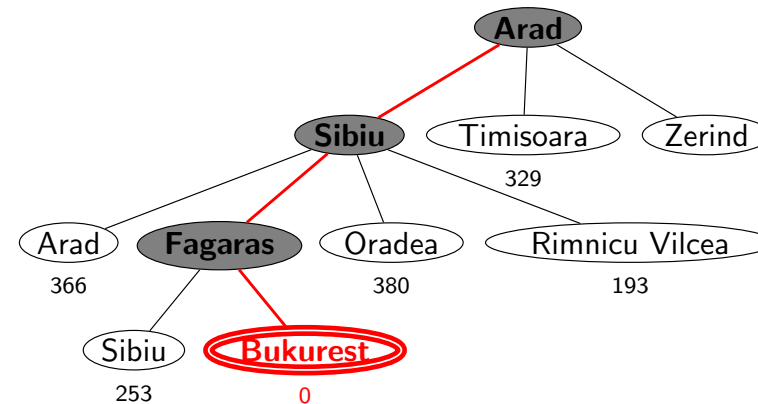


Heuristické hledání nejlepší cesty

- ▶ 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, $h(\text{Goal}) = 0$.
- ▶ 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 $f(n) = h(n) = h_{\text{vzd_Buk}}(n)$, **přímá vzdálenost** z n do Bukuresti



Hladové heuristické hledání – vlastnosti

- ▶ expanduje vždy uzel, který **se zdá** nejbližší k cíli
- ▶ cesta nalezená v příkladu ($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bukurest}) = 450$) je sice úspěšná, ale **není optimální**
($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Rimnicu Vilcea} \rightarrow \text{Pitesti} \rightarrow \text{Bukurest}) = 418$)
- ▶ **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:

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

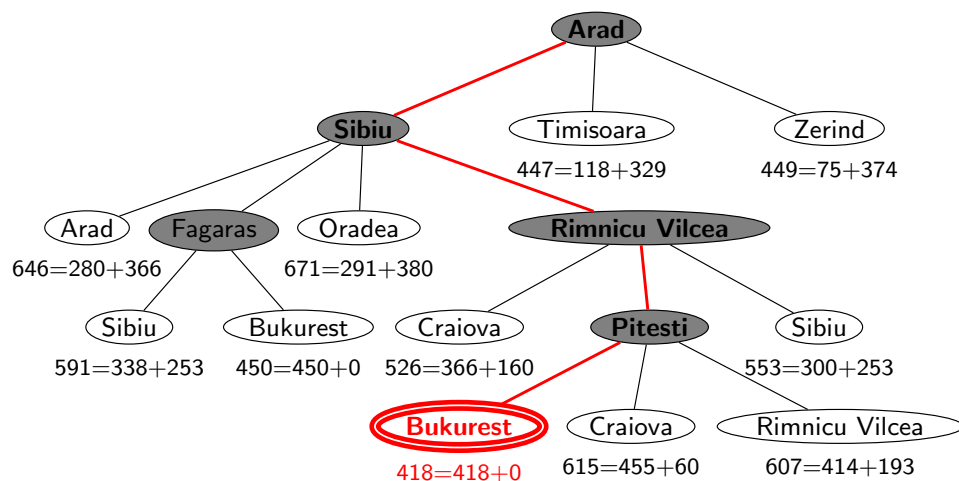
tj. odhad se volí vždycky **kratší** nebo roven ceně libovolné **možné** cesty do cíle

Např. přímá vzdálenost $h_{\text{vzd_Buk}}$ nikdy není delší než (jakákoliv) cesta

Heuristické hledání A* – příklad

Hledání cesty z města *Arad* do města *Bukurest*

ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{vzd.Buk}(n)$



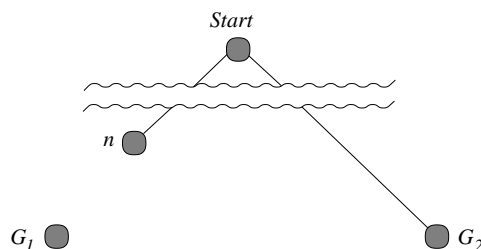
Hledání nejlepší cesty A* – vlastnosti

- ▶ 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 $f < C^*$] $\neq \infty$)
- optimálnost** je optimální
- časová složitost** $O((b^*)^d)$, exponenciální v délce řešení d
 $b^* \dots$ 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 necht' n je **neexpandovaný uzel** na nejkratší cestě k **optimálnímu cíli G_1** (tj. **chybně neexpandovaný uzel** ve správném řešení)



Pak

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

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*

reprezentace uzlů:

- ▶ Prolog: **I(N,F/G)** ... Python: **trojice (n, f, g)** ...
listový uzel N, $F = f(N) = G + h(N)$, $G = g(N)$
- ▶ Prolog: **t(N,F/G,Subs)** ... Python: **čtveřice (n, f, g, subs)** ...
podstrom s kořenem **N**, **Subs** podstromy seřazené podle f , $G = g(N)$ a $F = f$ -hodnota nejnadhřejšího následníka N

horní závora pro cenu nejlepší cesty
biggest = 9999

```
def best_search(start):
    for _, solved, sol in expand(None, (start, 0, 0), biggest):
        if solved == "yes":
            yield sol
```

```
expand(path,tree,bound)
path – cesta mezi kořenem a tree
tree – prohledávaný podstrom
bound – f-limita pro expandování Tr
vrací trojici:
tree1 – tree expandovaný až po bound
solved – 'yes', 'no', 'never'
sol – cesta z kořene do cílového uzlu
```

@odpovídá return pro generátorovou funkci

pokrač. →

Hledání nejlepší cesty – algoritmus A* – pokrač.

```
def expand(path, tree, bound):
    if len(tree) == 3: # listový uzel
        node, f, g = tree
        if is_goal(node): yield (None, "yes", (f, Cons(node, path)))
        if f <= bound:
            succ = Nil
            for m, c in move_anyYC(node):
                if not member(m, path): succ = Cons((m, c), succ)
            if succ == Nil: yield (None, "never", None)
            else:
                trees = succlist(g, succ)
                f1 = bestf(trees)
                for tree1, solved, sol in expand(path, (node, f1, g, trees), bound):
                    yield (tree1, solved, sol)
        elif f > bound: yield (tree, "no", None)
    else: # stromový uzel
        node, f, g, trees = tree
        if trees == Nil: yield (None, "never", None)
        else:
            if f <= bound:
                bound1 = min(bound, bestf(trees.tail))
                for t1, solved1, sol1 in expand(Cons(node, path), trees.head, bound1):
                    for tree1, solved, sol in continue_(path, (node, f, g, Cons(t1, trees.tail)),
                                                       bound, solved1, sol1):
                        yield (tree1, solved, sol)
            elif f > bound: yield (tree, "no", None)
```

vrací trojici: (tree1, solved, sol)

succlist setřídí seznam listů podle f-hodnot

continue – volba způsobu pokračování podle výsledků expand

pokrač. →

Hledání nejlepší cesty – algoritmus A* – pokrač.

```
def continue_(path, tree, bound, subtr_solved, sol):
    node, _, g, trees = tree
    if subtr_solved == "yes":
        yield (None, "yes", sol)
    elif subtr_solved == "no":
        nts = insert(trees.head, trees.tail)
        f1 = bestf(nts)
        for tree1, solved, sol in expand(path, (node, f1, g, nts), bound):
            yield (tree1, solved, sol)
    elif subtr_solved == "never":
        f1 = bestf(trees.tail)
        for tree1, solved, sol in expand(path, (node, f1, g, trees.tail), bound):
            yield (tree1, solved, sol)
```

continue – volba způsobu pokračování podle výsledků expand

pokrač. →

Hledání nejlepší cesty – algoritmus A* – pokrač.

```
def succlist(g0, succ):
    if succ == Nil: return Nil
    n, c = succ.head
    g = g0 + c
    f_ = g + h(n)
    ts1 = succlist(g0, succ.tail)
    ts = insert((n, f_, g), ts1)
    return ts

def f(tree):
    if len(tree) == 3: # listový uzel
        _, f_, _ = tree
    else: # stromový uzel
        _, f_, _, _ = tree
    return f_

def bestf(trees):
    if trees == Nil: return biggest
    return f(trees.head)

def insert(t, ts):
    if f(t) <= bestf(ts): return Cons(t, ts)
    return Cons(ts.head, insert(t, ts.tail))
```

succlist(g0, succ) setřídí seznam listů podle f-hodnot
vrací setříděný seznam

“vytáhne” f ze struktury

nejlepší f-hodnota ze seznamu stromů

vloží t do seznamu stromů ts podle f

Hledání nejlepší cesty – algoritmus A* – heapq

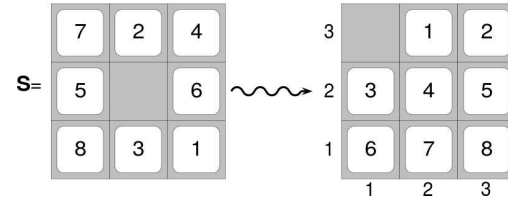
řešení pomocí modulu `heapq` – implementace **prioritní fronty**`import heapq`

```
def best_search(start):
    heap = [(0, 0, start, Nil)]
    while True:
        try:
            f, g, node, path = heapq.heappop(heap)
        except IndexError: # fronta je prázdná
            break
        path1 = Cons(node, path)
        if is_goal(node):
            yield (f, path1)
        if f <= biggest:
            for m, c in move_anyYC(node):
                if not member(m, path1):
                    heapq.heappush(heap, (g+c+h(m), g+c, m, path1))
```

Příklad – řešení posunovačky

konfigurace = seznam souřadnic **X/Y**: [pozice_{díry}, pozice_{kámen č.1}, ...]

start([2/2, 3/1, 2/3, 2/1, 3/3,
1/2, 3/2, 1/3, 1/1]).
goal([1/3, 2/3, 3/3, 1/2, 2/2,
3/2, 1/1, 2/1, 3/1]).



move(+Uzel, -NasiUzel, -Cena) pomocí pohybů mezery (cena vždy 1)

```
move([XB/YB | Numbers], [XL/YB | NewNumbers], 1) :- % doleva
    XB>1, XL is XB - 1, replace(XL/YB, XB/YB, Numbers, NewNumbers).
move([XB/YB | Numbers], [XR/YB | NewNumbers], 1) :- % doprava
    XB<3, XR is XB + 1, replace(XR/YB, XB/YB, Numbers, NewNumbers).
move([XB/YB | Numbers], [XB/YD | NewNumbers], 1) :- % dolu
    YB>1, YD is YB - 1, replace(XB/YD, XB/YB, Numbers, NewNumbers).
move([XB/YB | Numbers], [XB/YU | NewNumbers], 1) :- % nahoru
    YB<3, YU is YB + 1, replace(XB/YU, XB/YB, Numbers, NewNumbers).

% replace(+Co, +Cim, +Seznam, -NovySeznam)
replace(Co,Cim,[Co|T],[Cim|T):- !.
replace(Co,Cim,[H|T1],[H|T2]) :- replace(Co,Cim,T1,T2).
```

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 ⇔ A sousedí s B ∧ B je prázdná
- ▶ (a) dlaždice se může přesunout z A na B ⇔ A sousedí s B ... h_2
- ▶ (b) dlaždice se může přesunout z A na B ⇔ B je prázdná ... Gaschnigova h.
- ▶ (c) dlaždice se může přesunout z A na B h_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 ... $h^*(S) = 26$

```
:- start (Start), bestsearch (Start, Solution),
    reverse (Solution, RSolution), writelist (RSolution).
1: [2/2, 3/1, 2/3, 2/1, 3/3, 1/2, 3/2, 1/3, 1/1]
2: [1/2, 3/1, 2/3, 2/1, 3/3, 2/2, 3/2, 1/3, 1/1]
...
26: [1/2, 2/3, 3/3, 1/3, 2/2, 3/2, 1/1, 2/1, 3/1]
27: [1/3, 2/3, 3/3, 1/2, 2/2, 3/2, 1/1, 2/1, 3/1]
```

Určení kvality heuristiky

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

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

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

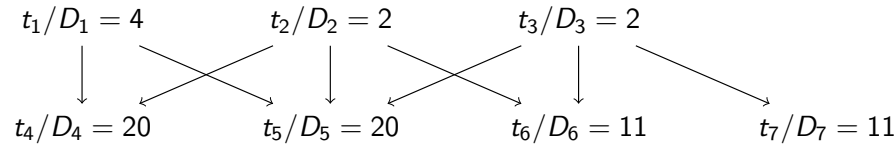
8-posunovačka

d	Průměrný počet uzlů			Efektivní faktor větvení b^*		
	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.: $i = 1, \dots, 7$)
- ▶ 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

	02	4	13	24	33
CPU ₁	t ₃	← t ₆ →	← t ₅ →		
CPU ₂	t ₂	← t ₇ →		
CPU ₃	t ₁	← t ₄ →		

	02	4	13	24	33
CPU ₁	t ₃	← t ₆ →	← t ₇ →	
CPU ₂	t ₂	← t ₅ →		
CPU ₃	t ₁	← t ₄ →		

Příklad – rozvrh práce procesorů – pokrač.

- ▶ počáteční uzel:
`start([t1/4, t2/2, t3/2, t4/20, t5/20, t6/11, t7/11]*[idle/0, idle/0, idle/0]*0).`

- ▶ heuristika
optimální (nedosažitelný) čas:

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

skutečný čas výpočtu:

$$Fin = \max(F_j)$$

heuristická funkce h :

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

```
h(Tasks * Processors * Fin, H) :-
    totaltime(Tasks, Totttime),
    sumnum(Processors, Ftime, N),
    Finall is (Totttime + Ftime)/N,
    (Finall > Fin, !, H is Finall - Fin
     ; H = 0).
```

```
totaltime([], 0).
totaltime([_/_ | Tasks], T) :-
    totaltime(Tasks, T1), T is T1 + D.
```

```
sumnum([], 0, 0).
sumnum([_/_ | Procs], FT, N) :-
    sumnum(Procs, FT1, N1),
    N is N1 + 1, FT is FT1 + T.
```

```
precedence(t1, t4). precedence(t1, t5).
```

...

Příklad – rozvrh práce procesorů – pokrač.

- ▶ stavy: **nezařazené_úlohy*****běžící_úlohy*****čas_ukončení**
např.: `[WaitingT1/D1,WaitingT2/D2,...]*[Task1/F1,Task2/F2,Task3/F3]*FinTime`
- ▶ **běžící_úlohy** udržujeme setříděné $F1 \leq F2 \leq F3$

- ▶ přechodová funkce **move(+Uzel, -NasiUzel, -Cena)**:

```
move(Tasks1*[_/F|Active1]*Fin1, Tasks2*Active2*Fin2, Cost) :-
    del1(Task/D, Tasks1, Tasks2),
    \+ (member(T/_ , Tasks2), before(T, Task)), % kontrola predence v čekajících
    \+ (member(T1/F1, Active1), F < F1, before(T1, Task)), % a v zařazených úlohách
    Time is F + D, insert(Task/Time, Active1, Active2, Fin1, Fin2), Cost is Fin2 - Fin1.
move(Tasks*[_/F|Active1]*Fin, Tasks*Active2*Fin, 0) :- insertidle(F, Active1, Active2).
```

```
before(T1, T2) :- precedence(T1, T2).
before(T1, T2) :- precedence(T, T2), before(T1, T).
```

before(+Task1, +Task2)
tranzitivní obal relace **precedence**

```
insert(S/A, [T/B|L], [S/A, T/B|L], F, F) :- A < B, !.
insert(S/A, [T/B|L], [T/B|L1], F1, F2) :- insert(S/A, L, L1, F1, F2).
insert(S/A, [], [S/A], _, A).
```

```
insertidle(A, [T/B|L], [idle/B, T/B|L]) :- A < B, !.
insertidle(A, [T/B|L], [T/B|L1]) :- insertidle(A, L, L1).
```

```
goal([], *_*_*).
```

Příklad – rozvrh práce procesorů – pokrač.

```
:- start(Start), write('Pocatecni stav:'), write(Start), nl,
    bestsearch(Start, Solution),
    write('Nalezene reseni:'), nl,
    reverse(Solution, RSolution), writelist(RSolution).
```

Pocatecni stav: `[t1/4,t2/2,t3/2,t4/20,t5/20,t6/11,t7/11]*[idle/0,idle/0,idle/0]*0`

Nalezene reseni:

- `[t1/4,t2/2,t3/2,t4/20,t5/20,t6/11,t7/11]*[idle/0,idle/0,idle/0]*0`
- `[t1/4,t2/2,t4/20,t5/20,t6/11,t7/11]*[idle/0,idle/0,t3/2]*2`
- `[t1/4,t4/20,t5/20,t6/11,t7/11]*[idle/0,t2/2,t3/2]*2`
- `[t4/20,t5/20,t6/11,t7/11]*[t2/2,t3/2,t1/4]*4`
- `[t4/20,t5/20,t6/11]*[t3/2,t1/4,t7/13]*13`
- `[t4/20,t5/20,t6/11]*[idle/4,t1/4,t7/13]*13`
- `[t5/20,t6/11]*[t1/4,t7/13,t4/24]*24`
- `[t6/11]*[t7/13,t5/24,t4/24]*24`
- `[]* [t6/24,t5/24,t4/24]*24`