Základy umělé inteligence

Lukáš Nevrkla

20. března 2022

Obsah

Kapitola 1

Úvod

- Expertní systémy
  - AI systémy pro konkrétní aplikaci (aplikace znalostí expertů)
  - PROLOG - PROgramming in LOGic
- Dělení AI
  - Klasická - počítačové vidění
  - Distribuovaná - roje
  - Soft computing - genetické algoritmy, fuzzy, pravděpodobnost, ...
- CSP (constraint satisfaction problems)
  - Nezáleží na cestě k řešení, jen na výsledku
  - Snaha určit hodnoty proměněných tak aby platily dané podmínky

Kapitola 2

Stavový prostor

- Dvojice (S, O)
  - S ... stavy
  - O ... operátory (činnosti, ...)

Úlohy

- Úloha 2 džbánů
  - Stav = (množství tekutiny většího džbánu, menšího)
    - Defaultně 4l a 3l
  - Operace = naplnění džbánu / přelití do jiného džbánu / vylití džbánu
  - Cíl = získat určité množství tekutiny ve džbánu
- Loydovy osmičky / patnáctky
  - Matice 3x3, pole 1, ... 8, 1 pole volné
  - Kameny lze posouvat na volné místo
    - Nebo posouvám prázdným polem
  - Cílem je seřadit kameny do daného pořadí
  - Stav = (č. kamenu na pozici 1, 2, ...)
    - Prázdné pole: _
  - Operace = posun volního pole do 4 směrů
- Úloha 8 dam (CSP)
  - Matice 8x8
  - Každá dáma má sloupec
  - Stav = pozice dam (číslo řádku pro každý sloupec)
  - Operátory = umístění nové dámy (řádek 1-8)
  - Konečný stav - dámy se navzájem neohrožují (nemají jinou dámu horizontálně / vertikálně / diagonálně)
- Kameny
  - 2 bílé, 2 černé, 1 prázdný

  - Strav (WWBBE)
  - Operace: přesun E doleva / doprava, přeskok E o 2 pole doleva / doprava
  - Pozor, nemůže přeskočit z jednoho konce na druhý
- Balanční váhy
  - Nalezení mince s jinou váhou než ostatní, jen pomocí porovnávaní 2 množin mincí
  - Stav = (neporovnané, stejné, množina s max. 1 lehčí, množina s max. 1 těžší)
- Hanojské věže
- Optimální cesta
- Obchodní cestující
  - Navštívení všech měst právě 1 + nejkratší cesta
  - Problém permutace
- Barvení map
  - Obarvení polí, tak aby sousední barvy byly jiné

Prohledávání stavového prostoru

- Hodnocení
  - Úplnost - vždy najde řešení, pokud existuje (nezacyklí se)
  - Optimálnost - vždy najde **nejlepší** řešení, pokud existuje
  - Časová a prostorová složitost
    * b = faktor větvení, d = hloubka stavového prostoru

Slepé

- Zkoušení všech možností

BFS (breadth first search)

- Vlastnosti
  - Úplné a optimální (úplná jen s CLOSE)
  - Časová složitost: $O(b^d)$
  - Prostorová složitost: $O(b^d)$
- **Fronta** uzlů k expanzi (OPEN)
- Skonči pokud
  - Aktuálně **vybraný** stav je cílový
  - OPEN je prázdná
- Vyjmi 1. uzel a expanduj jej (nové stavy do OPEN)
  - Pokud se stav v OPEN už vyskytuje => nepřidávej

  - Na zkoušce: vypište stav po expanzi 3. uzlu = aplikace 3. operátoru (ne expandování 3. uzlu)
- Lze přidat seznam CLOSE pro již expandované uzly
  - Zpracované uzly umísťuji do CLOSE
  - Nově expandované stavy umístím do OPEN jen pokud nejsou v CLOSE, ani OPEN
  - => zabrání zacyklení => úplná
- Rekonstrukce postupu / cesty
  - Uchovávat všechny předchůdce pro každý stav
  - S CLOSE stačí jen 1 předchůdce pro každý stav

```
-----------------------------------------------
OPEN  = [first]
CLOSE = [ ]

while (OPEN not empty)
    curr = OPEN.pop

    if (curr is END-STATE)
        return curr

    new = curr.expandAllNotIn(OPEN, CLOSE)
    OPEN.add(new)
    CLOSE.add(curr)

return None
-----------------------------------------------
```

DFS (depth first search)

- Úplná a neoptimální (úplná jen s CLOSE)
- BFS, jen místo fronty => zásobník
  - => do hloubky
  - => neoptimální
- Časová složitost: $O(b^d)$
- Prostorová složitost: $O(d)$
  - Lineární prostorová složitost - stačí si pamatovat aktuální větev

DLS (depth limited search)

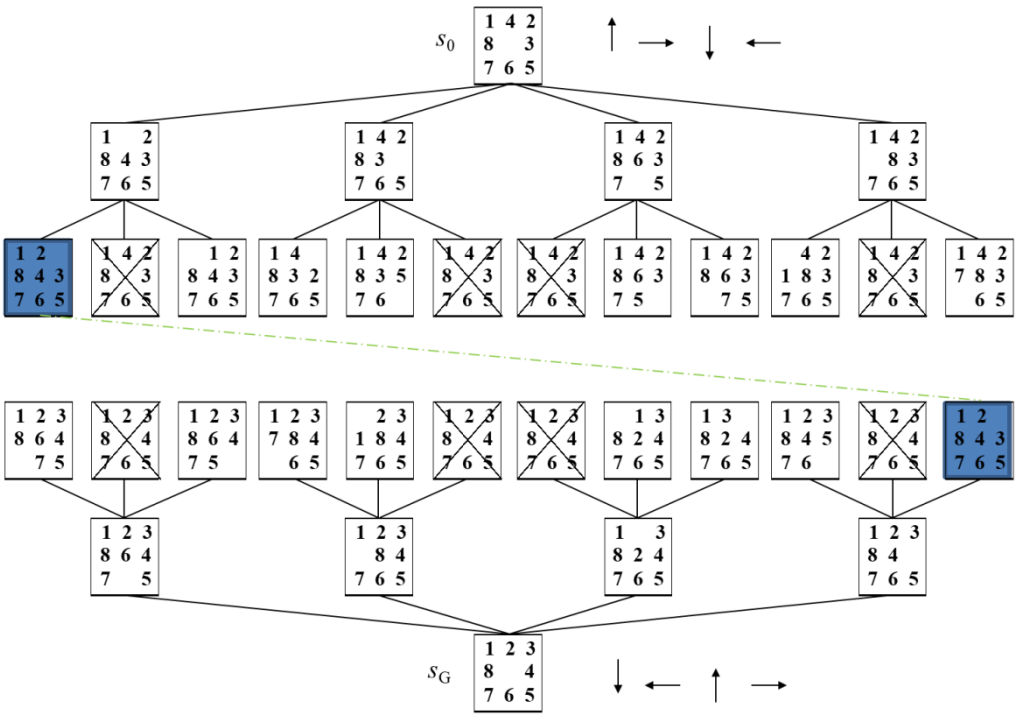
- Pokud lze pro danou aplikaci odhadnout maximální hloubku řešení
- Jak DFS + při dosažení limitu ukončí větev
  - Navíc ukládaná hloubka zanoření

IDS (iterative deepening search - postupné zanořování)

- Úplné a optimální
- Opakované volání DLS + zvyšování limitu hloubky (od 1)
- Konec, pokud poslední DLS expandovalo vše
- Větší časová složitost => po inkrementování hloubky se již provedené stupně počítají znovu
- Optimálnost jak BFS, ale lineární prostorová složitost jak DFS (ale delší doba výpočtu)
  - BFS: 3.5 let, 0.1 PB
  - IDS: 3.9 let, 12 kB

BS (bidirectional search - obousměrné BFS)

- Úplné a optimální
- Postupuje od cílového a počátečního stavu (2 fronty)
  - Pouze pokud existují reverzibilní operátory
- Konec jakmile je stejný uzel v obou frontách (oba směry se setkaly)
- Časová složitost: $O(b^{d/2})$
- Prostorová složitost: $O(b^{d/2})$
- Loydovy osmičky:



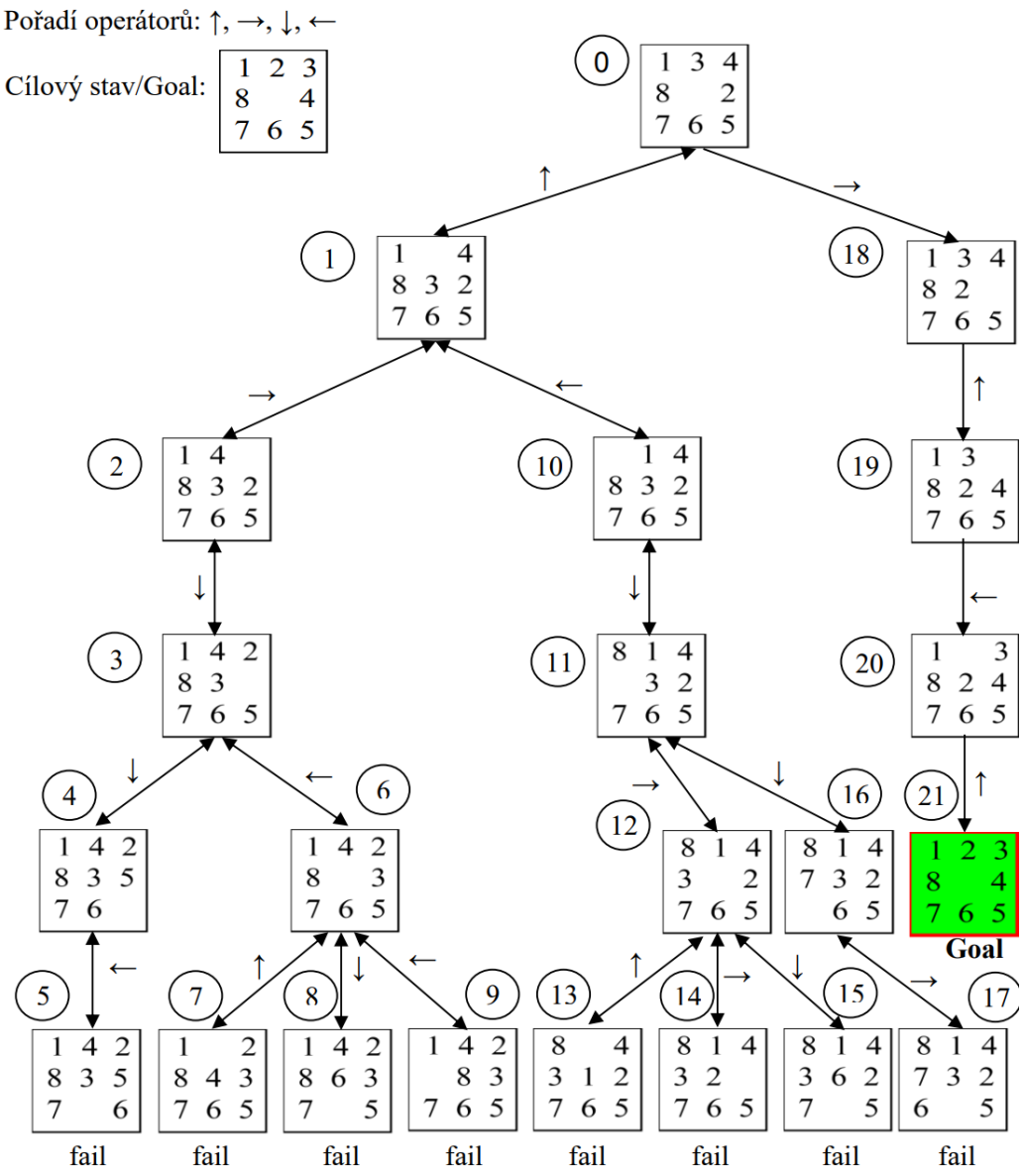
UCS (uniform cost search - metoda stejných cen)

- Úplné a optimální
- Přiřazení kladných cen jednotlivým operacím
  - Stav: $(s_i, g(s_i))$
  - Počáteční stav: $g(s_0) = 0$
- Expandují se operace (cesty) s nejmenší cenou
  - Seznam / prioritní vyhledávací fronta (OPEN)
  - Směr expandování není dán napevno jak BFS, DFS
  - Cena = délka prošlé cesty od počátku
- Ukončení, až se cílový uzel vybere z OPEN (bude mít nejnižší cenu)
- Dvě optimalizace (použít obě na zkoušce):
  - Pokud se v OPEN uzel již vyskytuje, nechám ten s nejmenší cenou
  - Seznam prozkoumaných stavů (CLOSE)
    - Pokud je nově vy-expandovaný stav v CLOSE, nevyložím ho do OPEN
    - Stav v CLOSE má vždy menší cenu - byl prozkoumán dřív
- Podobné jak hledání nejkratší cesty - Dijkstra

Backtracking (zpětné navrácení)

- Jak DFS (nebo i DLS), ale generuje se jen 1 následovník (ne všechny možné operace naráz)
  - U BFS nedává smysl
- Po ukončení větve se navrátí k předchozímu a u něj zkusí následující (nevyzkoušenou) operaci
- Oproti DFS má menší prostorovou složitost
- Opět, pokud je vy-expandovaný uzel už v OPEN, nepřidávat jej

```
-------------------------------------------------
OPEN  = [first]
CLOSE = [ ]

while (OPEN not empty)
    if (OPEN.canExpandNext)
        curr = OPEN.expandNext

        if (curr is END-STATE)
            return curr
        if (curr not in OPEN)
            OPEN.add(curr)
    else
        OPEN.pop

return None
-------------------------------------------------
```



Informované

- Cena stavu = cena cesty od počátku + odhad (heuristika) k cíly
- Záleží na kvalitě heuristiky
  - Nejlepší možná heuristika => složitost = počet kroků nejlepšího řešení
  - Čím blíže je heuristika skutečnosti, tím rychleji řešení najdeme

Best First Search

- Prohledávání od nejlepšího
- Jako UCS (slepá metoda)
  - Seznam OPEN
  - Expandování uzlu s nejmenší cenou dokud není cílový / OPEN není prázdné

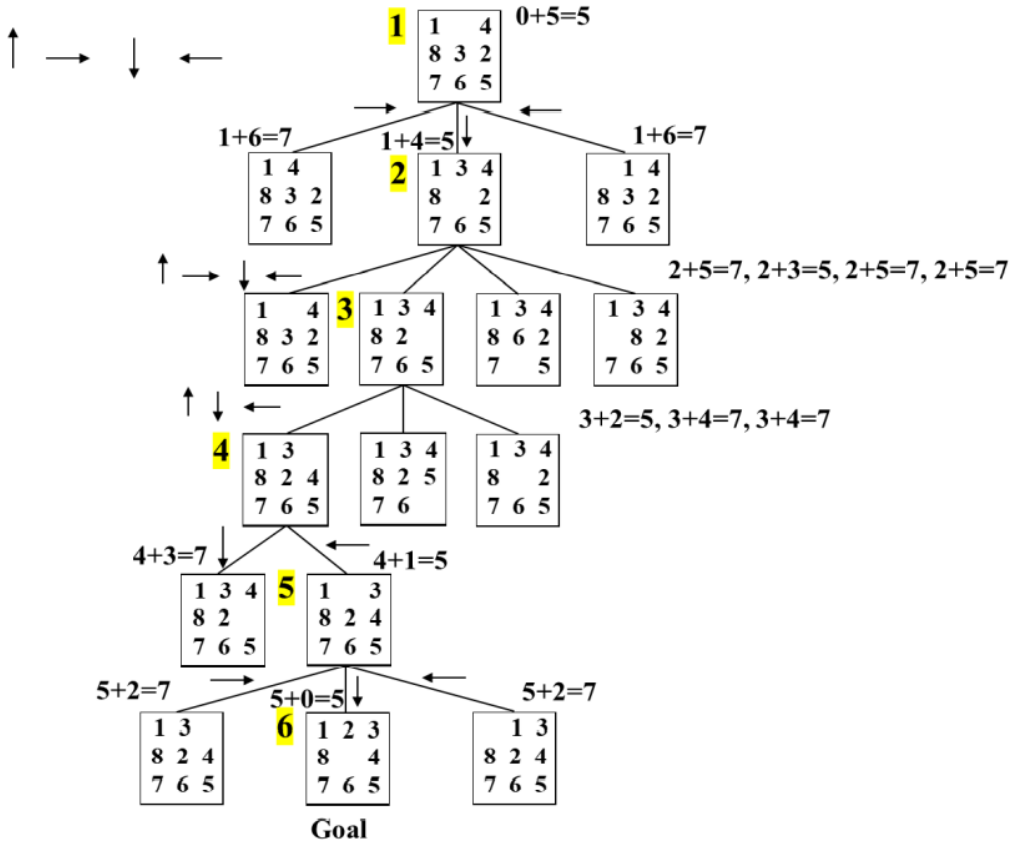
  - + CLOSE
- UCS
  - Cena stavu = cena cesty od počátku

Greedy Search (hltavé hledání)

- Cena stavu = odhad (heuristika) k cíly
- Úplná, neoptimální
  - Chytí se první potenciálně dobré cesty

A* Search

- Úplná, optimální
- Nejlepší metoda prohledávání stavového prostoru
- Cena stavu = cena cesty od počátku + odhad (heuristika) k cíly
- Požadavky na heuristiku pro zajištění optimálnosti
  - Podhodnocená - vždy menší, nebo rovna než skutečná cena
    - Pokud heuristika(a) < heuristika(b), pak musí platiti: real(a) < real(b)
  - Monotonní - s dalším krokem se zvyšuje
    - + cena nemůže být záporná
  - Faktor větvení je konečný
- Loydovy osmičky
  - Heuristika = součet vzdálenosti jednotlivých kamenů od cílových pozic (horizontální + vertikální vzdálenost)



Lokální prohledávání

- Zvolení počátku + prohledání pouze okolí
- Pro problémy, kde je složité projít všechny stavy
- Cena stavu je často jen hodnota heuristika k cíly
- Stačí znát aktuální a expandované uzly

Hill climbing

- Neúplné a neoptimální
- Expandování nejlepšího stavu jak Greedy search, ale zastaví se v kopci (na lokálních extrémech)

```
-------------------------------------------------
curr = FIRST

while (1)
    next = best(curr.expand)
    if (next.price > curr.price)    // curr is better
        return curr
    else
        curr = next
-------------------------------------------------
```

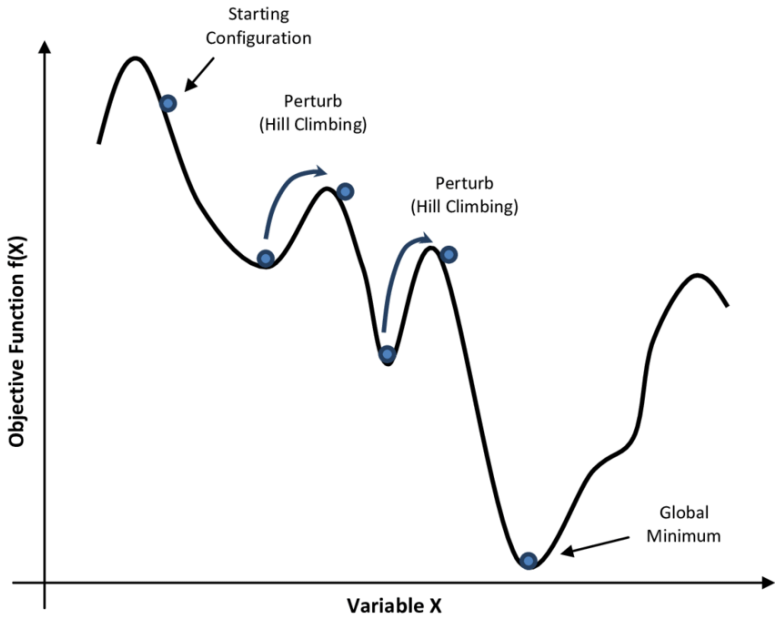
Simulated annealing (simulované žíhání)

- Teplotní funkce klesající s dalšími kroky výpočtu
- Jak hill climbing, ale
  - V kopci skončí jen při vychladnutí
  - Šance na překročení kopce (snižuje se s chladnutím)

$$e^{\frac{f(current)-f(next)}{Temperature}}$$

  - Následovník se vybere náhodně
    * Pokud je lepší, vždy se do nej přesune
    * Pokud není, je určitá šance, že se do něj přesune

```
-------------------------------------------------
curr = FIRST
temp = INIT-TEMP

while (1)
    next = random(curr.expand)
    delta = curr.price - next.price

    if (delta < 0)    // curr is better
        if (temp = 0)
            return curr
        else
            with probability = e ^ (delta / temp)
                curr = next
    else
        curr = next

    temp.update()
-------------------------------------------------
```



Algoritmy inspirované přírodou

Genetické algoritmy

- Princip
  - Vytvoření populace a její ohodnocení
  - Zvolení rodičů
  - Křížení (rekombinace)
  - Mutace
  - Update populace + opakování
- Jedinec je popsaný chromozomem
  - Složen z genů = bitů
  - Generováno např. náhodně
- Fitness funkce
  - Kvalita / hodnocení chromozomu (čím vyšší, tím kvalitnější)
- Ukončení
  - Daný počet iterací
  - Minimální změny populace
- Volení rodičů
  - Elitismus
    - Výběr nejlepších
    - Může se zaseknout na lokálních extrémech
  - Náhodné rozdělení na n-tice + z každé vybrání toho nejlepšího
  - Náhodně
  - Aby populace nerostla - vybírat polovinu počtu => 2x potomků
- Křížení
  - Zvolení náhodně velké části chromozomu z 1. rodiče a zbytku z 2.
  - + 2. potomek => opačné spojení (zbytku)
- Mutace
  - Invertování náhodného bitu
  - Malá pravděpodobnost (1/N)
- Tvorba nové populace
  - Generační model - všichni rodiče nahrazeni potomky
  - Inkrementační model - jen 1 jedinec nahrazen potomky
  - Překrytí generací (kombinace) - část rodičů nahrazena potomky

Ant cycle optimalization

- Na dobrých cestách vypouští feromon, který přiláká ostatní
- Řešení obchodního cestujícího
  - Inicializace
    * Intenzita feromonů na všech cestách nastavena na malé kladné číslo
    * Přírůstek feromonů na všech cestách = 0
    * Tabulka mravenec-město => jakými městy prošel
    * Náhodné rozmístění mravenců do různých měst
  - Pravděpodobnost přesunu do daného města

$$p_{ij} = \frac{(\text{feromony-na-cestě})^{\alpha} \cdot (\text{viditělnost})^{\beta}}{\sum[(\text{feromony-na-cestě})^{\alpha} \cdot (\text{viditělnost})^{\beta}]}$$

    * Viditelnost = 1 / váha cesty
    * $p_{ij} = 0$ ... pro již navštívená města
    * Větší u kratších cest a u cest s více feromony
  - Přesun všech mravenců dokud neprojdou všechna města (právě 1)
  - Celková vzdálenost mravence = projíte cesty + cesta zpět do počátku
  - Výpočet změny feromonů na všech cestách (kudy prošel)

$$\Delta\tau_{ij} = \sum_{\text{mravenci}} \frac{\text{celkové feromony mravence}}{\text{celková vzdálenost mravence}}$$

  - Úprava feromonů na všech cestách
    * Započítání vypařování feromonů (k)

$$\tau_{ij} = (1-k)\tau_{ij} + \Delta\tau_{ij}$$

  - Opakování do splnění některé podmínky

Particle swarm optimalization (PSO)

- Každá částice má
  - Pozici v prostoru
  - Vektor rychlosti
  - Poslední nejlepší řešení
- Počáteční polohy a rychlosti se určí náhodně
- Poloha se updatuje podle rychlosti
- Vektor rychlosti se updatuje podle
  - Setrvačnosti (váhuje původního vektor rychlosti)
  - Vektoru k nejlepšímu řešení (best - curr):
    * Nejlepší řešení dané částice - kognitivní
    * Nejlepší řešení celé skupiny - sociální

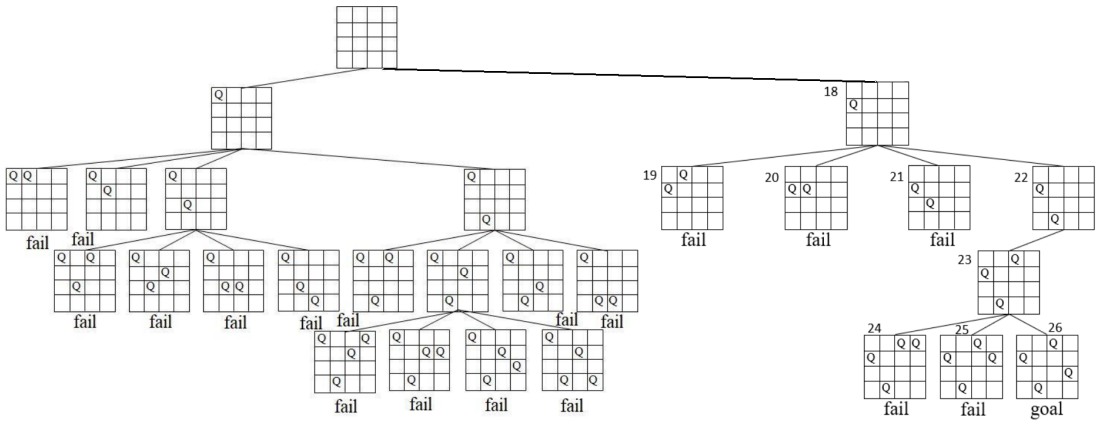
        * + váhování kognitivním a sociálním koeficientem
        * + váhování náhodnými veličinami (z normálního rozdělení)
        * Kvalita řešení podle heuristické funkce
- Po určité době se částice shromáždí kolem optimálního řešení
    - Je pokryta velká část prostoru (šance, že některá částice najde dobré řešení)
- Dobré, pokud je směr dán spojitě (ne diskrétní množina možných kroků)

Řešení CSP (constraint satisfaction problems)

- Množiny množných hodnot pro jednotlivé proměnné
- Volba proměnných tvoří stavový prostor
- V každém kroku se ověřují všechny podmínky
- Všechny úplné metody jsou optimální => všechna řešení jsou stejně kvalitní

Backtracking (zpětné navrácení)

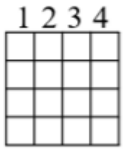
- Prohledání pomocí backtrackingu (viz. výše)
- Při nesplnění některé podmínky => vynořit se



Forward checking (dopředná kontrola)

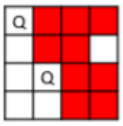
- Jak backtracking, ale vždy při zvolení nové proměnné se odstraní konfliktní hodnoty ostatních proměnných
    - Dopředné nalezení konfliktů => méně zanoření
- Postup
    - Množiny možných hodnot pro jednotlivé proměnné
    - Zvolíme hodnotu další proměnné (na začátku 1.)
    - Smazat všechny konfliktní hodnoty
        * Smazat u podmínek, kde je 1 volná proměnná
        * U více volných se musí určit další proměnná

- Pokud je některé množina prázdná
  - * Obnovit předchozí stav množin (=> potřeba stack)
  - * Odstranit aktuální hodnotu současné proměnné (je v konfliktu se zbývajícími)
- Konec při
  - * Nalezení vyhovujícího stavu
  - * Některá množina je prázdná v kořeni

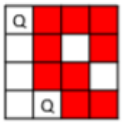


$S_1 = \{1,2,3,4\}, S_2 = \{1,2,3,4\}, S_3 = \{1,2,3,4\}, S_4 = \{1,2,3,4\}$

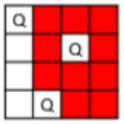
$x_1 = 1$
$S_1 = \{2,3,4\}, S_2 = \{3,4\}, S_3 = \{2,4\}, S_4 = \{2,3\}$



$x_1 = 1, x_2 = 3$
$S_1 = \{2,3,4\}, S_2 = \{4\}, S_3 = \{\}, S_4 = \{2\}$ fail !
$S_3 = \{2,4\}, S_4 = \{2,3\}$



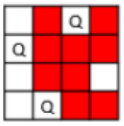
$x_1 = 1, x_2 = 4$
$S_1 = \{2,3,4\}, S_2 = \{\}, S_3 = \{2\}, S_4 = \{3\}$



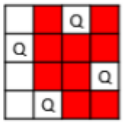
$x_1 = 1, x_2 = 4, x_3 = 2$
$S_1 = \{2,3,4\}, S_2 = \{\}, S_3 = \{\}, S_4 = \{\}$ fail !
$S_2 = \{1,2,3,4\}, S_3 = \{1,2,3,4\}, S_4 = \{1,2,3,4\}$

$x_1 = 2,$
$S_1 = \{3,4\}, S_2 = \{4\}, S_3 = \{1,3\}, S_4 = \{1,3,4\}$

$x_1 = 2, x_2 = 4,$
$S_1 = \{3,4\}, S_2 = \{\}, S_3 = \{1\}, S_4 = \{1,3\}$



$x_1 = 2, x_2 = 4, x_3 = 1,$
$S_1 = \{3,4\}, S_2 = \{\}, S_3 = \{\}, S_4 = \{3\}$

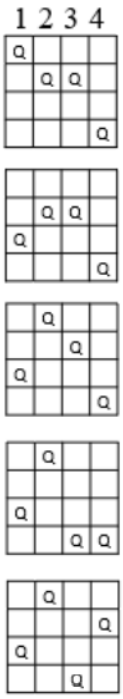


$x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3,$
$S_1 = \{3,4\}, S_2 = \{\}, S_3 = \{\}, S_4 = \{\}$
$X = (x_1, x_2, x_3, x_4) = (2, 4, 1, 3)$ Goal

- Heuristiky výběru proměnné pro přiřazení
  - Most-constrained-variable - nejvíce omezená proměnná (nejméně možných hodnot)
  - Most-constraining-variable - nejvíce omezuje ostatní
- Heuristiky výběru hodnoty
  - Leas-constraining-value - hodnota omezuje nejméně ostatní proměnné

Min-conflict

- Proměnné inicializuji náhodnými hodnotami
- Cyklicky procházím přes všechny proměnné (1, ... N, 1 ... N, ...)
  - U aktuální proměnné určím pro každou její hodnotu počty konfliktů
  - Pokud existuje menší počet konfliktů než u aktuálně zvolené hodnoty (nebo alespoň stejný, ale pro jinou hodnotu), zvolím tuto hodnotu a pokračuji na další proměnnou
- Konec jakmile nešlo změnit ani jednu proměnnou (proběhl 1 cyklus bez změny)
- Minimální paměťová složitost (jen aktuální stav)
- Není dokázaná její konvergence => není jisté, zda je úplná a optimální



Počet ohrožení dámy *i* na jednotlivých řádcích sloupce *i*:

$Q_1 = \{2,2,1,2\}$

$Q_2 = \{1,3,2,2\}$

$Q_3 = \{2,1,2,1\}$

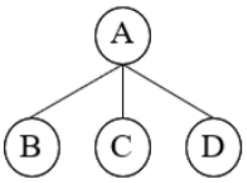
$Q_4 = \{1,0,3,1\}$

$Q_1 = \{1,3,0,1\}$, $Q_2 = \{0,2,2,3\}$, $Q_3 = \{3,2,2,0\}$, $Q_4 = \{1,0,3,1\}$

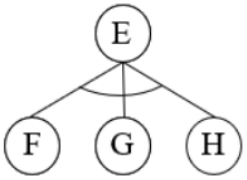
$\Rightarrow$ Goal

Rozklad na podproblémy (AND-OR grafy)

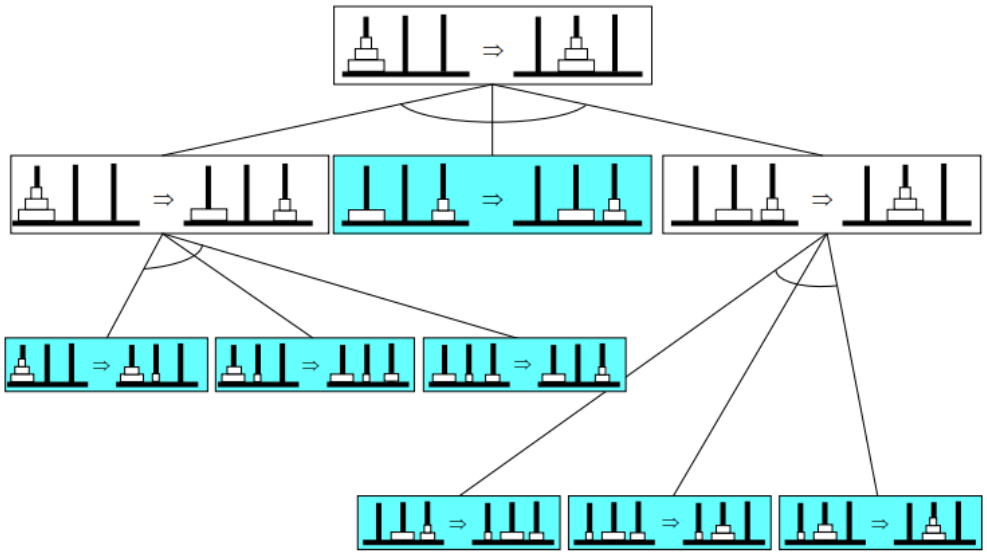
- OR graf - aspoň 1 podproblém splnitelný => celý problém je splnitelný



- AND graf - každý podproblém splnitelný => celý problém je splnitelný



- Obyčejné procházení stavového prostoru je OR graf
- Hanojské věže



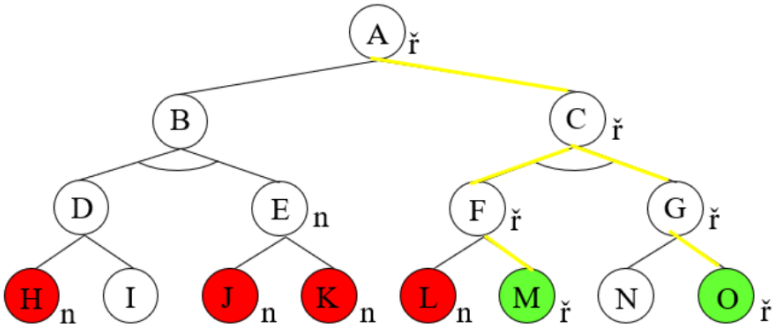
Slepé AO algoritmy

- AO = AND-OR
- Jak BFS / DFS, ale
  - Navíc strom uchovávající, zda jsou dané uzly řešitelné / neřešitelné / neprozkoumané
  - Expandování na podproblémy
  - V listech (řešitelný / neřešitelný problém) => propagace nové informace zpět k rodičům
    * + vyhodnocování AND, OR
    * Pokud je kořen řešitelný / neřešitelný => konec

– Ostatní uzly => uložit do OPEN

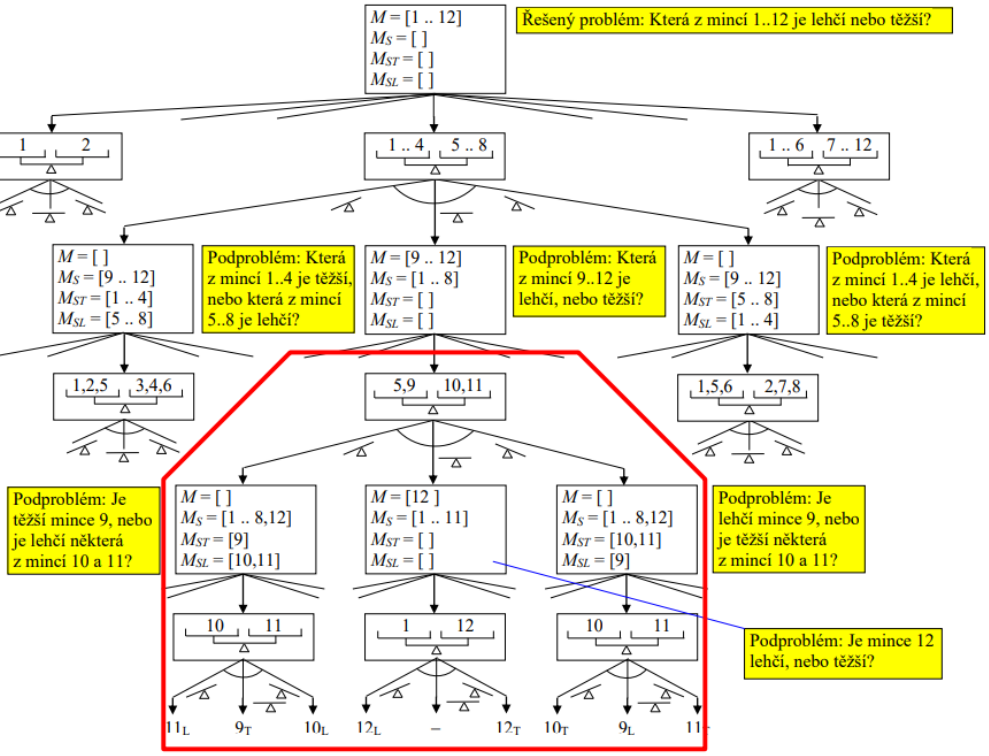
```
-------------------------------------------------
// state = SOLVED / FAILED / UNKNOWN
// solvable = (state != UNKNOWN)

OPEN = [first]
TREE = [first]

while (OPEN not empty)
    curr = OPEN.pop

    new = curr.expandAll()
    solvable, others = new.splitToSolvable()

    TREE.add(solvable)
    TREE.propagateChanges()

    if (TREE.root is solvable)
        return TREE.root.state

    OPEN.add(others)
    OPEN.removeSolved() // solved = parent is solvable

return None
-------------------------------------------------
```

- Lze vylepšit backtrackingem (DFS)
- Příklad s BFS



Krok	OPEN	Graph
0	[A]	[(A,?)]
1	[B,C]	[(A,?),(B,?),(C,?)]
2	[C,D,E]	[(A,?),(B,?),(C,?),(D,?),(E,?)]
3	[D,E,F,G]	[(A,?),(B,?),(C,?),(D,?),(E,?),(F,?),(G,?)]
4	[E,F,G,I]	[(A,?),(B,?),(C,?),(D,?),(E,?),(F,?),(G,?),(H,n),(I,?)]
5	[F,G,I]	[(A,?),(B,?),(C,?),(D,?),(E,n),(F,?),(G,?),(H,n),(I,?),(J,n),(K,n)]
6	[G,I]	[(A,?),(B,?),(C,?),(D,?),(E,n),(F,ř),(G,?),(H,n),(I,?),(J,n),(K,n),(L,n),(M,ř)]
7	[I]	[(A,ř),(B,?),(C,ř),(D,?),(E,n),(F,ř),(G,ř),(H,n),(I,?),(J,n),(K,n),(L,n),(M,ř),(N,?),(O,ř)]

- Úloha balančních vah

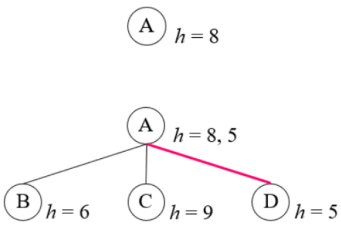


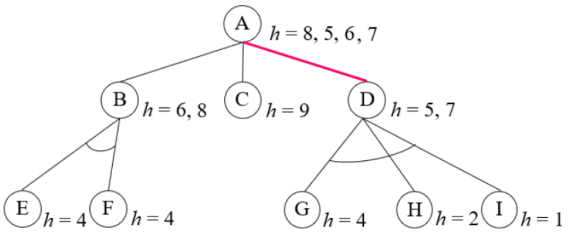
Informované AO* algoritmy

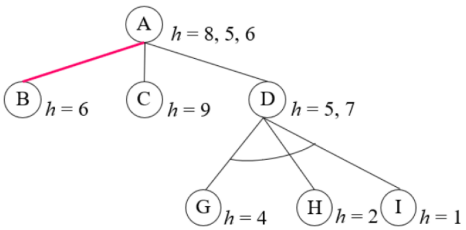
- Heuristická funkce pro odhad kvality daného stavu

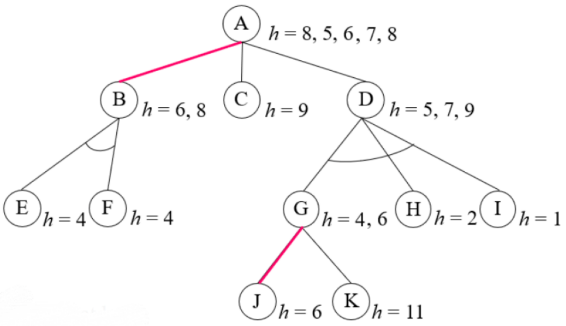
- Strom aktuálních odhadů (heuristik) pro jednotlivé stavy
- Expanze uzlu s nejmenší cenou
  - Určení heuristik nových stavů (0 / max. pro řešitelné)
- Rekurzívní update odhadů rodičů
  - OR uzel => min dílčích heuristik
  - AND uzel => součet heuristik
- Pokračuji dokud není kořen řešitelný (cena = 0) / neřešitelný (nastavení max. ceny řešení)
- Oproti A* se může vracet (nalezne lepší cestu)
  - Kvůli zpětné propagaci ve stromu

```
-------------------------------------------------
// heuristic == 0   ... SOLVED
// heuristic == MAX ... FAILED
// solvable = (heuristic == 0 | heuristic == MAX)

TREE = [first]

while (TREE.root not solvable)
    curr = TREE.moveRecursiveToBest()

    if (curr.canExpand)
        nexts = TREE.expand(curr)

        for (n in nexts)
            n.heuristic = getHeuristic(n)

        curr.add(nexts)
        TREE.propagateHeuristic()

    else
        curr.heuristic = MAX

return (TREE.root.heuristic == 0) ? SOLVED : FAILED
-------------------------------------------------
```


A
h = 8
A
h = 8, 5
B
h = 6
C
h = 9
D
h = 5


A
h = 8, 5, 6, 7
B
h = 6, 8
C
h = 9
D
h = 5, 7
E
h = 4
F
h = 4
G
h = 4
H
h = 2
I
h = 1


A
h = 8, 5, 6
B
h = 6
C
h = 9
D
h = 5, 7
G
h = 4
H
h = 2
I
h = 1

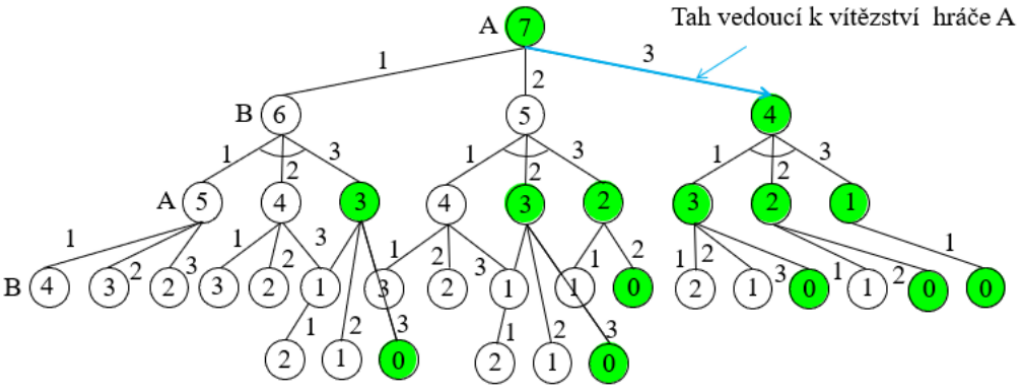

A
h = 8, 5, 6, 7, 8
B
h = 6, 8
C
h = 9
D
h = 5, 7, 9
E
h = 4
F
h = 4
G
h = 4, 6
H
h = 2
I
h = 1
J
h = 6
K
h = 11

Metody hraní her

- Určení aktuálního tahu, tak aby daný hráč vyhrál / zvýšil pravděpodobnost
- Hry, kde se 2 hráči střídají, lze znázornit AND / OR grafem
  - Svůj tah ... alespoň 1 výherní řešení => OR
  - Protivníkův tah ... všechny řešení musí být výherní => AND (umím vyhrát na každou jeho akci)

Jednoduché hry - bruteforce

- Lze prozkoumat všechny možné tahy => AO*
  - Sestavování AND, OR grafu (střídaní hráčů - AND <=> OR)
  - V listech (prohra / výhra) propagování vzhůru
- Hra NIM
  - Odebírání 1/2/3 zápalek z hromady 7 zápalek
  - Kdo odebere poslední, prohrál



Složité hry

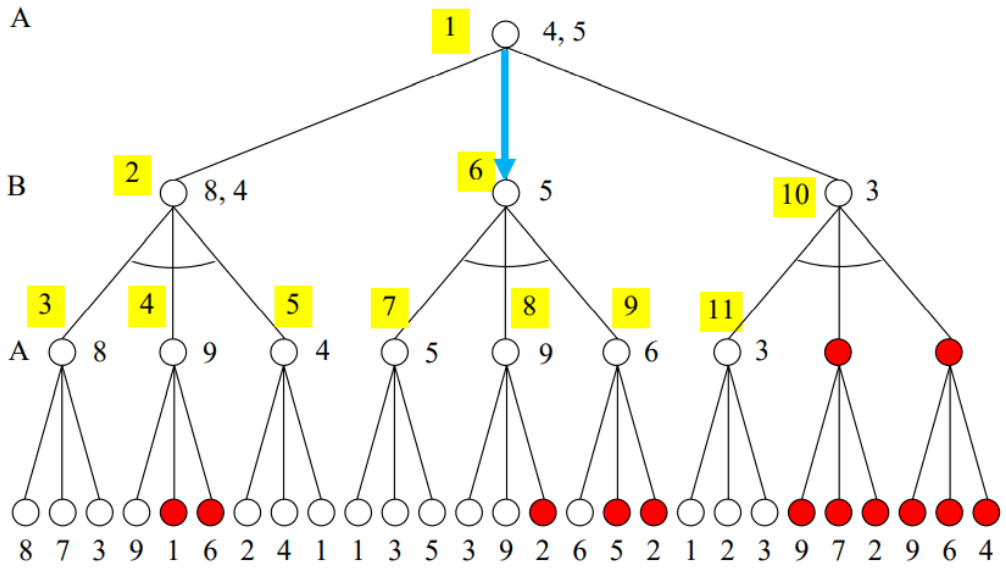
- Nelze brute force
- Šachy, ...
- Ohodnocení heuristikou - užitková funkce (utility func)
  - Kladné hodnoty - větší šance na výhru
  - Záporné hodnoty - větší šance na výhru protivníka
  - Výhra => inf
  - Prohra => -inf

MinMax

- Zanoření do určité hloubky (ořezání stavového prostoru)
  - Aplikace operátorů
  - Odhad heuristiky v listech

        * Listy se mohou objevit i ve vyšších vrstvách - prohra / výhra dříve

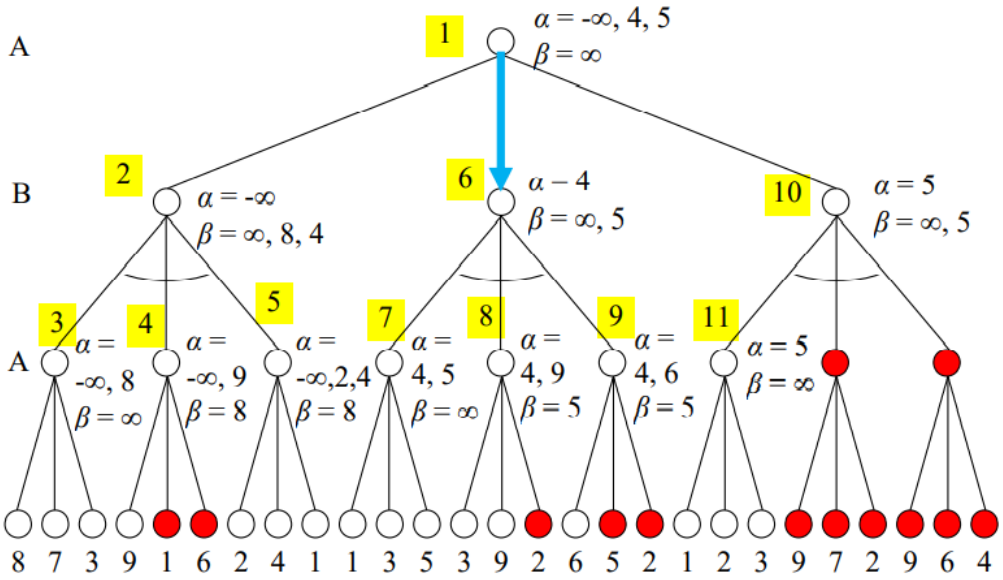
- Navrácení hodnoty zpět nahoru
  - Listy ... navrácení jejich odhadnuté heuristiky
  - Svůj uzel (OR) ... MAX z aktuální nejlepší a navrácené hodnoty
  - Protivníkův uzel (AND) ... MIN z aktuální nejlepší a navrácené hodnoty
  - Takto pro všechny potomky
  - Na zkoušce znázorňovat postupnou historii vyšších uzlů
- Neefektivní
  - Řeší i problémy, které není nutné řešit
  - Řeší všechny další pod-uzly, i když se již aktuální, kvůli min / max nemůže uložit do rodiče
- Výsledek je jen 1. krok = strategie (ne celý postup)
  - Je to jen heuristika => další krok se udělá podle zaktualizovaných výsledků
- Tic-Tac-Toe (piškvorky)
  - 3x3
  - Vyhrává, kdo udělá linku přes 3
  - Heuristika
    * Počet linek, které mohu udělat mínus, které může udělat soupeř
  - Zbytečně se budou prověřovat červené uzly



AlfaBeta

- Optimalizovaná verze MinMax
  - Vynoření, hned když se zjistí, že se uzel nemůže dostat do rodiče
- 2 registry pro aktuálně nejlepší výsledky obou hráčů

  - Alfa ... aktuální hráč (init => -inf)
  - Beta ... protivník (init => +int)
- Zanoření do určité hloubky (ořezání stavového prostoru)
  - Kopírování registrů směrem dolů (alfa => alfa, beta => beta)
  - Ale jen dokud platí alfa < beta
    * Při porušení => vynoření (nevrátí se žádná nová hodnota)
  - Zanoření s již novými hodnotami registrů z předchozího vynoření
- Propagování / navrácení hodnot směrem nahoru
  - Min (soupeř) / max (hráč A) mezi aktuální a navrácenou hodnotou
  - Navrácení jen do 1 registru (do alfy u sebe, do bety u soupeře)
    * Svůj uzel => navrácení alfa do protivníkova beta
    * Protivníkův uzel => navrácení beta do alfa
- Výsledek je jen 1. krok
  - Pokud větve mají stejné hodnoty => tu první (v levo)



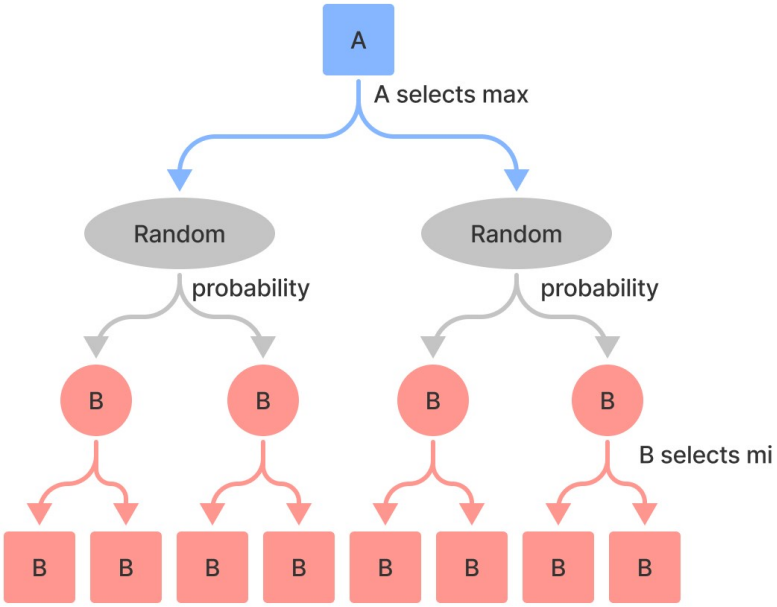
- Simulator: $https://raphsilva.github.io/utilities/minimax_simulator/\#$

Hry s neurčitostí

- Pomocí MinMax / AlfaBeta s upraveným výpočtem ohodnocení
- Výpočet ohodnocení
  - Hráč na tahu nejdříve zjistí výsledek náhodného jevu (pak se potřebuje rozhodnout)
  - Na počátku se jde do hloubky 3
    * Moje aktuální možnosti (vyberu MAX)
    * Různé výsledky náhodného jevu soupeře po mém rozhodnutí (váhování pravděpodobností)

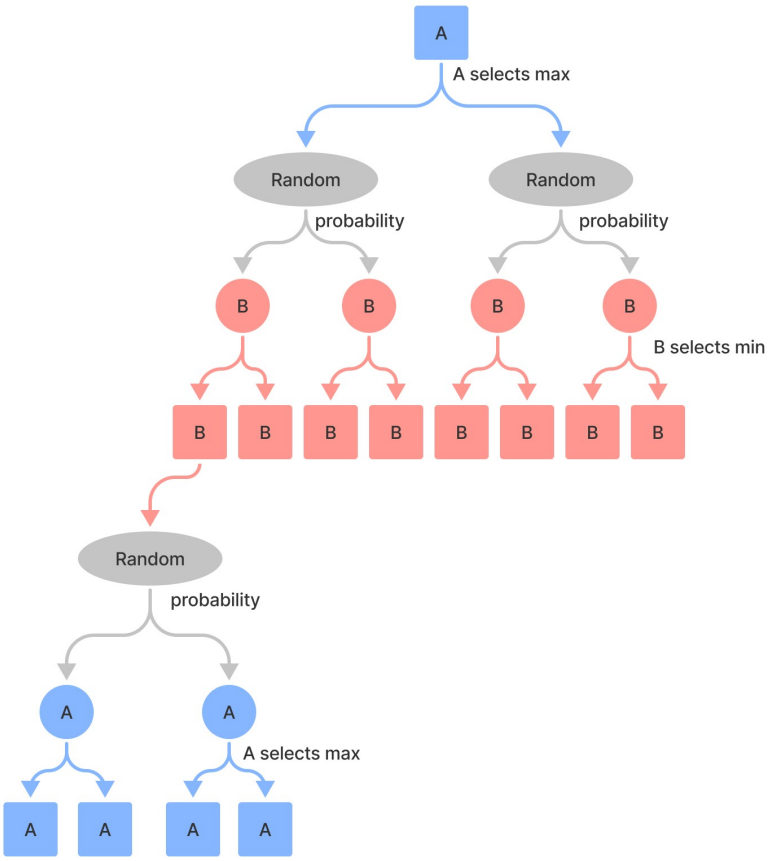
* Různé rozhodnutí soupeře po náhodném jevu (odhad heuristikou + výběr MIN)

$$\text{expectMin}(X) = \sum_{\text{možnosti rand jevu}} P(\text{výsledek i}) \cdot min(\forall \text{ možnosti soupeře po výsledku i})$$



– Pro větší přesnost lze expandovat listy + přepočítat rodiče
  * Různé možnosti náhodného jevu (váhování pravděpodobností)
  * Různé možnosti hráče A - pro dané hodnoty náhodného jevu (odhad heuristikou + výběr MAX)

$$\text{expectMax}(X) = \sum_{\text{možnosti rand jevu}} P(\text{výsledek i}) \cdot max(\forall \text{ mé možnosti po výsledku i})$$



* Další zanoření => expectMin (hází soupeř), ...

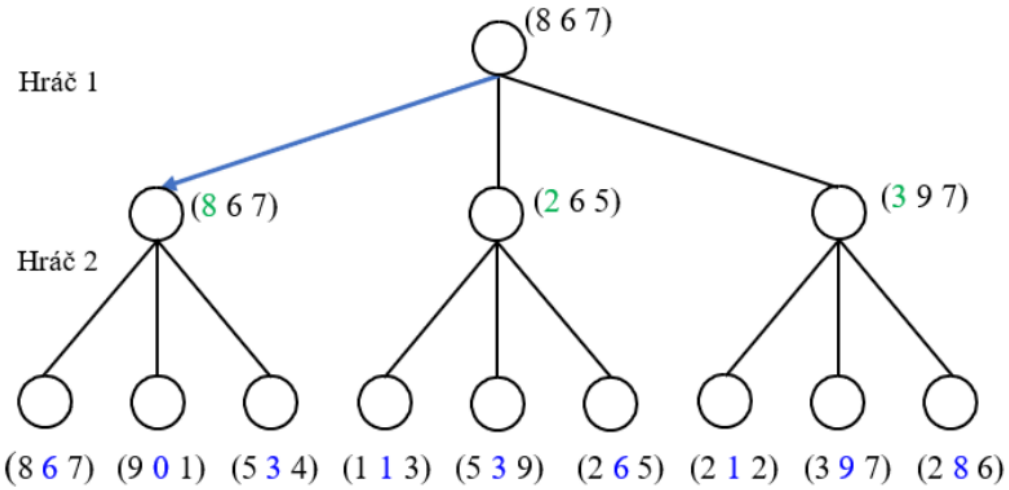
- Člověče nezlob se
  - Oběhnout se všemi figurkami + umísti všechny do domu
  - Lze vyhodit figurku soupeře
  - Stav: [pozice figurek hráče A, pozice B, odhad heuristiky]
  - Heuristika:

$$= 15(\text{A-home} - \text{B-home}) + 5(\text{A-not-home} - \text{B-not-home}) +$$
$$+2(\text{A-can-go-home} - \text{B-can-go-home}) + (\text{A-dangered-by-B} - \text{B-dangered-by-A})$$

  - Dále MinMax

Hry s N hráči

- Hráči se pravidelně střídají
- Stavy ohodnoceny vektory - heuristika pro každého hráče
  - (player 1, player 2, ...)
- Odhad vektorů heuristikou v listech
- Hráč na řadě si vybírá nejvýhodnější stav podle heuristik ve vektoru na jeho indexu
- Vektor nejvýhodnějšího stavu daného hráče se celý zkopíruje do rodiče



- Pokud má soupeř více stejných nejlepších možností => hráči výše si musí zvolit rozsah který jim lépe vyhovuje (odhad risku)
  - Větší min + malé max / velké max + malé min
- Exploitation x exploration
  - Zůstat na aktuální směru / zkusit nový
  - Malá šance získat velkou hodnotu / velká šance získat průměrnou