Prohledávání stavového prostoru

Doc.Ing. Jiří Krejsa, PhD

krejsa@fme.vutbr.cz, A2/710

Stavový prostor

Neinformované metody

Informované metody

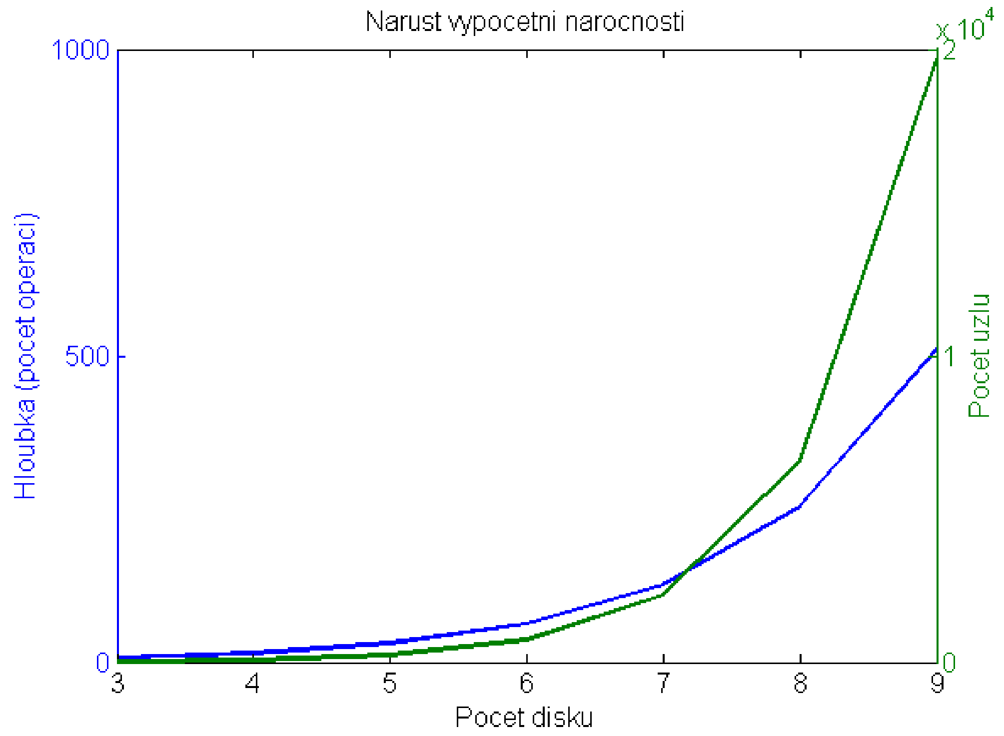
Aplikace

Stavový prostor

- Model prostředí = stav
- Akce -> přechod (změna) stavu
- Množina všech stavů = stavový prostor
- Počáteční stav, koncový stav (může jich být víc, nemusí být popsán explicitně, stačí podmínky které musí splňovat)
- Prohledávání stavového prostoru = nalezení posloupnosti akcí, která vede od počátečního stavu ke koncovému = řešení úlohy
- Problém: stavový prostor může být velmi, velmi, velmi velký, i když je diskrétní

Příklady úloh

- Hledání cesty pro mobilní robot, auto, …
- Hanojská věž, Rubikova kostka, přelévání vody, vlk koza zelí, …
- Pohyb nepřátel v počítačových hrách



Výpočetní náročnost – Hanojská věž

Formální popis problému

S: <S,A,Akce(s),Cil(s),Nasledník(s,a),Cena(s,a),…>

Kde

S: množina všech stavů

A: množina všech akcí

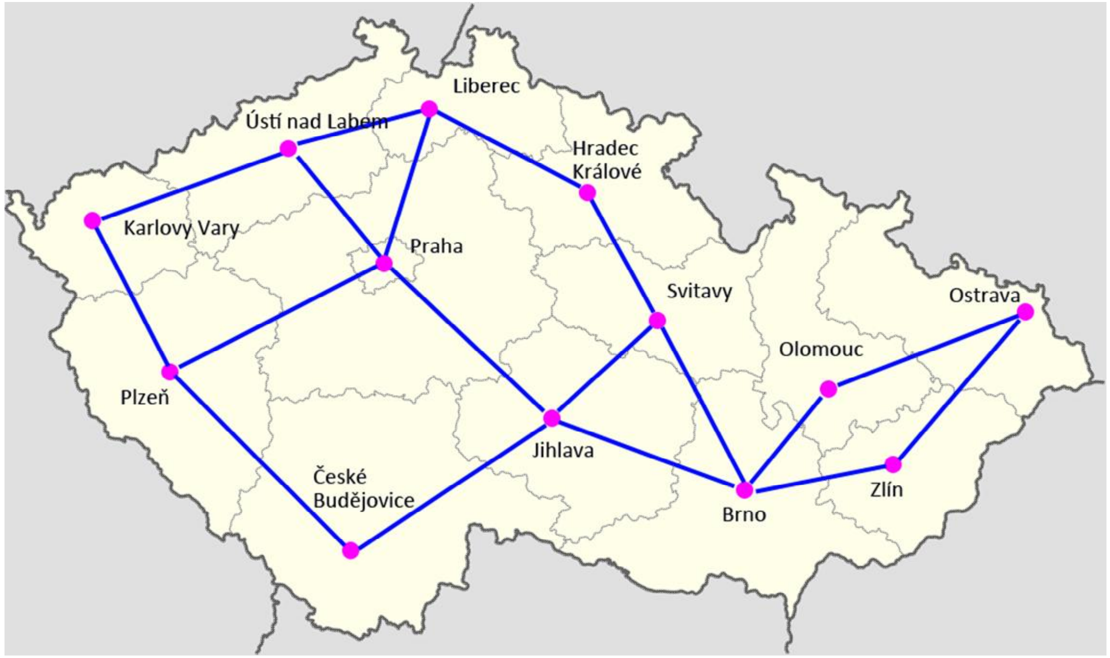
Akce(s): funkce, která určuje kterou akci je možné provést na daném stavu

Cil(s): funkce, která určuje zda je daný stav cílovým stavem

Následník(s,a): funkce, která vrací nový stav po provedení akce a na stavu s

Cena(s,a): funkce, která vrací cenu provedení akce a na stavu s. Může být konstantní

Příklad 1: hledáme cestu z Plzně do Hradce Králové



Akce(Plzeň) = {Karlovy Vary, Praha, České Budějovice }

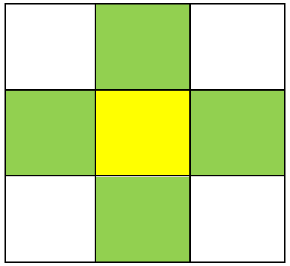
Akce(České Budějovice) = {Plzeň, Jihlava }

Cíl(Liberec) = false

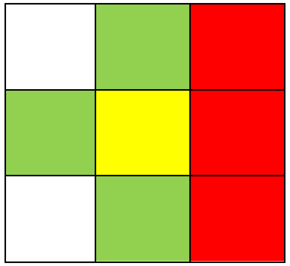
Cena(České Budějovice, Jihlava) = 113 km

Příklad 2: hledáme cestu ze startu do cíle na mřížce

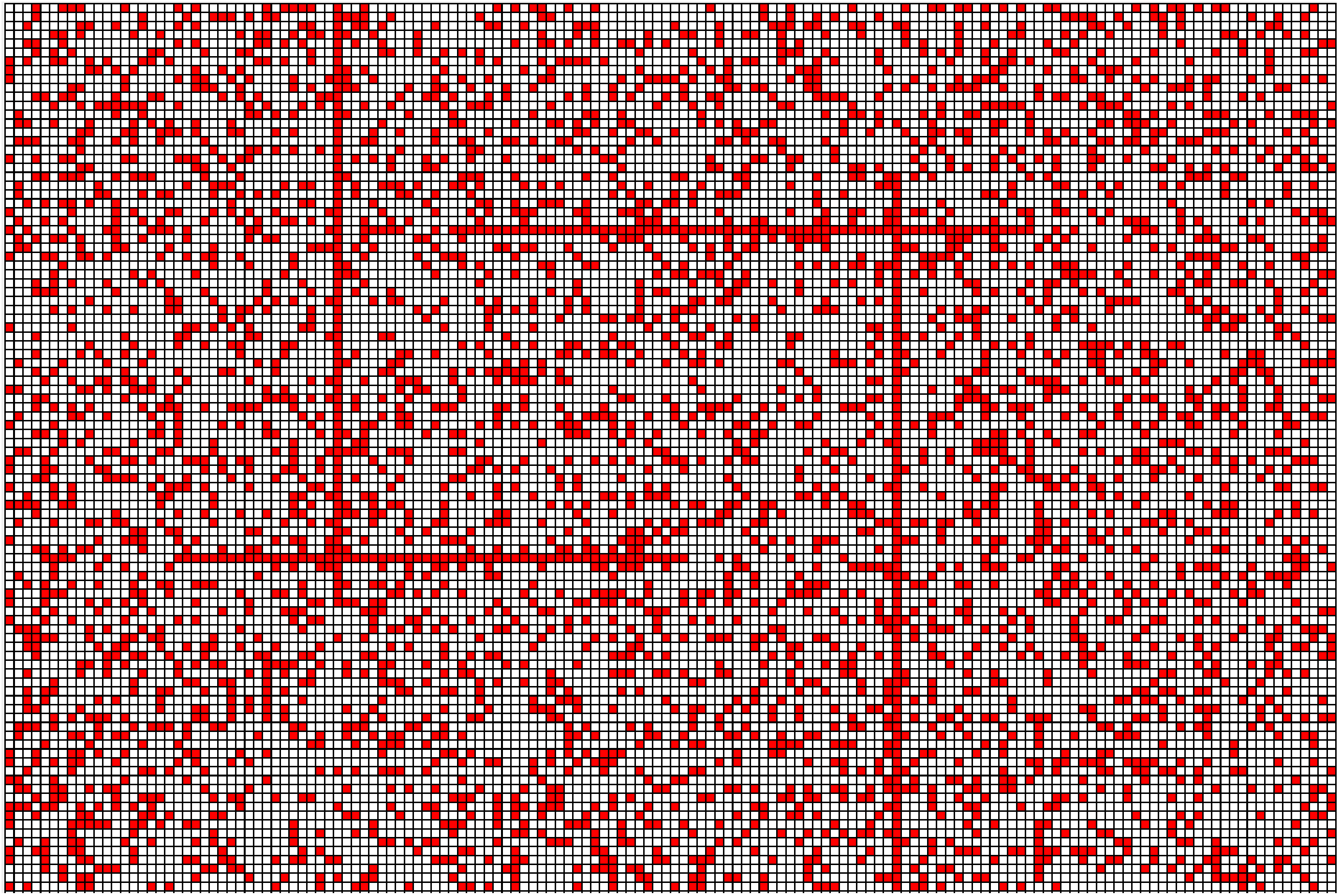
Pohyb pouze po dlaždicích, tedy A= {nahoru, dolů, vlevo, vpravo}



Volný prostor



Překážka (zeď vpravo)



Příklad 3: Hanojská věž

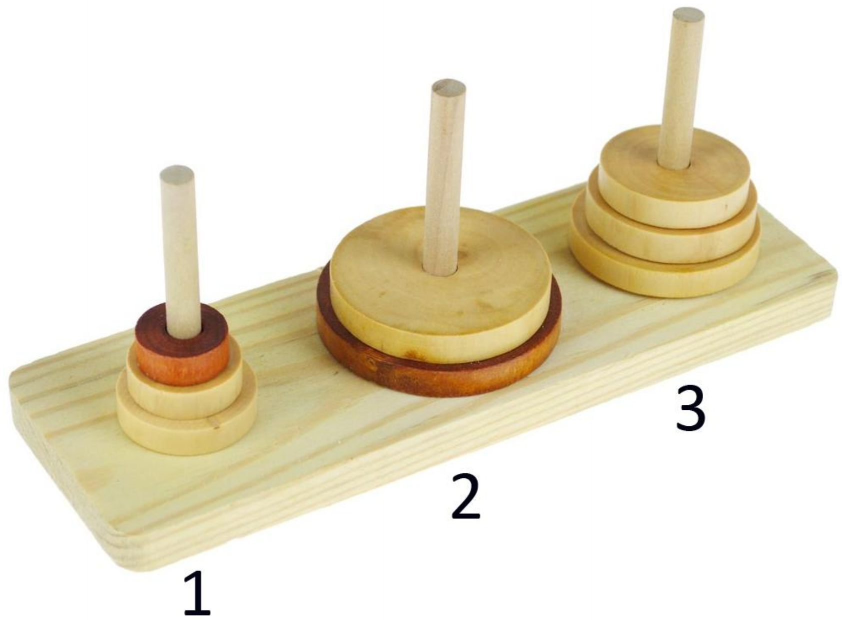
Celková množina akcí je přesun mezi věžemi, tedy

A= {1→2, 1→3, 2→3, 2→1, 3→1, 3→2}

Ale ne vždy jsou všechny akce přípustné

Akce(s)= {1→2, 1→3, 3→2}

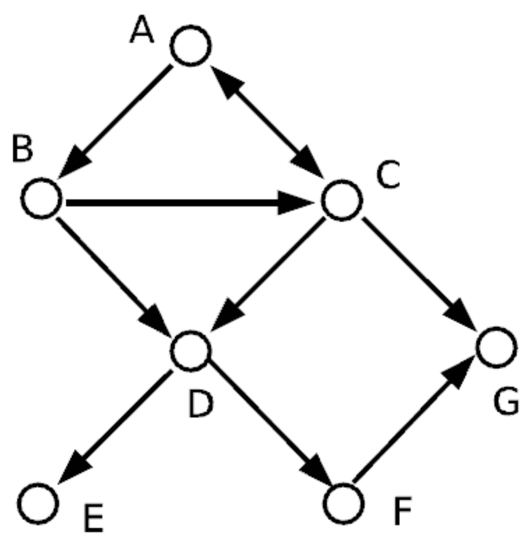
důvod: ostatní akce porušují pravidla

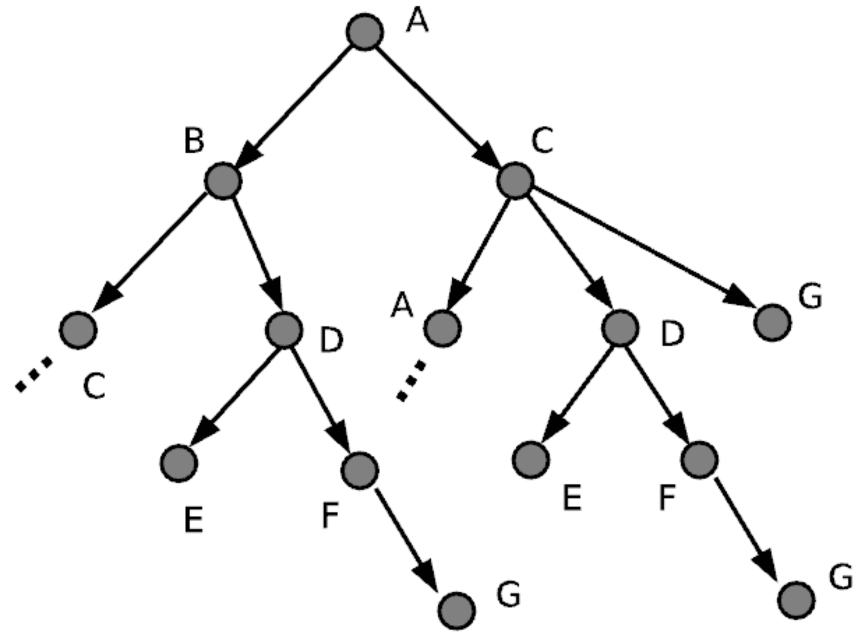


Prohledávání stavového prostoru

- Stavový prostor reprezentujeme orientovaným grafem (oriented graph)
- Uzel grafu (node) reprezentuje stav
- Orientovaná hrana (edge) reprezentuje přechod (transition) mezi stavy
- Řešení úlohy – hledání cesty mezi počátečním uzlem a uzlem cílového stavu
- V grafu se mohou vyskytovat cykly
- Graf může obsahovat cenu (cost) přechodu – váhy hran (weights)
- Prohledáváním grafu vytváříme strom (search tree) s „kmenem“ v počátečním uzlu
- Strom vytváříme expanzí jednotlivých uzlů pomocí akcí
- Dva typy uzlů: již expandované (zavřené, closed), neexpandované (otevřené, open). Oba typy ve zvláštních seznamech.
- **Řídící strategie (search strategy) určuje v jakém pořadí dochází k expanzi**

Rozdíl mezi stavovým prostorem a stromem

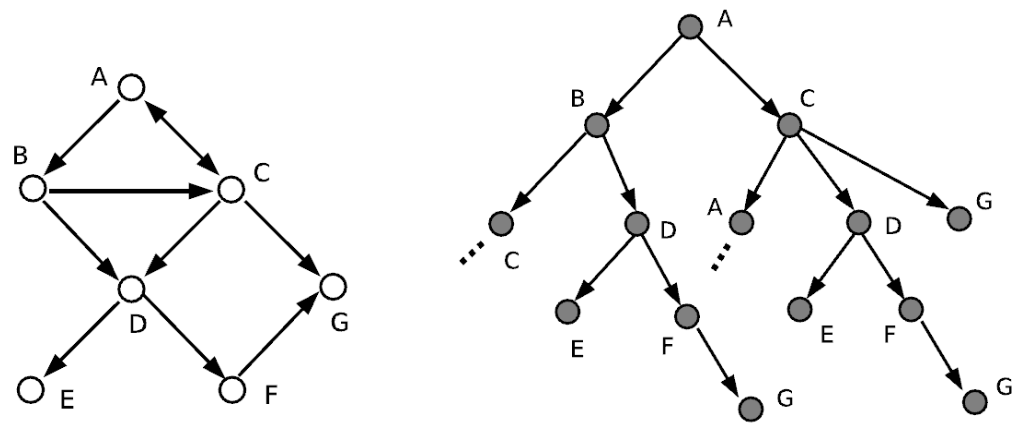




Stavový prostor (reprezentovaný orientovaným grafem) strom řešení

- Strom je generován v průběhu prohledávání
- Strom může být nekonečný (cykly), i když je stavový prostor konečný

Rozdíl mezi stavem a uzlem



- Uzel obsahuje stav PLUS další informace
  - Hloubka uzlu
  - Kořenový uzel má hloubku 0
  - Link na předchozí uzel
  - Pomocí linků snadno zrekonstruujeme cestu od cíle k počátku
- Uzel může obsahovat stav, který už je v jiném uzlu (ale cesta k němu je jiná)

Obecný algoritmus hledání

s – stav

a - akce

O – seznam otevřených uzlů

```
FUNKCE Hledej(s_0,S_G,A)
    Vlož s_0 do O
    WHILE O není prázdný DO
        z O odeber stav s
        IF s je cílový stav, tj. s ∈ S_G
            RETURN s // rešení nalezeno
        ELSE
            s_nový = f(s,a) // expanduj uzel se stavem s
            hloubka nového uzlu je +1
            zařaď s_nový do O
        END IF
    END WHILE
RETURN fail // rešení neexistuje, seznam O je prázdný, stavový prostor je prohledaný úplně)
end funkce
```

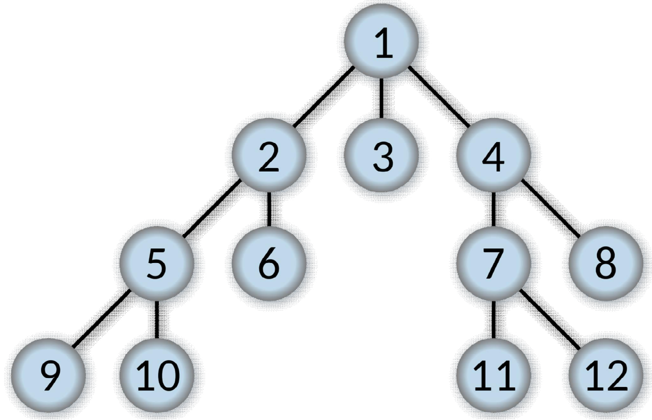
Rozdělení algoritmů

- Strategie rozhoduje o tom, kam v seznamu se řadí nové uzly
- Systematické prohledávání nemusí stačit (velký stavový prostor)
- Můžeme využít všechny informace, které o úloze máme (heuristiky)

Základní dělení

- Neinformované (slepé, blind, uninformed)
  - Prohledávání do šířky
  - Prohledávání do hloubky
  - …
- Informované (heuristické, heuristic, informed)
  - Dijkstrův algoritmus
  - A* algoritmus
  - …

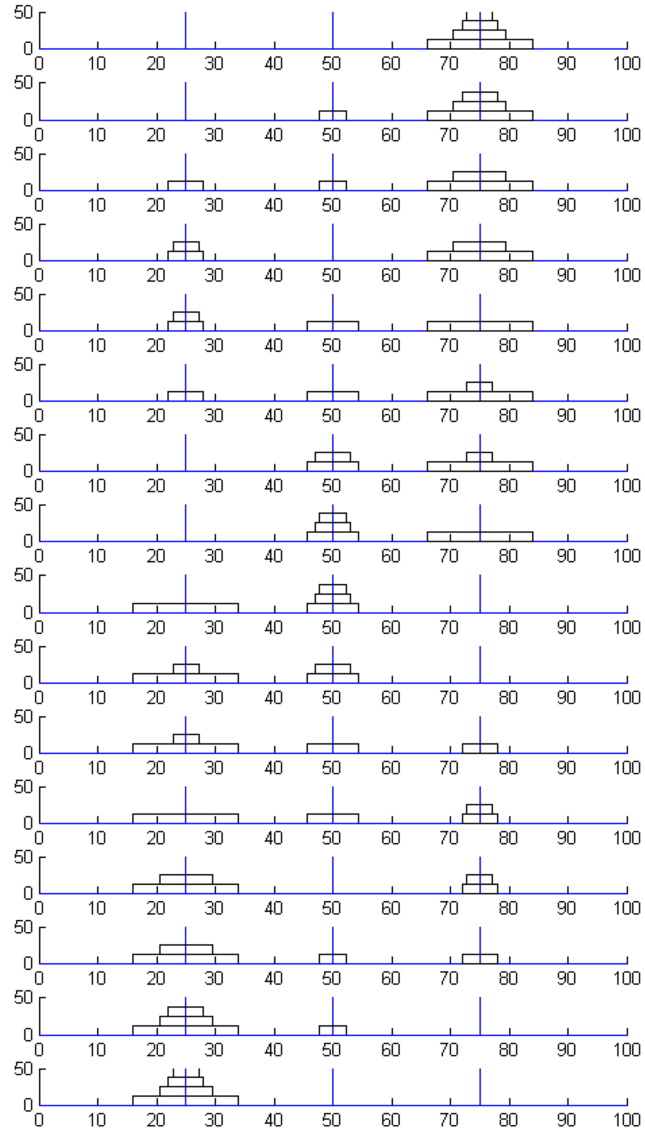
Prohledávání do šířky (breadth-first search)



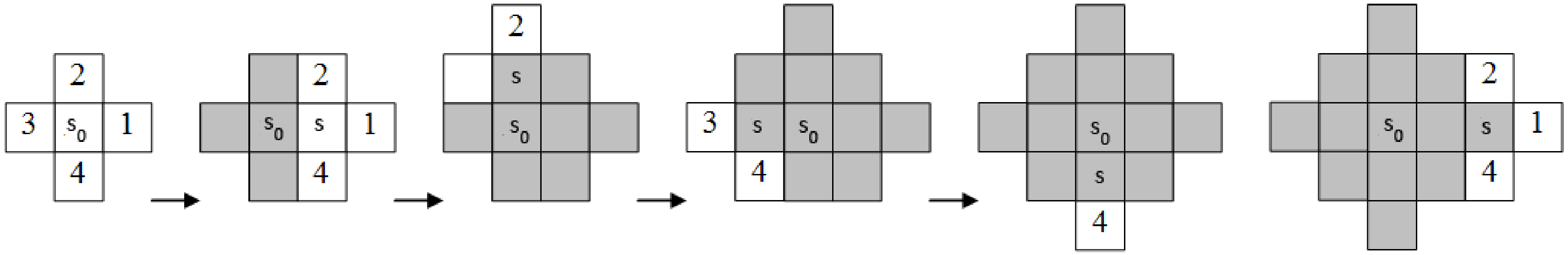
- Nejjednodušší slepé prohledávání
- Expandujeme kořenový uzel
- Postupně expandujeme všechny potomky
- Expanze probíhá po vrstvách
- Seznam O funguje jako fronta (FIFO)
- Zařazení nového stavu (řádek algoritmu č.11) vždy na konec seznamu O
- Nalezené řešení má nejmenší možnou hloubku
- Paměťově a výpočetně náročná metoda
- Použitelná jen na jednoduché úlohy

Příklad: Hanojská věž

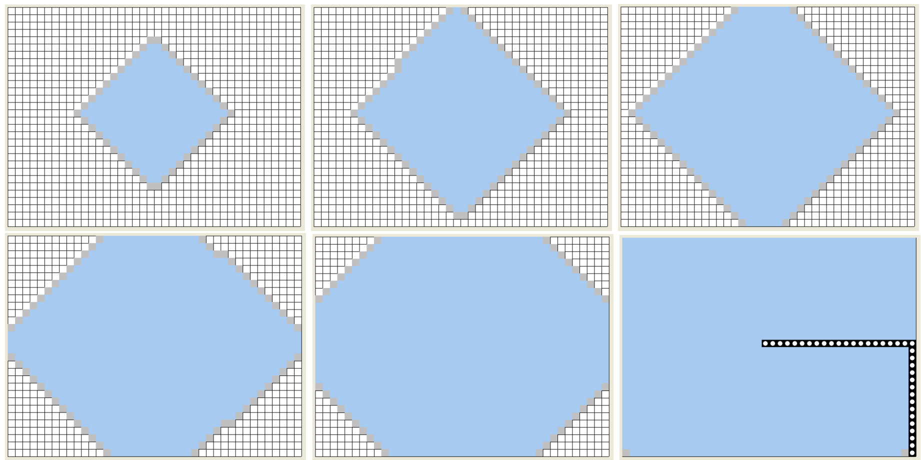
- Stav: trojice seznamů disků na jednotlivých věžích
- Počáteční stav pro 4 disky: s0 = ([4 3 2 1],[],[])
- Cílový stav sG = ([],[],[4 3 2 1])
- 6 akcí: A= {1→2, 1→3, 2→3, 2→1, 3→1, 3→2}



Příklad: Hledání cesty

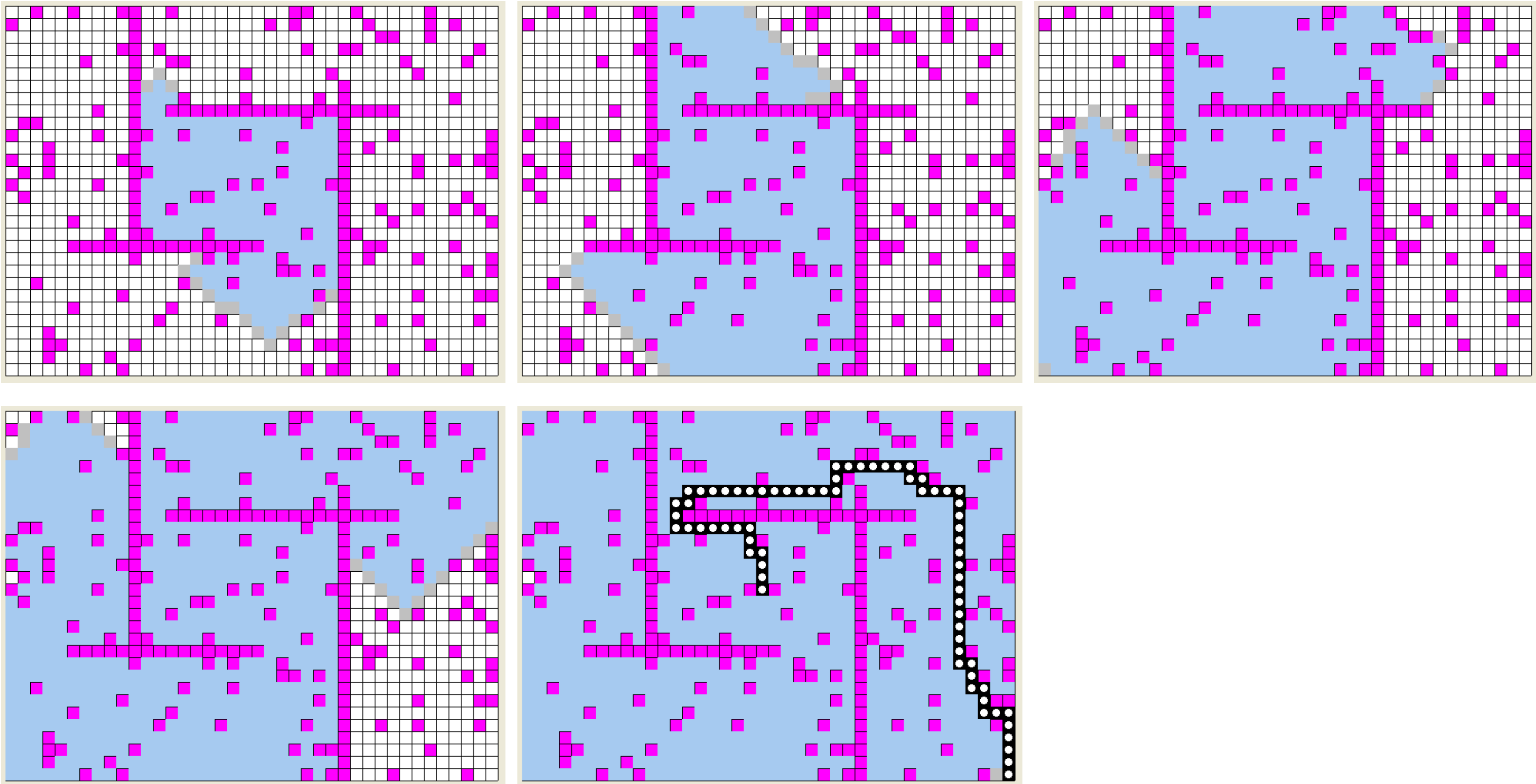


- Stav: poloha robota (souřadnice x,y)
- Počáteční stav: počáteční poloha
- Cílový stav: koncová poloha
- 4 akce: A= {vpravo, nahoru, vlevo, dolů}
- Počátek ve středu, cíl v pravém dolním rohu
- Průběh prohledávání po 200 krocích

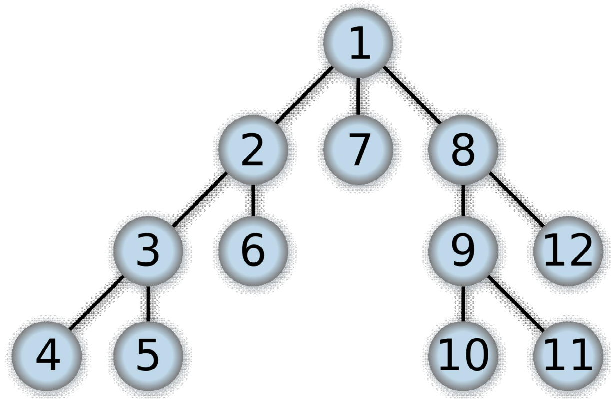


Příklad: Hledání cesty – přidáme překážky

Úloha se nezmění, pouze některé stavy nejsou přípustné (akce nevede k vytvoření nového uzlu, neboť je na daném stavu překážka)

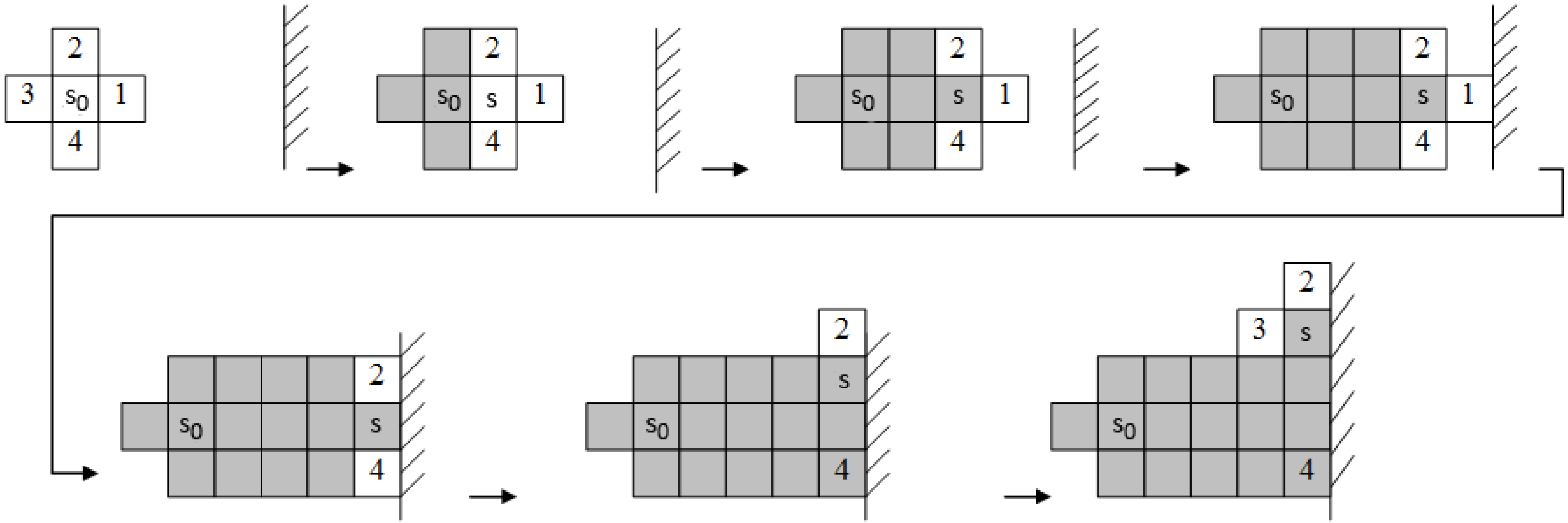


Prohledávání do hloubky (depth-first search)



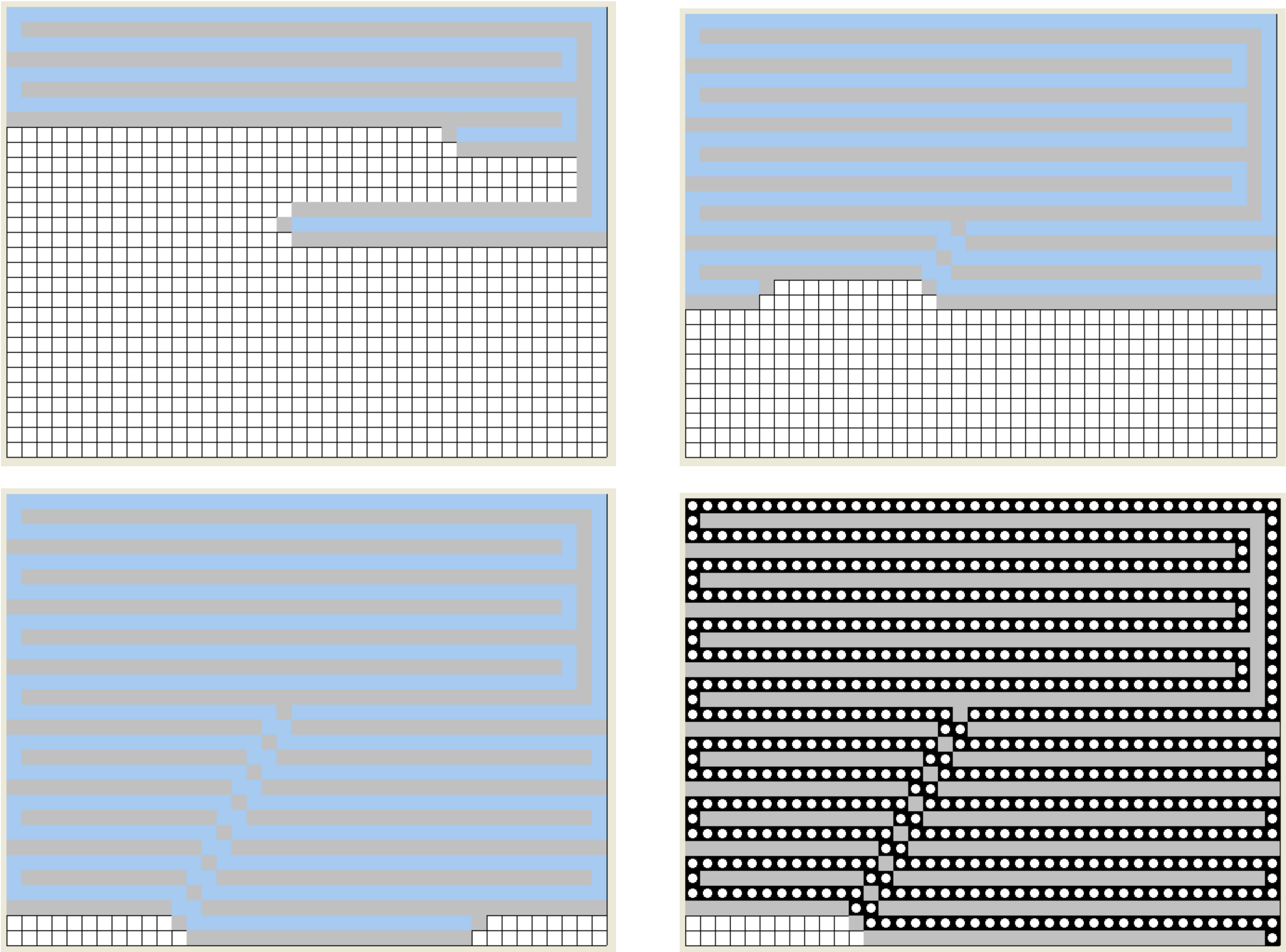
- Expandujeme vždy nejhlubší uzel
- Vracíme se zpět až po dosažení uzlu, ze kterého není možné pokračovat dál
- Seznam O funguje jako zásobník (LIFO)
- Zařazení nového uzlu (řádek 11 obecného algoritmu) na začátek seznamu O
- Paměťově méně náročná metoda
- Nezaručuje optimální řešení
- Vylepšení: omezení maximální hloubky

Příklad: Hledání cesty

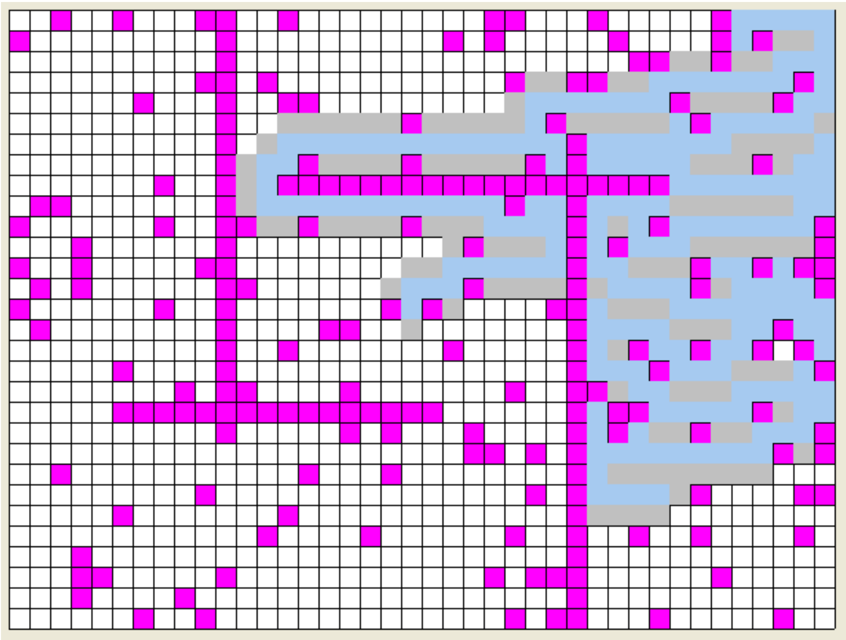
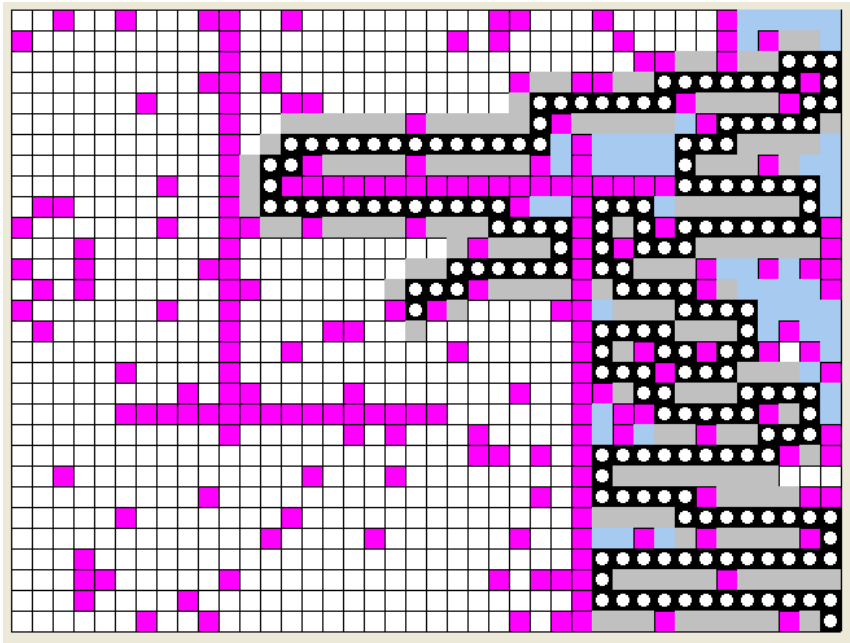


- Stav: poloha robota (souřadnice x,y)
- Počáteční stav: počáteční poloha
- Cílový stav: koncová poloha
- 4 akce: A= {vpravo, nahoru, vlevo, dolů}
- Počátek ve středu, cíl v pravém dolním rohu

Průběh prohledávání po 200 krocích



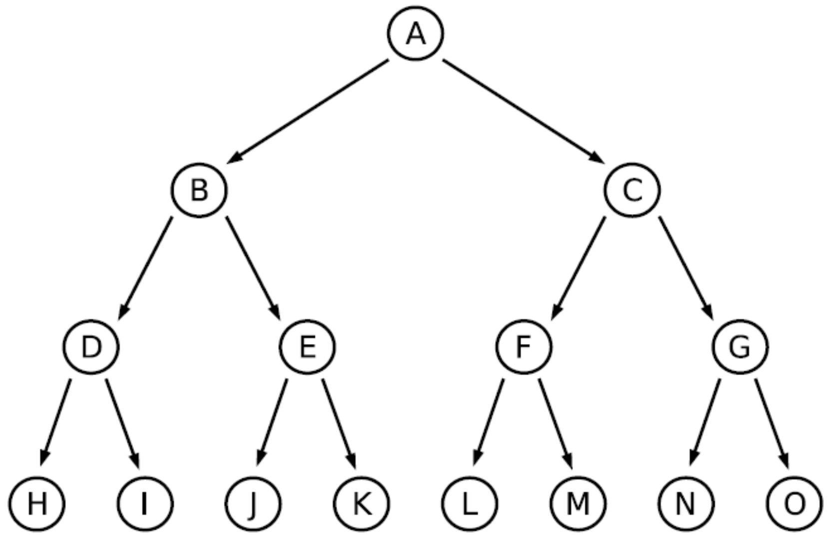
Přidání překážek

- 200 a 238 iterace
- Značná část prostoru není prohledávána
- Cesta není optimální
- Nalezená cesta je silně závislá na počátečních podmínkách

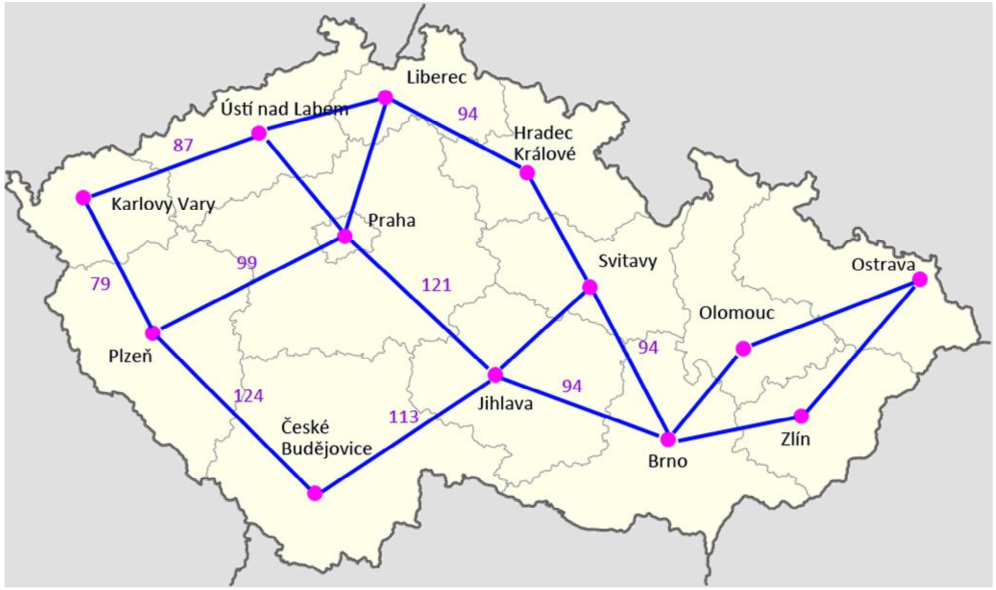
Algoritmus iterativního prohlubování (iterative deepening)

- Prohledávání do hloubky s omezením hloubky
- Projedeme všechny uzly v zadané hloubce. Pokud není nalezeno řešení, zvýšíme hloubku
- Kombinace prohledávání do šířky a do hloubky
- Pozor, projíždíme celý strom od začátku
- Vypadá to jako plýtvání, ale na začátku není uzlů mnoho
- Doporučený algoritmus, pokud neznáme hloubku ve které je řešení

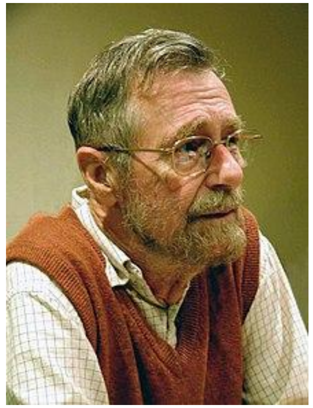


A, A, B, C, A, B, D, E, C, F,
G A, B, D, H, I, E, J, K, . . .

Zahrnutí ceny



- Prozatím měly všechny přechody stejnou cenu. Celková cena řešení tak odpovídala hloubce.
- U některých úloh tomu tak je (Hanojská věž). U některých tomu tak není (hledání cesty z Plzně do Hradce).
- Cenu můžeme zahrnout do funkce Následník(s,a). Nový uzel má místo hloubky **celkovou cenu** od startu k uzlu. Tuto informaci lze použít při výběru uzlu k expanzi.



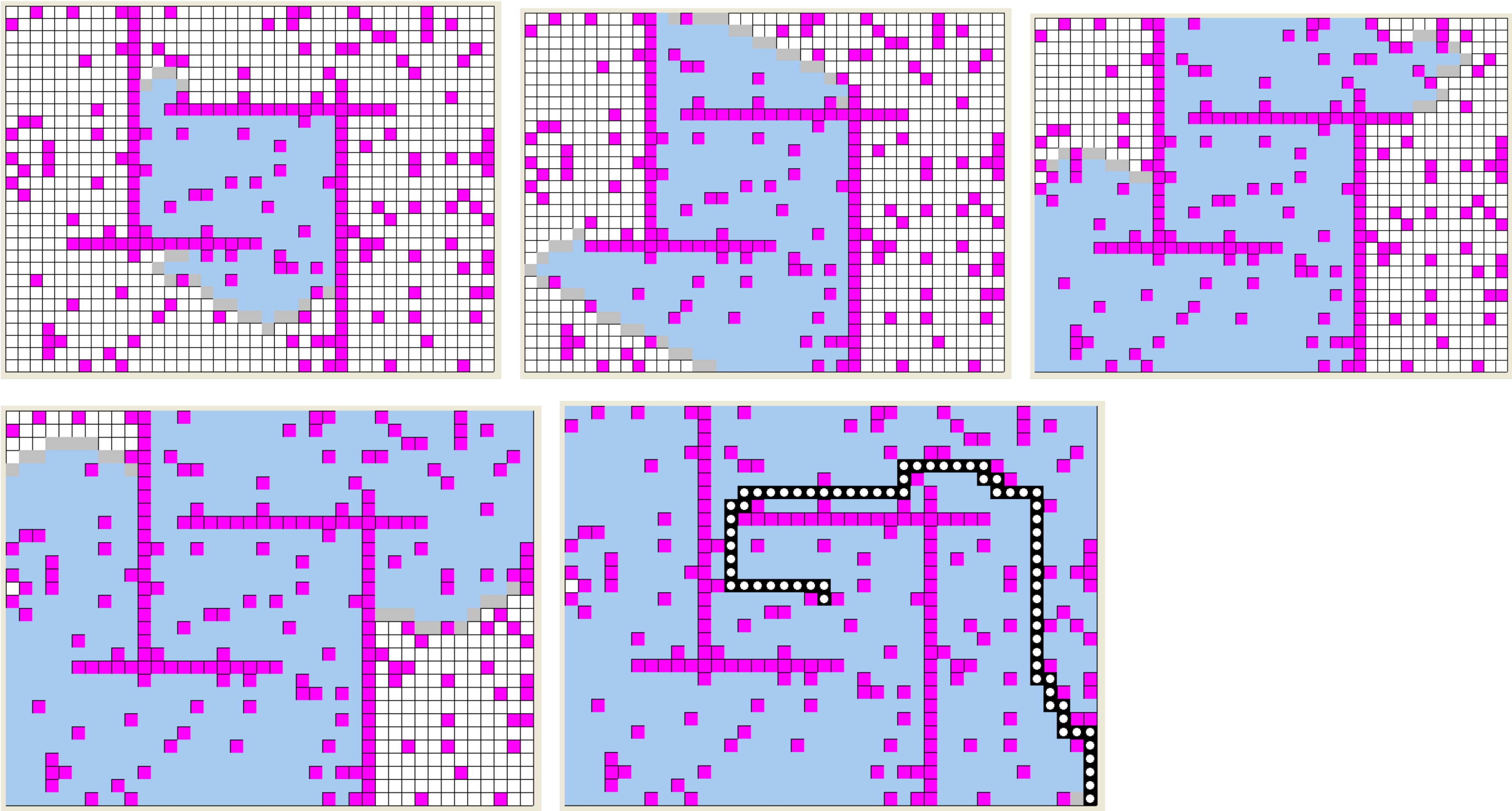
Dijkstrův algoritmus (uniform cost search)

Edsger Dijkstra 1930-2002

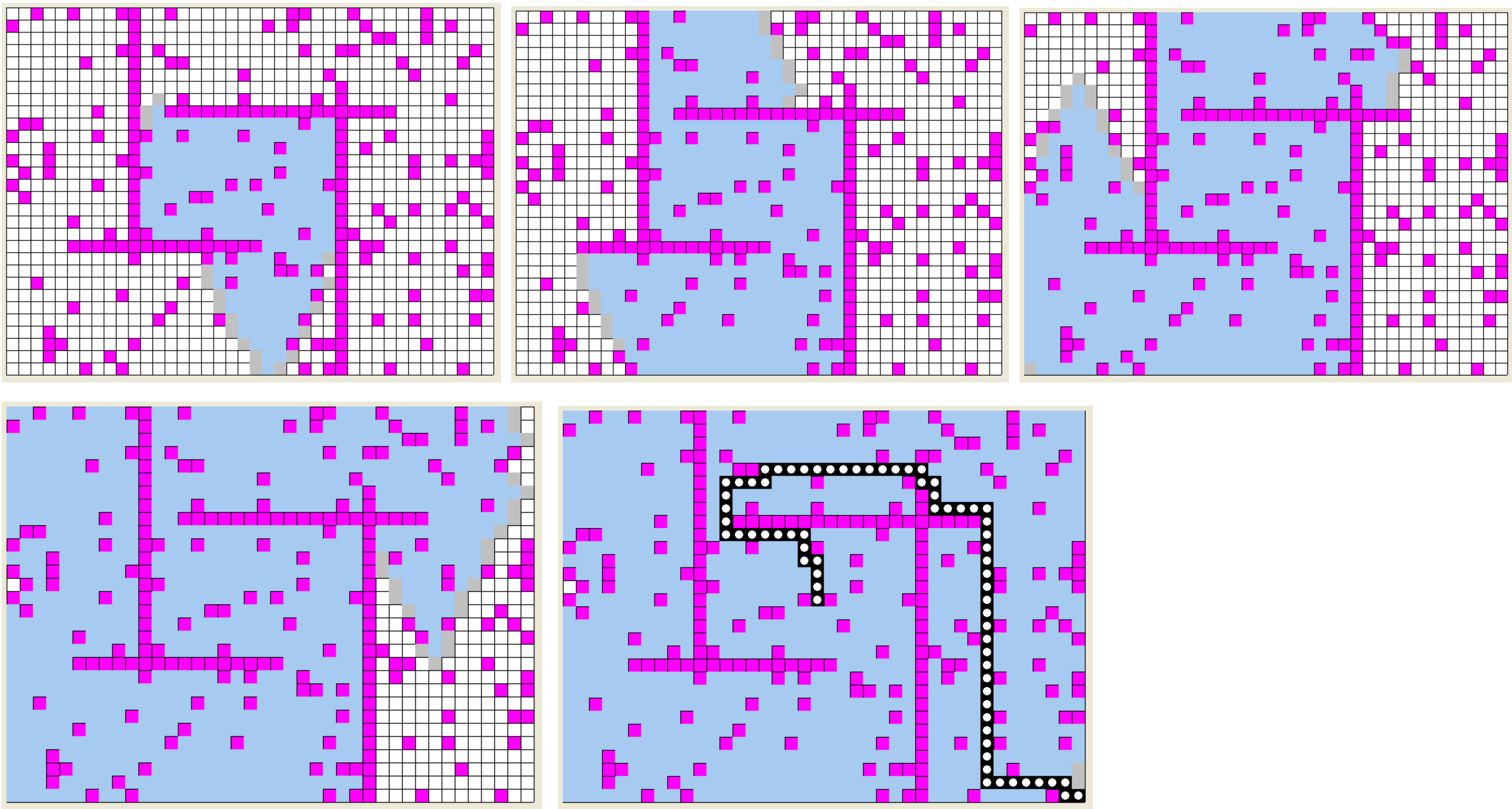
- Zahrnuje cenu od startu k uzlu (cost to come)
- Cena přechodu *g(stav,akce)* může být definována různě (vzdálenost měst v příkladu, různá náročnost vodorovného / svislého pohybu, …)
- Zařazení expandovaného uzlu (řádek 11 algoritmu prohledávání) do seznamu O podle ceny (třídění seznamu O)
- Duplicity: pokud se ocitneme v uzlu, kde jsme již byli, je rozhodující cena, za kterou jsme do uzlu dorazili (ve stejném stavu se můžeme ocitnout různými cestami).
- Pokud je cena přechodu jednotková, algoritmus se neliší od prohledávání do šířky
- Obecnější koncept **Uspořádané prohledávání** (Best-first search)

Příklad: hledání cesty, rozdílná cena pro vertikální a horizontální pohyb

$$g(s,a)=\begin{cases}1 \quad pro \quad a=\{vlevo, vpravo\} \\ 2 \quad pro \quad a=\{nahoru, dolu\}\end{cases}$$



$$g(s,a)=\begin{cases}2 & pro \quad a=\{vlevo, vpravo\}\\1 & pro \quad a=\{nahoru, dolu\}\end{cases}$$

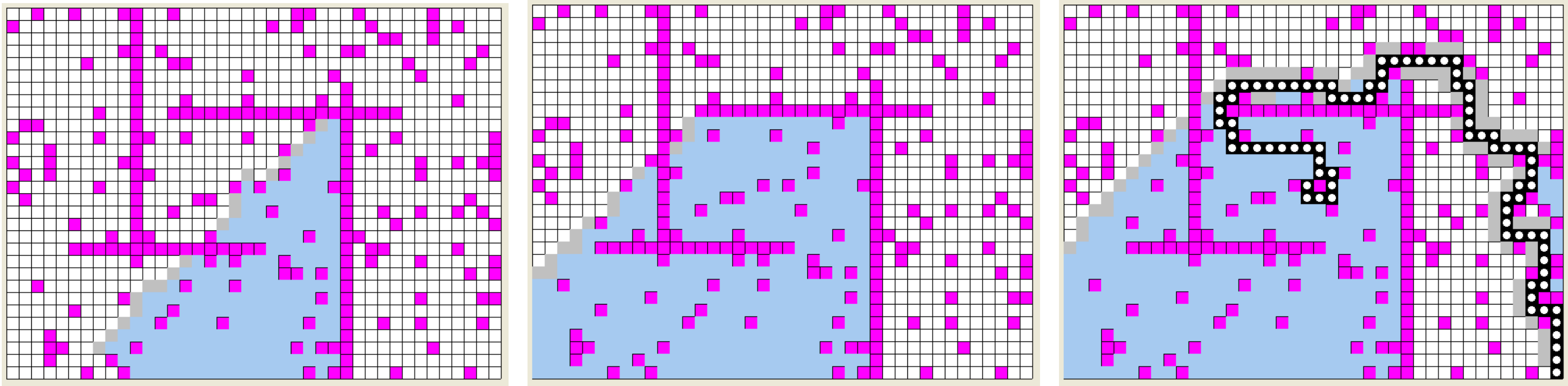


Heuristika

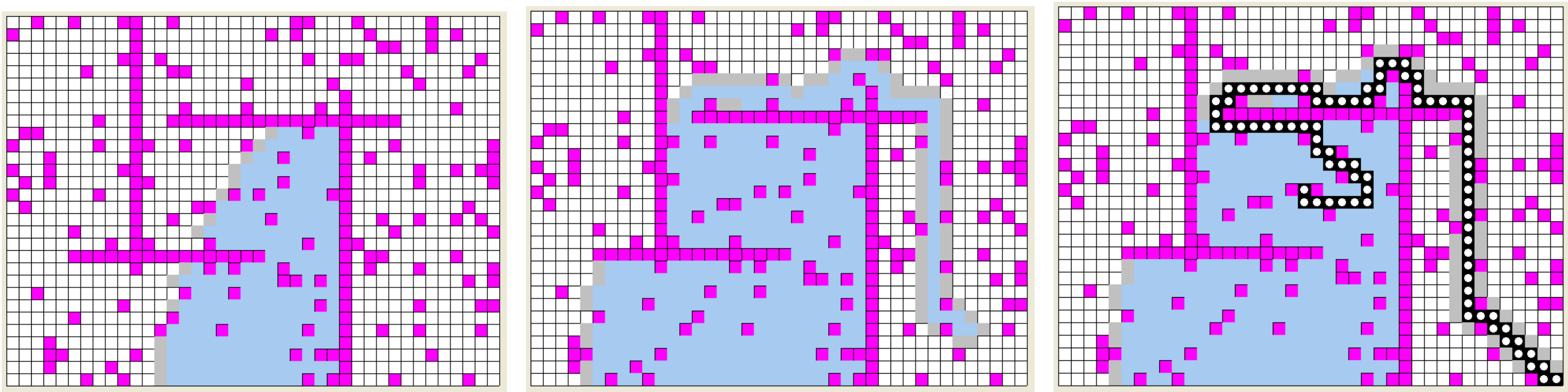
- Prozatím jsme jako hodnotící funkci (tedy to, podle čeho budeme vybírat uzel ze seznamu O k expanzi) brali pouze cenu od počátku k danému uzlu
- Obecný tvar hodnotící funkce $f(s,a)=g(s,a)+h(s,a)$
- $f(s,a)$ - cena z počátku do cíle přes stav *s*
- $g(s,a)$ - cena z počátku do stavu *s*
- $h(s,a)$ - cena z daného stavu do cíle – **nevím! Odhaduji – heuristická funkce**
- Algoritmus uspořádaného prohledávání (best-first search) – algoritmus A
- Algoritmus se stává přípustným (vždy najde optimální cestu, pokud existuje), pokud je funkce $h^*(s,a)$ přípustná, pak se mu říká A* (ejstár)

Příklad – hledání cesty: varianta uspořádaného prohledávání

- ignorujeme cenu z počátku do stavu, bereme jen funkci *h()*
- heuristika (vzdálenost od cíle): různé metriky
- manhattanská $h_1(s) = |x - x_G| + |y - y_G|$
- eukleidovská $h_2(s) = \sqrt{(x - x_G)^2 + (y - y_G)^2}$



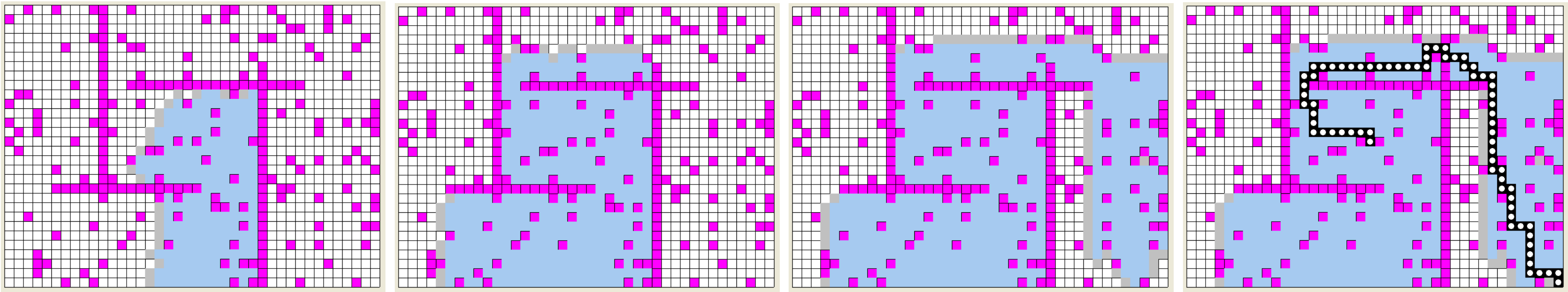
Manhattanská metrika



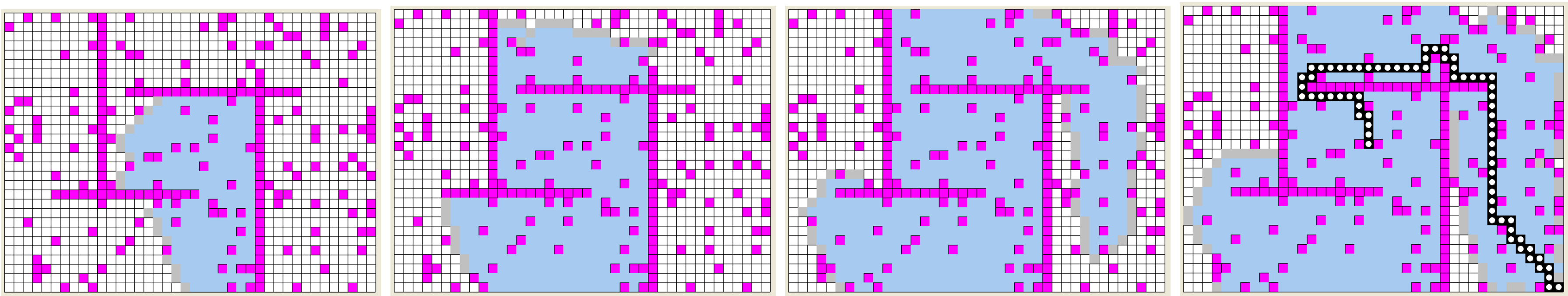
Eukleidovská metrika

Příklad – hledání cesty: A*

- Bereme v úvahu obě ceny, se stejnou vahou



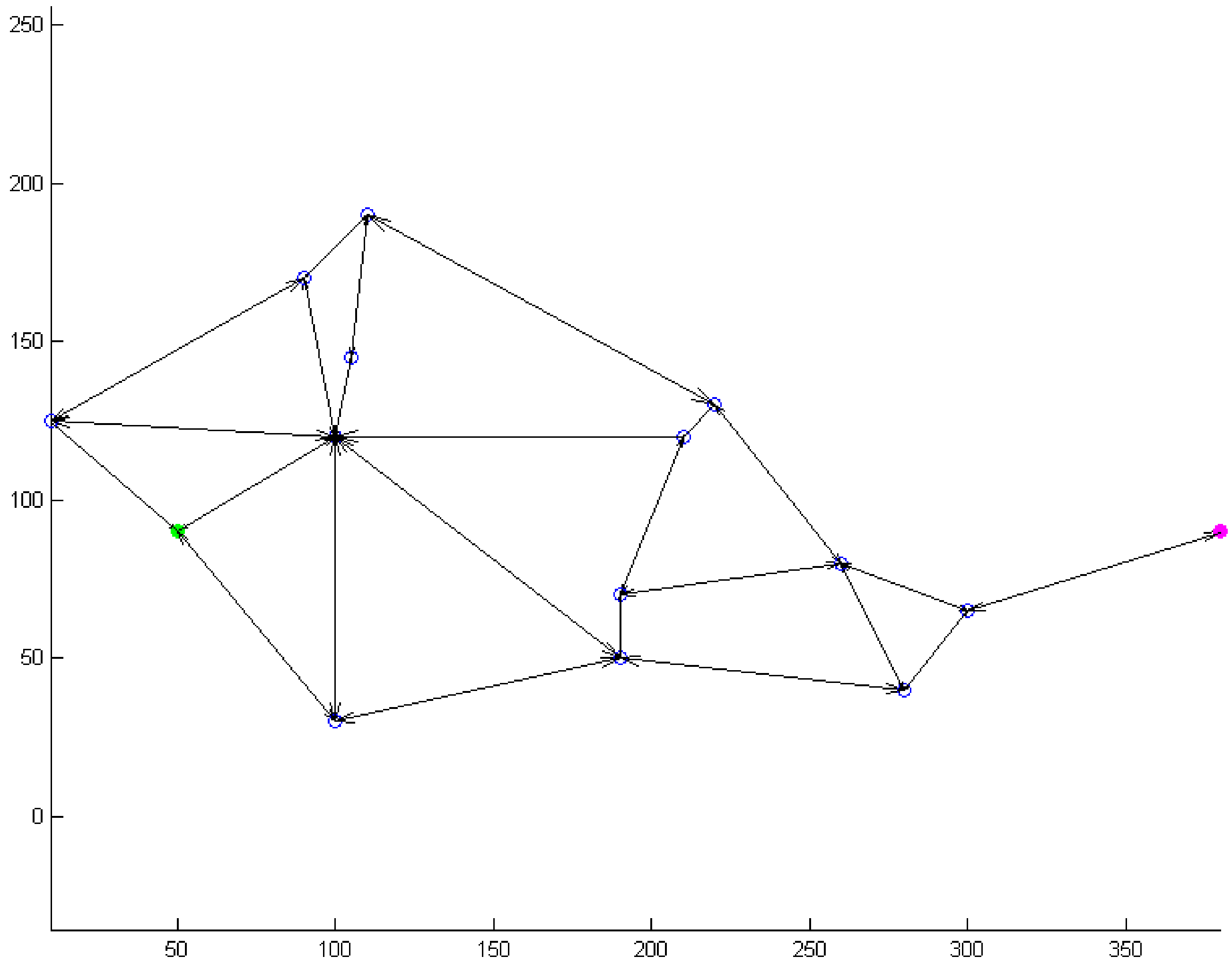
Manhattanská metrika



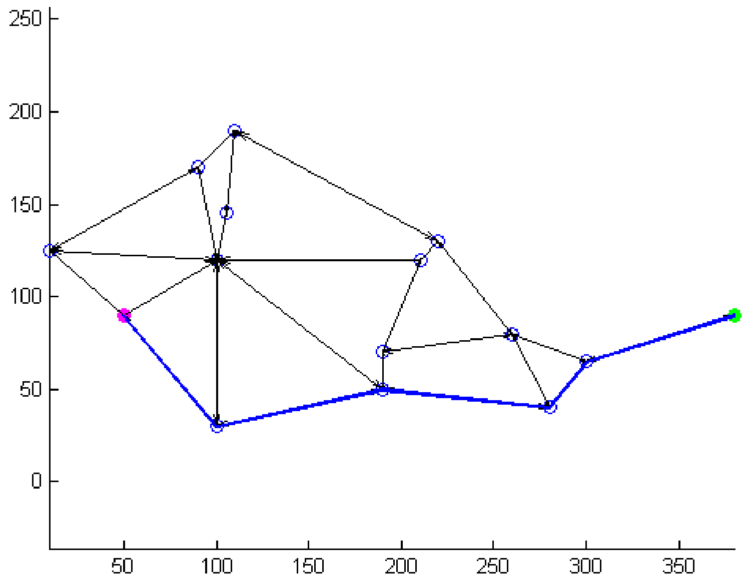
Eukleidovská metrika

Příklad: hledání cesty z města do města

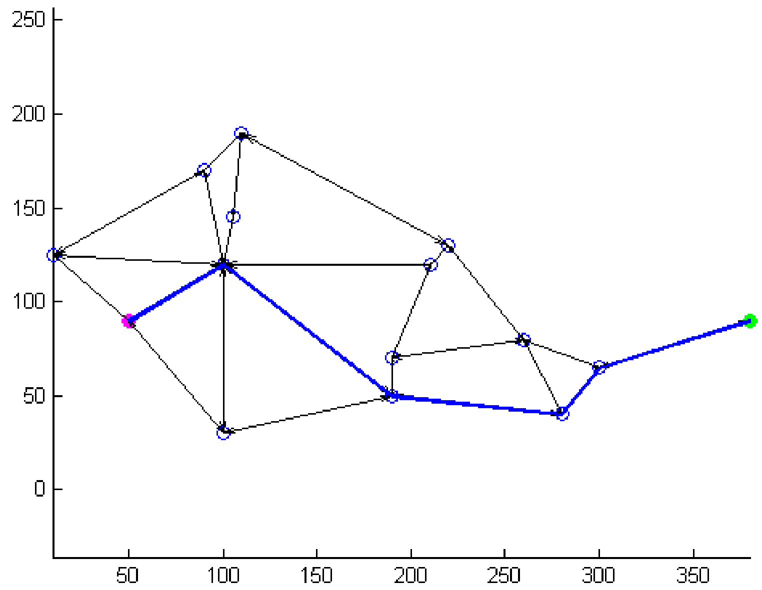
- Mapa: seznam měst, každé má svůj identifikátor, polohu a seznam cest, které z něj vedou
- Mapa = stavový prostor
- Uzel: obsahuje navíc informace o ceně od startu do uzlu (costToCome) a od uzlu k cíli (costToGoal)
- Podle ceny se třídí seznam O
    1. Pouze cena costToCome: uspořádané prohledávání
    2. Pouze cena costToGo
    3. Hledání cesty z města do města pomocí A*: celková cena zahrnuje obě ceny
- Mapa České republiky, pozor, Pardubice - Praha jednosměrka


250
200
150
100
50
0
50
100
150
200
250
300
350

Plzeň - Ostrava



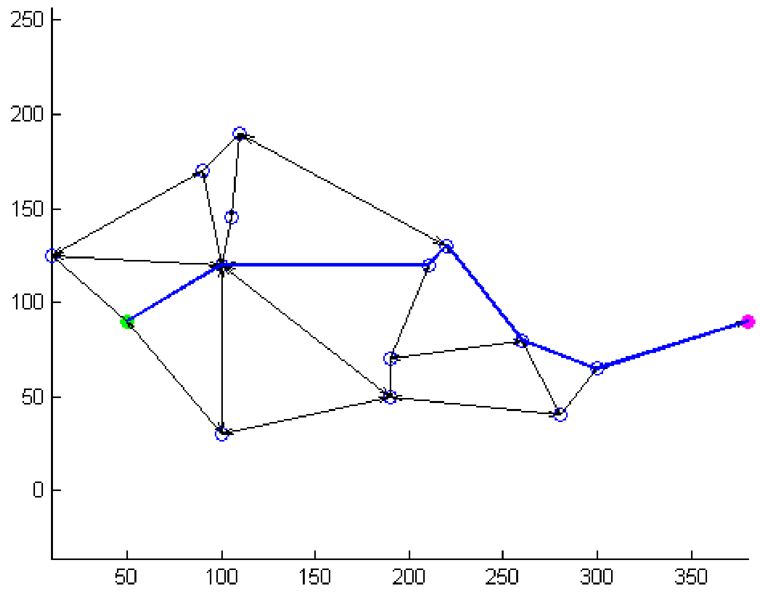
Uspořádané prohledávání, A*



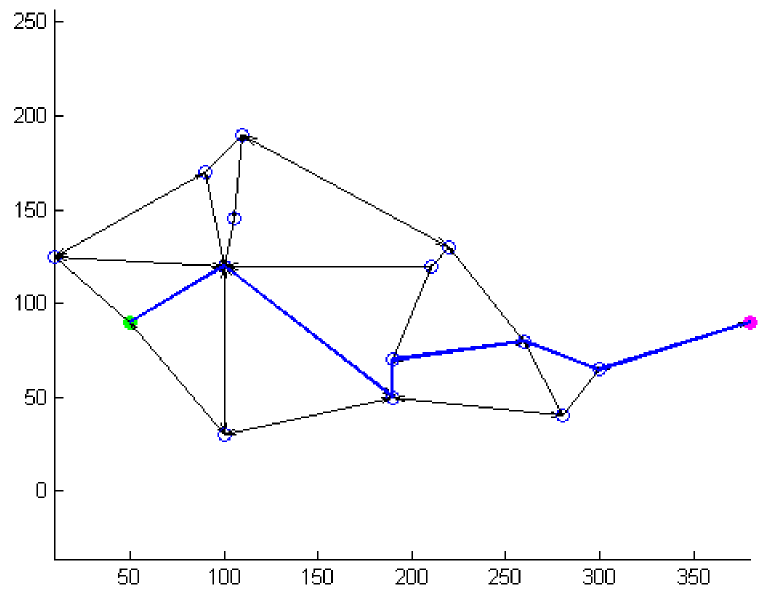
CostToGo

- Uspořádané prohledávání: 5610 uzlů, délka cesty 376km
- A*: 30 uzlů, délka cesty stejná
- costToGo only: 18 uzlů, délka cesty 378km

Opačný směr (Ostrava – Plzeň)



Uspořádané prohledávání, A*



CostToGo

- A*: strom 44 uzlů, délka cesty 373km
- Uspořádané prohledávání: strom 2364 uzlů, délka cesty 373km
- costToGo only: 21 uzlů, 389km

Shrnutí

- Pokud je $h(s,a)=0$ bereme v úvahu jen cenu od počátku do stavu – Dijkstra (uniform-cost search)
- Pokud je $h(s,a)=0\ AND\ g(s,a)=1$ dostaneme slepé prohledávání do šířky
- Pokud je $h(s,a)=0\ AND\ g(s,a)=0$ dostaneme algoritmus náhodného prohledávání
- Další vylepšení (rozdíl není velký)
  - Algoritmus paprskovitého prohledávání (beam search)
  - Algoritmus větví a mezí (branch-and-bound search)
  - IDA* (iterative deepening A*)
  - Metaznalosti (závislé na konkrétní úloze)
  - …

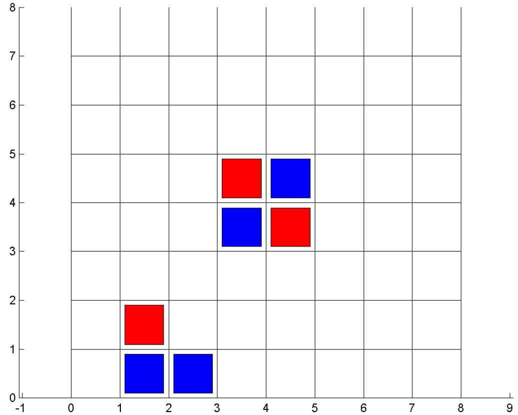
Lehký náhled do teorie her

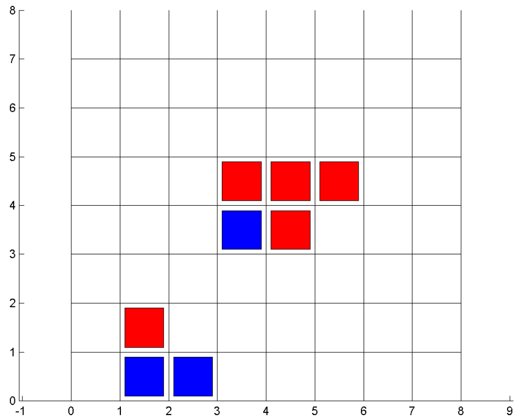
Hra s nulovým součtem (zero-sum game)

- Antagonistické konflikty: co jeden získá, to druhý ztratí
- Příklad: hrajeme kámen-nůžky-papír o korunu. Jeden ji získá, druhý o ni přijde
- Příklad: dělení dortu. Pokud má někdo větší kousek, tak druhý má přesně o tolik méně
- Zero-sum bias: V reálném světě lidé propadají přesvědčení, že např. ekonomika je zero-sum game (aby mohl být někdo hodně bohatý, musí být ostatní chudí)
- Jak s tím souvisí prohledávání stavového prostoru?
- Problém dvou hráčů – oba chtějí vyhrát

Příklad: reverzi

- Čtvercová hrací plocha 8x8
- Střídavé pokládání kamenů na volné pozice
- Pokud je po položení kamene hráčem X uzavřena souvislá sekvence soupeřových kamenů z obou stran, přechází do vlastnictví hráče X.
- Pravidlo pro řady, sloupce
- Pravidlo pro diagonály
- Hra končí zaplněním celé hrací plochy
- Vítězí hráč s větším množstvím kamenů





Implementace

1. Stav: **board** - matice 8x8, hodnoty 0 (=volno), 1 (=hráč 1), 2 (=hráč 2)
2. Tah: změna hodnot v matici pro daný kámen + kontrola na změnu kamenů v uzavřené oblasti
3. Funkce Player: zástupce algoritmu, vrací aktuální tah
4. Turnaj: volání odkazu na funkci

```
function [myX,myY] = Player(board,myColor)
% very simple player
% input: board 8x8, 0 - free space, 1,2 - players
%        myColor: 1 or 2
% output: myX, myY coordinates of the move

%% this simple player finds free spaces and randomly
chooses one of them
idx = 1;
for j=1:8
    for k=1:8
        if board(j,k) == 0
            fs(idx,:) = [j, k];
            idx = idx + 1;
        end
    end
end
% fs now contains free spaces coordinates
[L c] = size(fs);
idx = randi(L,1);
myX = fs(idx,1);
myY = fs(idx,2);
```

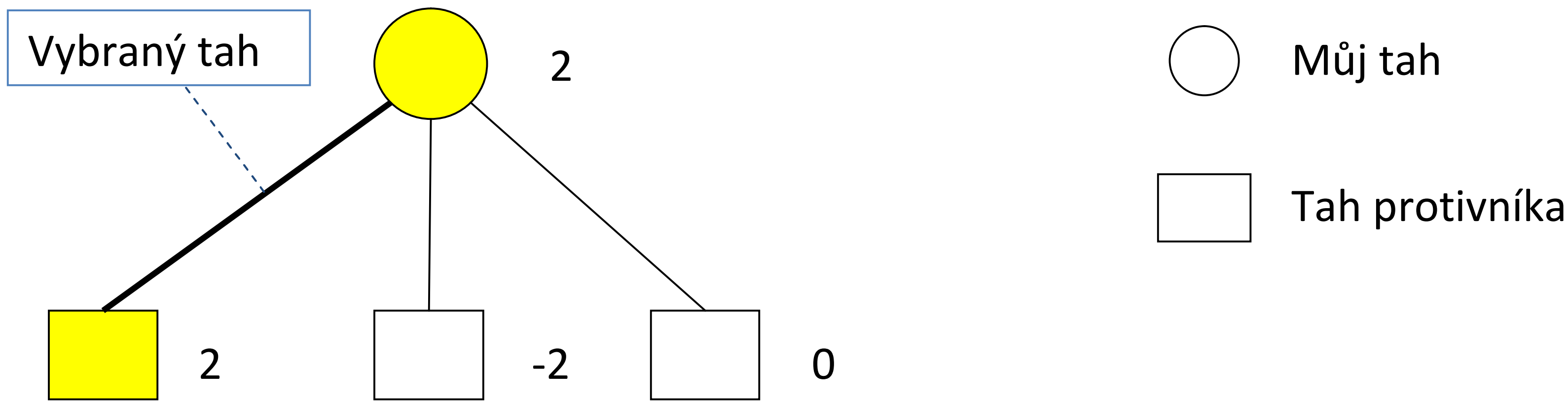
```
function [newboard] =
MakeMove(board,x,y,myColor,enemyColor)
% performs the single move, updates board
% check if possible and make all the updates for all
directions
newboard = board;
% check for illegal move
% perform the move and update newboard
```

```
function [gameResult, p1cnt, p2cnt] =
RunSingleGame(player1f,player2f,view)
% run the game function!
% board has [X,Y] notation !
% player 1 - BLUE
% player 2 - RED
% view == 0 means no display, full figure otherwise
% player1f and player2f are function handles

player1 = 1;
player2 = 2;

%% init board
board = zeros(8);
board(5,5) = 1;   board(4,4) = 1;
board(5,4) = 2;   board(4,5) = 2;

%% run the game in the loop
[eg, gameResult, p1cnt, p2cnt] = BoardCheck2(board);
while eg == 0
    [idx,idy] = player1f(board,player1);
    board = MakeMove(board,idx,idy,player1,player2);
    [idx,idy] = player2f(board,player2);
    board = MakeMove(board,idx,idy,player2,player1);
    [eg, gameResult, p1cnt, p2cnt] = BoardCheck2(board);
end
```

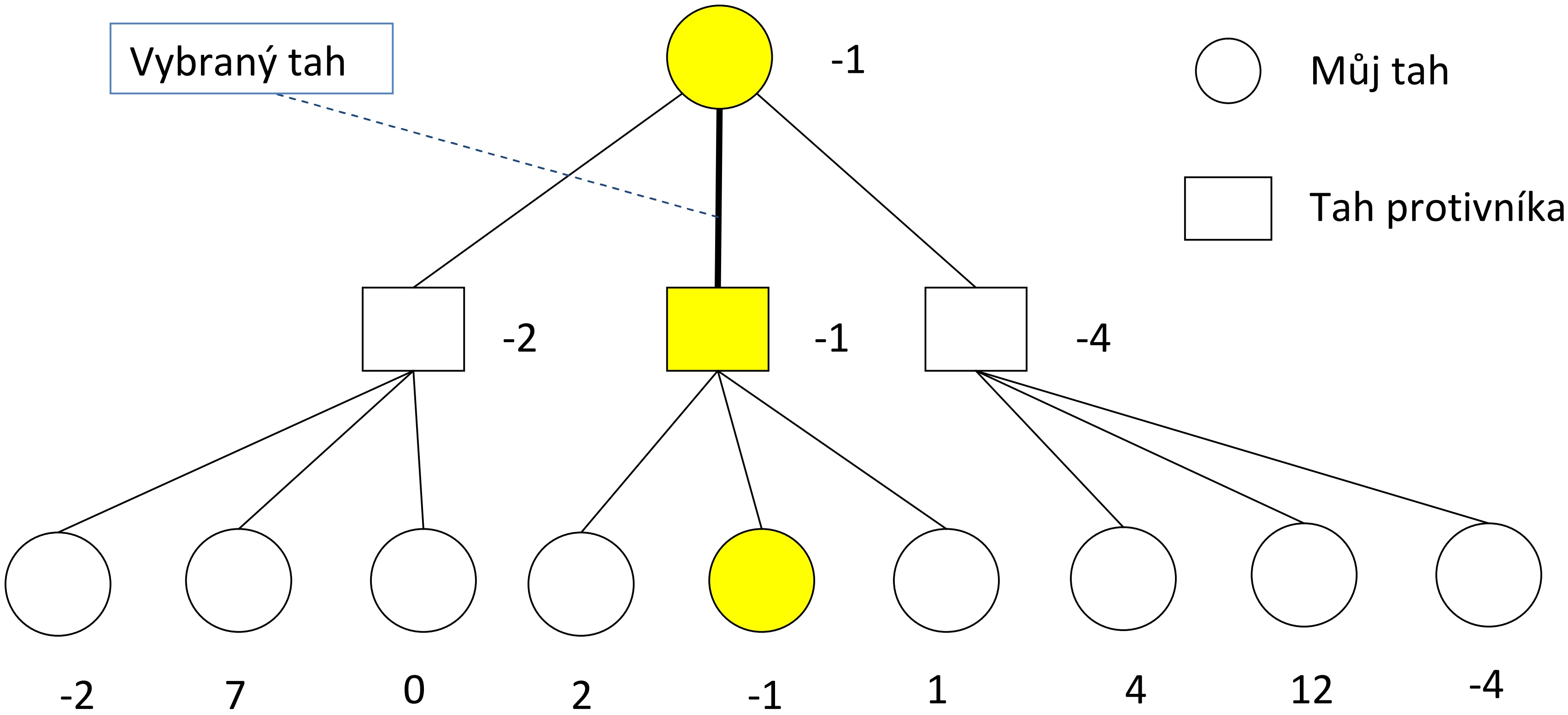
```
%% reversi rournament script

%% get a list of players
players{1} = @DBplayer;
players{2} = @PlayerRuja;
players{3} = @PlayerJez3;
players{4} = @PlayerVrana;

view = 1;   % to show game results

playerCount = length(players);

%% run games everyone with everyone
for j=1:playerCount
  for k= 1:playerCount
    if j==k
        % nemuzu hrat sam se sebou
    else
        % jednotlive hry
        p1 = players{j};
        p2 = players{k};
        fprintf('%s VS %s ',func2str(p1),func2str(p2));
        [Res, p1cnt, p2cnt] = RunSingleGame(p1,p2,view);
        % prideleni score ….
    end
  end
end
```

Algoritmus minimax

- Základní myšlenka: hráč A chce maximalizovat svoji šanci (ohodnocení stavu), hráč B chce totéž, ale pro sebe. Takže oba chtějí maximalizovat svoje šance, a minimalizovat šance protivníka.
- Pořád je to stavový prostor a jeho prohledávání
- Každý stav musí být ohodnotitelný hodnotící funkcí (např. o kolik kamenů mám víc)
- Lichý tah – můj tah, Sudý tah – tah protivníka
- Můj tah – maximalizace ohodnocení
- Protivníkův tah – minimalizace ohodnocení protivníka (předpokládám, že protivník potáhne nejlépe, jak je to možné)
- **Minimalizace maximálních ztrát**
- Jakmile protivník táhne, tak buď celý strom začnu počítat znovu, nebo využiji předpočítaných kousků z minula



Hledání do první úrovně (můj tah)



Hledání do druhé úrovně (tah protivníka)

Vyberu takový tah, kdy protivník může docílit nejlépe hodnoty -1 (v ostatních případech by mohl táhnout chytřeji a docílit -2 (levý podstrom) nebo dokonce -1 (pravý podstrom)

Další možná vylepšení

Alfa-beta prořezávání (alpha-beta pruning)

- Stanovím dvě meze
  - Alfa – dolní mez kdy jsem na tahu (maximalizuji)
  - Beta – horní mez kdy je na tahu protivník (minimalizuji)
- Jakmile jsem překročil mez, již tuto část stromu nebudu dále rozvíjet (ušetřím kus stromu)

Animace na Wikipedii: https://upload.wikimedia.org/wikipedia/en/7/79/Minmaxab.gif