000 + 999\,990 &&= 1\,111\,100 \end{aligned}$$

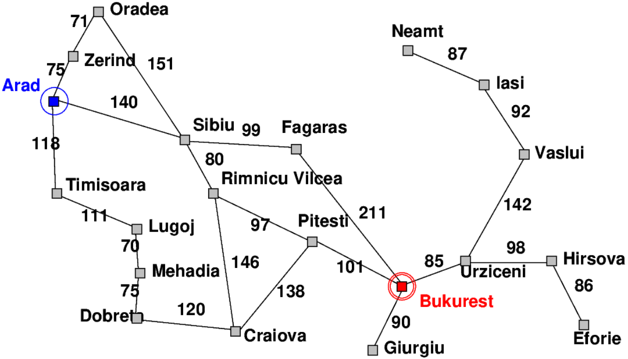
IDS je nejvhodnější neinformovaná strategie pro velké prostory a neznámou hloubku řešení.

Shrnutí vlastností algoritmů neinformovaného prohledávání

Vlastnost	*do hloubky*	*do hloubky s limitem*	*do šířky*	*podle ceny*	*s postupným prohlubováním*
úplnost	ne	ano, pro $l \geq d$	ano*	ano*	ano*
optimálnost	ne	ne	ano*	ano*	ano*
časová složitost	$O(b^m)$	$O(b^\ell)$	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^d)$
prostorová složitost	$O(bm)$	$O(b\ell)$	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bd)$

Příklad – cesta na mapě

Najdi cestu z města Arad do města Bukurest



Příklad – schéma rumunských měst

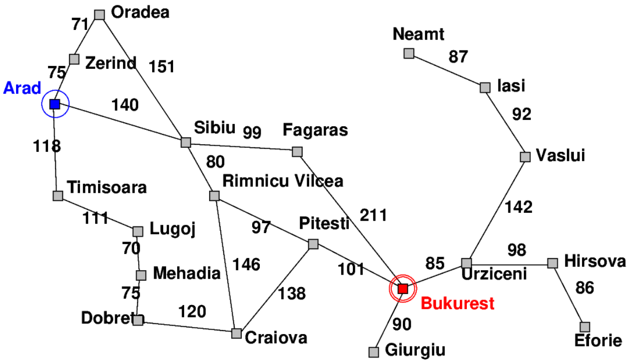
Města:

- Arad
- Bukurest
- Craiova
- Dobreta
- Eforie
- Fagaras
- Giurgiu
- Hirsova
- Iasi
- Lugoj
- Mehadia
- Neamt
- ...

Cesty:

Arad	↔	Timisoara	118
Arad	↔	Sibiu	140
Arad	↔	Zerind	75
Timisoara	↔	Lugoj	111
Sibiu	↔	Fagaras	99
Sibiu	↔	Rimnicu Vilcea	80
Zerind	↔	Oradea	71
...	↔	...	
Giurgiu	↔	Bukurest	90
Pitesti	↔	Bukurest	101
Fagaras	↔	Bukurest	211
Urziceni	↔	Bukurest	85

Příklad – schéma rumunských měst



Arad	366
Bukurest	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

Příklad – cesta na mapě

Neinformované prohledávání:

- DFS, BFS a varianty
- nemá (téměř) žádné informace o pozici cíle – slepé prohledávání
- zná pouze:
  - počáteční/cílový stav
  - přechodovou funkci

Informované prohledávání:

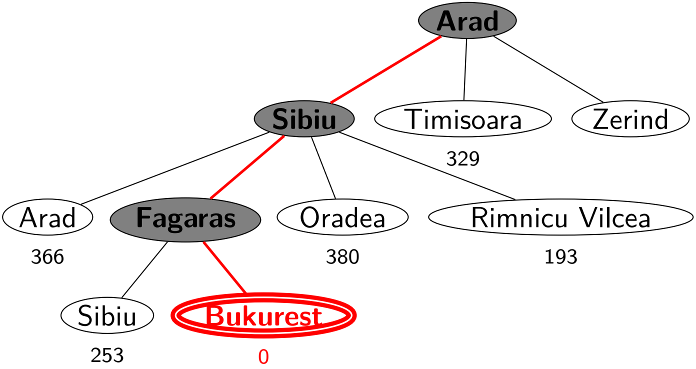
má navíc informaci o (odhadu) blízkosti stavu k cílovému stavu – heuristická funkce (heuristika)

Heuristické hledání nejlepší cesty

- Best-first Search
- použití ohodnocovací funkce $f(n)$ pro každý uzel – výpočet přínosu daného uzlu
- udržujeme seznam uzlů uspořádaný (*vzestupně*) vzhledem k $f(n)$
- použití heuristické funkce $h(n)$ pro každý uzel – odhad vzdálenosti daného uzlu (stavu) od cíle
- čím *menší* $h(n)$, tím blíže k cíli, $h(\mathrm{Goal}) = 0$.
- nejjednodušší varianta – hladové heuristické hledání, *Greedy best-first search*
  $f(n) = h(n)$

Hladové heuristické hledání – příklad

Hledání cesty z města *Arad* do města *Bukurest*
ohodnocovací funkce $f(n) = h(n) = h_{\mathrm{vzd\_Buk}}(n)$, přímá vzdálenost z n do Bukuresti



Hladové heuristické hledání – vlastnosti

- expanduje vždy uzel, který se zdá nejblíže k cíli
- cesta nalezená v příkladu ($g(\text{Arad} \to \text{Sibiu} \to \text{Fagaras} \to \text{Bukurest}) = 450$) je sice úspěšná, ale není optimální ($g(\text{Arad} \to \text{Sibiu} \to \text{RimnicuVilcea} \to \text{Pitesti} \to \text{Bukurest}) = 418$)
-

úplnost	obecně není úplný (nekonečný prostor, cykly)
optimálnost	není optimální
časová složitost	$O(b^m)$, hodně záleží na h
prostorová složitost	$O(b^m)$, každý uzel v paměti

Hledání nejlepší cesty – algoritmus A*

- některé zdroje označují tuto variantu jako Best-first Search
- ohodnocovací funkce – kombinace $g(n)$ a $h(n)$:

$$f(n) = g(n) + h(n)$$

$g(n)$	je cena cesty do n
$h(n)$	je odhad ceny cesty z n do cíle
$f(n)$	je odhad ceny nejlevnější cesty, která vede přes n

- A* algoritmus vyžaduje tzv. přípustnou (*admissible*) heuristiku:

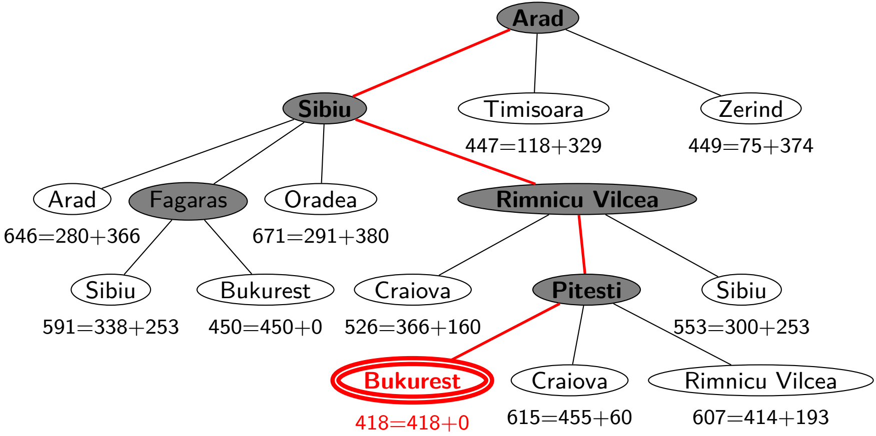
$$0 \leq h(n) \leq h^*(n),\ \text{kde } h^*(n) \text{ je skutečná cena cesty z } n \text{ do cíle}$$

tj. odhad se volí vždycky kratší nebo roven ceně libovolné možné cesty do cíle

Např. přímá vzdálenost $h_{\mathrm{vzd\_Buk}}$ nikdy není delší než (jakákoliv) cesta

Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*
ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\mathrm{vzd\_Buk}}(n)$



Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$

  A* expanduje všechny uzly s $f(n) < C^*$
  A* expanduje některé uzly s $f(n) = C^*$
  A* neexpanduje žádné uzly s $f(n) > C^*$

- *úplnost* je úplný (pokud [počet uzlů s $f < C^*$] $\neq \infty$, tedy cena $\geq \epsilon$ a b konečné)

  optimálnost je optimální

  časová složitost $O((b^*)^d)$, exponenciální v délce řešení d

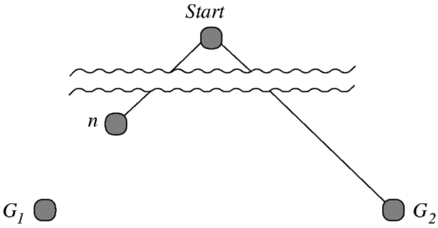
  b^* ... tzv. *efektivní faktor větvení*, viz dále

  prostorová složitost $O((b^*)^d)$, každý uzel v paměti

Problém s prostorovou složitostí řeší algoritmy jako *IDA**, *RBFS*

Důkaz optimálnosti algoritmu A*

- předpokládejme, že byl vygenerován nějaký suboptimální cíl G_2 a je uložen ve frontě.
- dále nechť n je neexpandovaný uzel na nejkratší cestě k optimálnímu cíli G_1 (tj. chybně neexpandovaný uzel ve správném řešení)



Pak

$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{protože } h(G_2) = 0 \\
&> g(G_1) && \text{protože } G_2 \text{ je suboptimální} \\
&\geq f(n) && \text{protože } h \text{ je přípustná}
\end{aligned}
$$

tedy $f(G_2) > f(n)$ $\Rightarrow$ A* nikdy nevybere G_2 pro expanzi dřív než expanduje n $\rightarrow$ spor s předpokladem, že n je *neexpandovaný uzel* $\square$

Hledání nejlepší cesty – algoritmus A*

řešení pomocí prioritní fronty

f-hodnota, *g*-hodnota a cesta k aktuálnímu uzlu

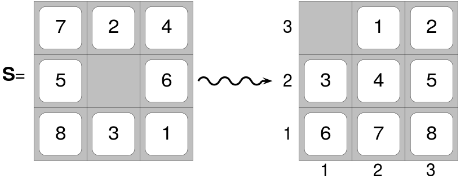
```
function A*SEARCH(problem)
    process_heap ← [(0, 0, [problem.init_state ])]     # prioritní fronta podle f
    while length(process_heap) > 0 do
        f, g, current_path ← process_heap.heap_pop()     # nejmenší f-hodnota
        current_node ← current_path.last ()
        if problem.is_goal (current_node) then
            print current_path     # vypíše cestu k řešení
        foreach child , cost in problem.moves(current_node) do
            if child ∉ current_path then     # detekce cyklů - efektivnější
                process_heap.heap_add( (g+cost+h(child),
                                        g+cost,
                                        current_path + [child]))
```

Příklad – řešení posunovačky

konfigurace = seznam souřadnic **(x, y)**: [pozice$_{díry}$, pozice$_{kámen\ č.1}$, ...]

```
8p. init_state ← [(2,2), (3,1), (2,3), (2,1),
    (3,3), (1,2), (3,2), (1,3), (1,1)]

function 8P.IS_GOAL(S)
    return S = [(1,3), (2,3), (3,3), (1,2),
        (2,2), (3,2), (1,1), (2,1), (3,1)]
```




```
function 8P.MOVES(state)  # pohyby mezery, cena vždy 1
    x_b, y_b ← state.first()            # pozice mezery
    numbers ← state.without_first()     # pozice čísel
    moves ← []
    if x_b > 1 then
        x_nb ← x_b -1; moves.append([(x_nb, y_b)] + numbers.replace((x_nb, y_b), (x_b, y_b)))
    if x_b < 3 then
        x_nb ← x_b + 1; moves.append([(x_nb, y_b)] + numbers.replace((x_nb, y_b), (x_b, y_b)))
    if y_b > 1 then
        y_nb ← y_b -1; moves.append([(x_b, y_nb)] + numbers.replace((x_b, y_nb), (x_b, y_b)))
    if y_b < 3 then
        y_nb ← y_b + 1; moves.append([(x_b, y_nb)] + numbers.replace((x_b, y_nb), (x_b, y_b)))
    return map move in moves to (move, 1)
```

Příklad – řešení posunovačky pokrač.

Volba přípustné heuristické funkce h:

- $h_1(n) =$ počet dlaždiček, které nejsou na svém místě $h_1(\mathbf{S}) = 8$
- $h_2(n) =$ součet manhattanských vzdáleností dlaždic od svých správných pozic $h_2(\mathbf{S}) = 3_7 + 1_2 + 2_4 + 2_5 + 3_6 + 2_8 + 2_3 + 3_1 = 18$

h_1 i h_2 jsou přípustné ... $h^*(S) = 26$

A∗Search(8p):

```
[[(2,2), (3,1), (2,3), (2,1), (3,3), (1,2), (3,2), (1,3), (1,1)],
 [(1,2), (3,1), (2,3), (2,1), (3,3), (2,2), (3,2), (1,3), (1,1)],
 ...
 [(1,2), (2,3), (3,3), (1,3), (2,2), (3,2), (1,1), (2,1), (3,1)],
 [(1,3), (2,3), (3,3), (1,2), (2,2), (3,2), (1,1), (2,1), (3,1)]]
```

Jak najít přípustnou heuristickou funkci?

- je možné najít obecné pravidlo, jak objevit heuristiku h_1 nebo h_2?
- h_1 i h_2 jsou délky cest pro zjednodušené verze problému Posunovačka:
  - při přenášení dlaždice kamkoliv – h_1=počet kroků nejkratšího řešení
  - při posouvání dlaždice kamkoliv o 1 pole (i na plné) – h_2=počet kroků nejkratšího řešení
- relaxovaný problém – méně omezení na akce než původní problém

  Cena optimálního řešení relaxovaného problému je přípustná heuristika pro původní problém.

  optimální řešení původního problému = řešení relaxovaného problému

Posunovačka a relaxovaná posunovačka:

- dlaždice se může přesunout z A na B $\Leftrightarrow$ A sousedí s B $\wedge$ B je prázdná
- (a) dlaždice se může přesunout z A na B $\Leftrightarrow$ A sousedí s B . . h_2
  (b) dlaždice se může přesunout z A na B $\Leftrightarrow$ B je prázdná . . . Gaschnigova h.
  (c) dlaždice se může přesunout z A na B . . . . h_1

Určení kvality heuristiky

efektivní faktor větvení b^* – $N\ldots$ počet vygenerovaných uzlů, $d\ldots$ hloubka řešení, idealizovaný strom s $N+1$ uzly má faktor větvení b^* (reálné číslo):

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

např.: když A* najde řešení po 52 uzlech v hloubce 5 ... $b^* = 1.92$
heuristika je tím lepší, čím blíže je b^* hodnotě 1.

☞ měření b^* na množině testovacích sad – dobrá představa o přínosu heuristiky

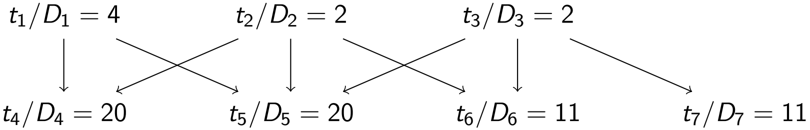
8-posunovačka

	Průměrný počet uzlů			Efektivní faktor větvení b^*		
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
6	680	20	18	2.73	1.34	1.30
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
18	–	3056	363	–	1.46	1.26
24	–	39135	1641	–	1.48	1.26

h_2 dominuje h_1 $(\forall n : h_2(n) \geq h_1(n))$ ... h_2 je lepší (nebo stejná) než h_1 ve všech případech

Příklad – rozvrh práce procesorů

- úlohy t_i s potřebným časem na zpracování D_i (např.: $i = 1, \ldots, 7$)
- m procesorů (např.: $m = 3$)
- relace precedence mezi úlohami – které úlohy mohou začít až po skončení dané úlohy



- problém: najít rozvrh práce pro každý procesor s minimalizací celkového času

	0 2	4	13	24 33
CPU_1	t_3	$\Longleftarrow t_6 \Longrightarrow$	$\Longleftarrow t_5 \Longrightarrow$	
CPU_2	t_2	$\Longleftarrow t_7 \Longrightarrow$	....	
CPU_3	$t_1 \Rightarrow$	$\Longleftarrow t_4 \Longrightarrow$		.....

	0 2	4	13	24	33
CPU_1	t_3	$\Longleftarrow t_6 \Longrightarrow$	$\Longleftarrow t_7 \Longrightarrow$	.....	
CPU_2	t_2 .	$\Longleftarrow t_5 \Longrightarrow$		.....	
CPU_3	$t_1 \Rightarrow$	$\Longleftarrow t_4 \Longrightarrow$		.....	

Příklad – rozvrh práce procesorů – pokrač.

- stavy: **(nezařazené_úlohy, běžící_úlohy, čas_ukončení)**

  př.: **([(***WaitingT*$_1$,*D*$_1$**),(***WaitingT*$_2$,*D*$_2$**),...], [(***Task*$_1$,*F*$_1$**),(***Task*$_2$,*F*$_2$**),(***Task*$_3$,*F*$_3$**)],** *FinTime***)**

  běžící_úlohy udržujeme setříděné $F_1 \leq F_2 \leq F_3$

- počáteční uzel:

```
proc.init_state ← ([("t1",4), ("t2",2), ("t3",2), ("t4",20), ("t5",20), ("t6",11),
                    ("t7",11)], [("idle",0), ("idle",0), ("idle",0)], 0)
```

- přechodová funkce **proc.moves(Stav) → nové stavy s cenami**:

```
function PROC.MOVES (state)
    waiting, active, fintime = state
    moves ← []
    for task in waiting do  # přednost v čekajících nebo nedokončených
        if not check_precedence_waiting(task, waiting.without(task)) then next
        if not check_precedence_active(task, active) then next
        newactive, newfintime ← active.without_first().insert_sorted(task)
        moves.append((waiting.without(task), newactive, newfintime))
    moves ← moves + insert_idle(waiting, active, fintime)  # čekání na procesor
    return moves
```

- cílová podmínka

```
function PROC.IS_GOAL (state)
    return length(state[1]) = 0  # žádné čekající
```

Příklad – rozvrh práce procesorů – pokrač.

- heuristika

  optimální (nedosažitelný) čas:

  $$\textbf{Finall} = \frac{\sum_i D_i + \sum_j F_j}{m}$$

  skutečný (průběžný) čas výpočtu:

  $$\textbf{Fin} = \max(F_j)$$

  heuristická funkce

  $$h = \begin{cases} \textbf{Finall} - \textbf{Fin}, & \text{když } \textbf{Finall} > \textbf{Fin} \\ 0, & \text{jinak} \end{cases}$$

```
function PROC.H (state)
  tasks, processors, fintime ← state
  total_task_time ← Σ_task_time(tasks) # čas ke zpracování
  total_proc_time ← Σ_proc_time(processors) # zpracovaný čas
  finall ← (total_task_time + total_proc_time)/length(processors)
  if finall > fintime then
      return finall - fintime
  return 0
```

Příklad – rozvrh práce procesorů – pokrač.

A*Search(proc):

[([(t_1,4),(t_2,2),(t_3,2),(t_4,20),(t_5,20),(t_6,11),(t_7,11)],	[(idle,0),(idle,0),(idle,0)],	0),
([(t_1,4),(t_2,2),(t_4,20),(t_5,20),(t_6,11),(t_7,11)],	[(idle,0),(idle,0),(t_3,2)],	2),
([(t_1,4),(t_4,20),(t_5,20),(t_6,11),(t_7,11)],	[(idle,0),(t_2,2),(t_3,2)],	2),
([(t_4,20),(t_5,20),(t_6,11),(t_7,11)],	[(t_2,2),(t_3,2),(t_1,4)],	4),
([(t_4,20),(t_5,20),(t_6,11)],	[(t_3,2),(t_1,4),(t_7,13)],	13),
([(t_4,20),(t_5,20),(t_6,11)],	[(idle,4),(t_1,4),(t_7,13)],	13),
([(t_5,20),(t_6,11)],	[(t_1,4),(t_7,13),(t_4,24)],	24),
([(t_6,11)],	[(t_7,13),(t_5,24),(t_4,24)],	24),
([],	[(t_6,24),(t_5,24),(t_4,24)],	24)]