Algoritmy a programování

Stavový prostor a jeho prohledávání

Vojtěch Vonásek

Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

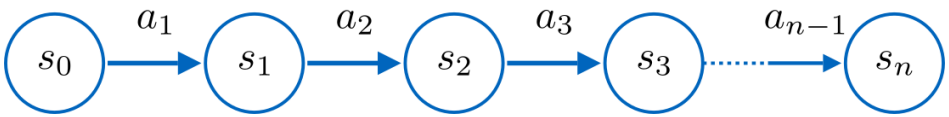
Stavový prostor

Definice

- S je množina všech stavů
- A je množina všech akcí
- $A(s), s \in S$ je množina akcí, které lze aplikovat ve stavu s
- $E(s, a) \in S, s \in S, a \in A(s)$ je výsledný stav získaný aplikací a ve stavu s
- $c(s, a), s \in s, a \in A(s)$ je cena provedení akce a ve stavu s
- Jako stavový prostor označujeme n-tici: $(S, A, A(s), E(s, a), c(s, a))$

Prohledávání stavového prostoru

- Počáteční stav $s_0 \in S$
- Cílové stavy $T \subseteq S$
- Přechodová funkce $f : S \times A \to S$
- Úkolem je najít sekvenci akcí $a_1, a_2, \ldots, a_n$, $a_i \in A$, tak, aby bylo dosaženo nějakého cílového stavu z T
- Výsledkem hledání je též sekvence stavů $s_1, s_2, \ldots, s_n$ tak, že $s_i = f(s_{i-1}, a_{i-1})$



Použití

- Plánování akcí (stavebnictví, vykládka lodí, výrobní postupy)
- Plánování pohybu v robotice
- Hry, počítačové hry
- Řešení hlavolamů
- a mnoho dalších ...

Příklad

Patnáctka

8		5
4	3	6
1	2	7

- Úkolem je seřadit hrací kameny
- Dílky lze posouvat do volného místa
- Stav: matice čísel, $s = M_{i,j}$, $i, j = 1, \ldots 3$
- Akce $A = \{R, L, U, D\}$ (pohyb dílku do mezery vlevo, vpravo, nahoru, dolů)
- Ne vždy lze provést všechny akce

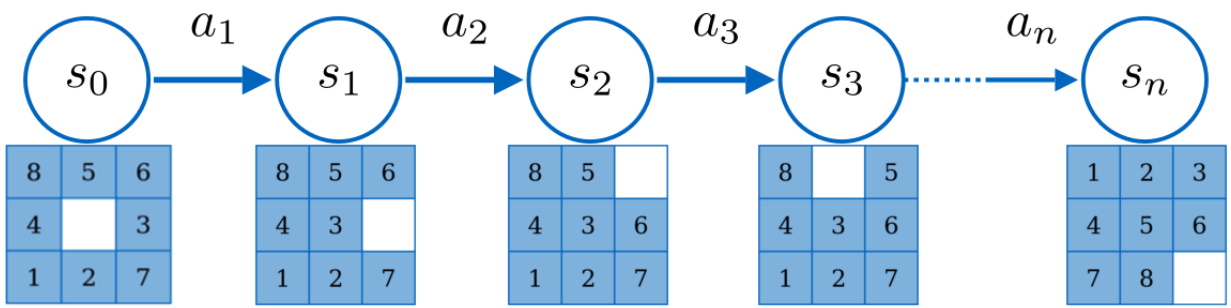
```
8 0 5
4 3 6
1 2 7
```

Příklad sekvence

- Počáteční stav: `s=[[8,5,6],[4,0,3],[1,2,7]]`
- Cílový stav: `s=[[1,2,3],[4,5,6],[7,8,0]]`
- Akce: $a_1 = L, a_2 = D, a_3 = R$

1	2	3
4	5	6
7	8	

Cílový stav



Prohledávání stavového prostoru

Neinformované

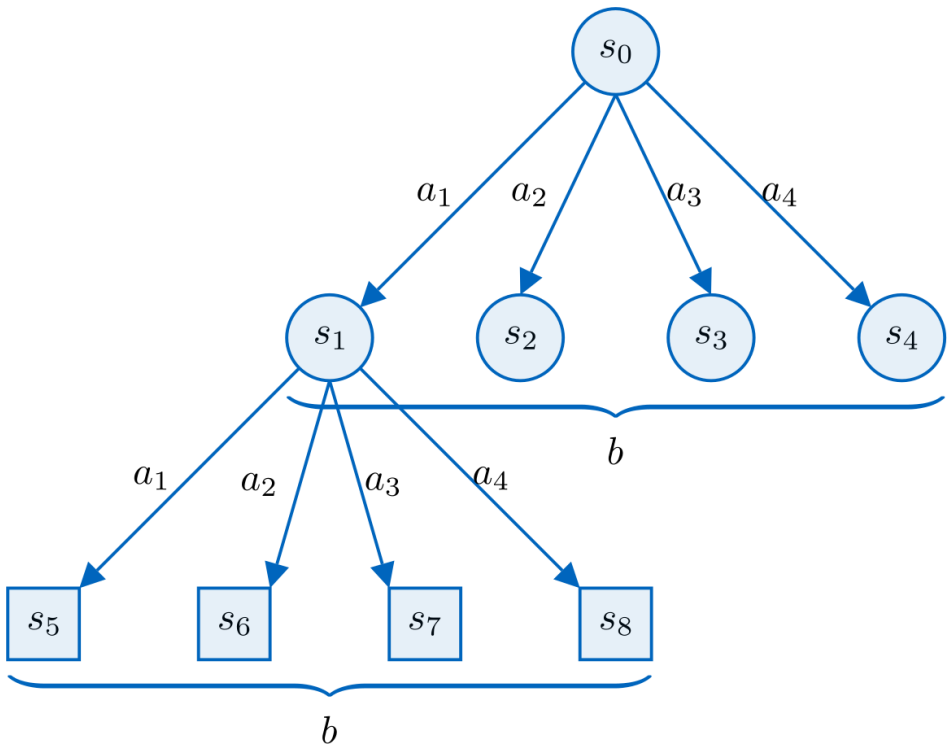
- Systematické prohledání celého prostoru
- Úkolem je najít cílový stav (a cestu do něj)
- Případně zjistit, jestli nějaký stav existuje
- Neznáme ohodnocení stavů (nevíme, v jakém pořadí je prohledávat)
- Například metody BFS, DFS

Informované

- Známe ohodnocení stavů
- Přesné ohodnocení vs. heuristika
- Prohledáváme dle ohodnocení stavů (“lepší” stavy prohledáme dříve)
- Při vhodné ohodnocovací funkci (heuristice) lze prohledání stavového prostoru významně urychlit
- Např. best-first search, beam-search, A*

Terminologie: strom prohledávání

- Zobrazení toho, jak algoritmy prohledávají stavový prostor
- Začínáme v počátečním stavu s_0
- Expanze stavu s_0: aplikace všech přípustných akcí dostáváme potomky — stavy s_1, s_2, s_3 a s_4
- Na každý z nich můžeme aplikovat přípustné akce, atd. pro každý stav
- Například expanze s_1 vede na stavy s_5, s_6, s_7 a s_8
- Stavy s_5, s_6, s_7 a s_8 jsou koncové
- Kolik akcí lze aplikovat (průměrně) — faktor větvení b (branching factor)



Reprezentace stavu prohledání

- Ukládáme stav problému (pole, samostatné proměnné, obrázek, . . .)
- Pozor: v Pythonu je vhodné kopírovat celý stav
- Vazbu na předchůdce ve stromu hledání + informaci o akci, která do stavu vedla
- Metody: `expand()`, `isGoal()`
- Pro uchování navštívených stavů použijeme `__repr__()`

```
1  import copy
2  class State:
3      def __init__(self, state, action = None, parent = None):
4          self.state = copy.deepcopy(state) # important to copy !!!!
5          self.parent = parent #parent in the search tree
6          self.action = action #action to this state
7      def expand(self):
8          result = []
9          #algorithm returns list on new States, each points to
10         #'self' in their parent
11         return result
12     def isGoal():
13         #return true if self.state is goal
14         return False
15     def __repr__(self):
16         return str(self.state)
```

Prohledávání do šířky

- Podobný princip jako BFS pro grafy
- Přechody mezi stavy jsou určeny přechodovou funkcí
- Prvky k prozkoumání jsou uloženy ve frontě

BSF

- Vlož počáteční stav s_0 do fronty, označ s_0 jako navštívený
- Dokud není fronta prázdná:
  - s_n = vyjmi prvek z fronty
  - Pokud $s_n \in T$, konec, vrať cestu z s_n do s_0
  - Pro všechny akce a, které lze aplikovat ve stavu s_n:
    - $s = f(s_n, a)$ (aplikuj akci a na stav s_n)
    - Pokud není s navšívený, vlož ho do fronty a označ jako navštívený
- Pokud je fronta prázdná, řešení nebylo nalezeno

Prohledávání do šířky: implementace

- Funkční implementace bude probrána na cvičeních

```
1  from stateNode import State
2  from queue import Queue
3  #pseudocode of BFS for state-space search
4  def bfs(start, goal):
5      q = Queue()
6      q.put( start )
7      known = {}
8      known[ str(start) ] = True
9      while not q.empty():
10         actual = q.get()
11         if actual.isGoal():
12             path = traverse(actual)
13             return path[::-1]
14         for child in actual.expand():
15             if not str(child) in known:
16                 known[n str(child) ] = True
17                 q.put( child )
18     return []
19
20 Start = State( [] ) #define start here
21 Goal = State( [] ) #define goal here
22 path = bfs(Start, Goal)
```

Prohledávání do šířky: vlastnosti

- Neinformované prohledávání
- V nejhorším případě musí navštívit všechny možné stavy $n = |S|$
- Stavy k prohledání jsou ve frontě
- Velká paměťová náročnost $\mathcal{O}(b^d)$, maximálně $\mathcal{O}(n)$
- Velká časová náročnost $\mathcal{O}(b^d)$, maximálně $\mathcal{O}(n)$
- Vždy najde řešení (bez omezení hloubky)
- Vždy najde nejkratší řešení (z pohledu počtu akcí vedoucích ze startu do cíle)

Prohledávání do hloubky

- Přechody mezi stavy jsou určeny přechodovou funkcí
- Prvky k prozkoumání jsou uloženy v zásobníku

DFS

- Vlož počáteční stav s_0 do zásobníku, označ s_0 jako navštívený
- Dokud není zásobník prázdný:
  - s_n = vyjmi prvek ze zásobníku
  - Pokud $s_n \in T$, konec, vrať cestu z s_n do s_0
  - Pokud je s_n navštívený, pokračuj další iterací (continue)
  - Označ s_n jako navštívený
  - Pro všechny akce a, které lze aplikovat ve stavu s_n
    - Aplikuj akci, vlož výsledný stav do zásobníku
- Pokud je zásobník prázdný, řešení nebylo nalezeno

Implementace

- Stejně jako BFS na přechozích slidech, pouze nahradíme frontu zásobníkem

Prohledávání do hloubky: vlastnosti

- Neinformované prohledávání
- Stavy k prohledání jsou v zásobníku
- Menší paměťová náročnost než u BFS — $\mathcal{O}(d)$
- Časová náročnost: $\mathcal{O}(b^d)$ pro hloubku d a větvení b
  - Maximálně $\mathcal{O}(n)$, $n = |S|$ počet stavů
- Vhodné pokud kompletní změna stavu není jednoduchá (rychlá) — fyzické prohledávání
- Vždy najde řešení (bez omezení hloubky)
- Negarantuje nalezení nejkratšího řešení

Varianty DFS

- Omezení maximální hloubky — negarantuje nalezení řešení, pokud je "za hloubkou" prohledání
- Pokud si nepamatujeme navštívené stavy, pak se algoritmus může zacyklit
  - To platí i pro BFS, pokud nepoužíváme paměť navštívených stavů

Informované prohledávání

- Uvažujeme cenu (kvalitu) stavů
- Pokud nemáme přesné ohodnocení stavu, použijeme heuristiku
- Stavy čekající na zpracování organizujeme v prioritní frontě
- Časová a paměťová náročnost stejná jako u prohledávání do šířky

Heuristika

- Aproximace ceny (vhodnosti) stavu pokud neznáme přesnou funkci pro cenu stavu
- Heuristika závisí na problému, který řešíme
- Musí být výpočetně jednoduchá, nenáhodná, konzistentní
- Příklad:
  - Hledání cesty v mřížce (gridu): Euklidovská vzdálenost do cíle

Informované prohledávání: best-first search

- Modifikace BFS/DFS
- Prvky ve frontě jsou seřazeny dle hodnoty stavu/heuristiky (tj. “nejslibnější” stavy jsou dříve)
- Efektivní reprezentace: prioritní fronta (viz další přednášky)
- Zbytek prohledání je stejný jako u BFS/DFS

Implementace

- Kromě reprezentace stavů (třída State) potřebujeme heuristiku
- Funkce, která stavu přiřazuje číslo (např. čím lepší stav, tím menší číslo)

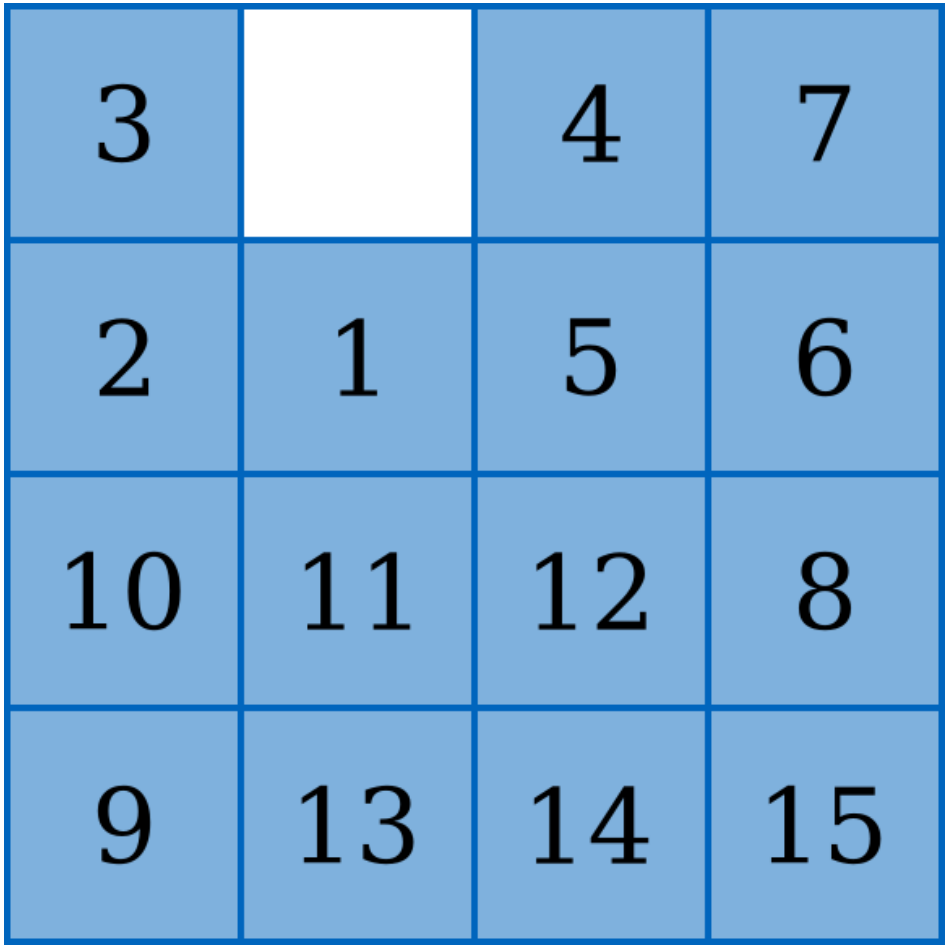
```
1 from stateNode import State
2 #pseudocode of best-first search (not effective way!)
3 #we mimick priority queue by sorting
4
5 def heuristic(state):
6     # heuristic for this state (better = lower value)
7     return 0
```

Informované prohledávání: best-first search

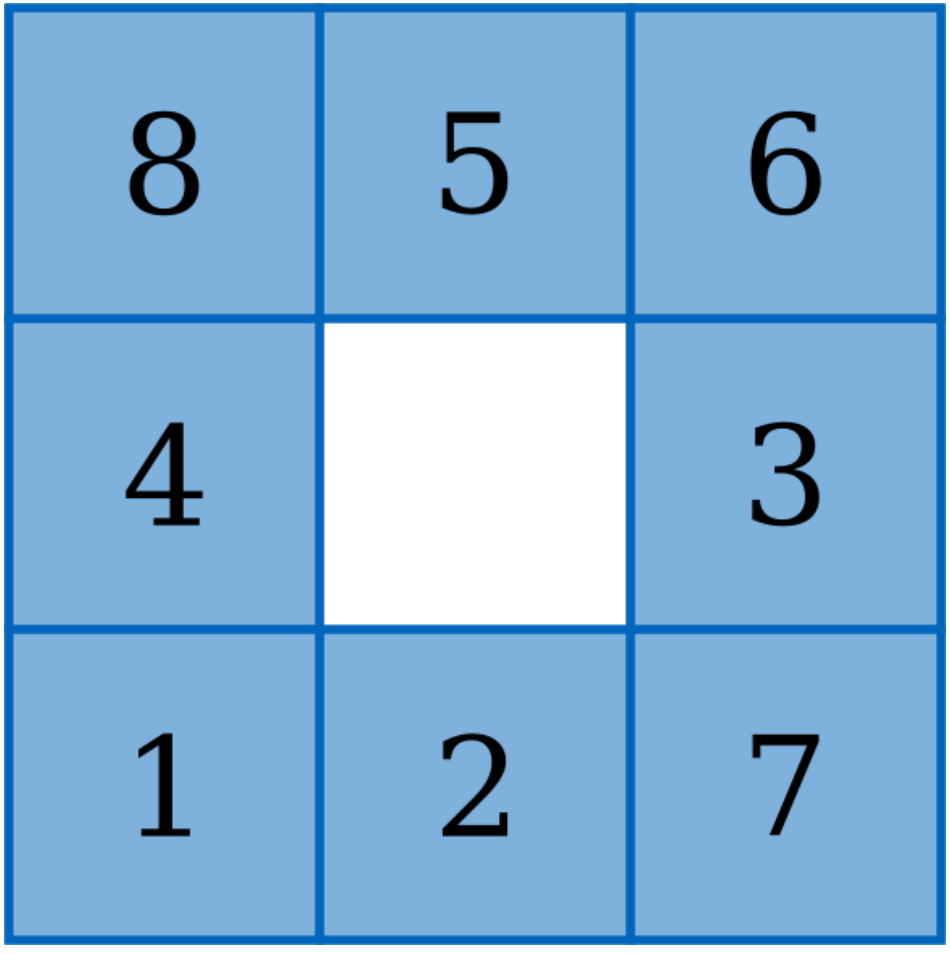
- Kostra implementace pro Python

```
1  def bestfs(start, goal):
2      queue = [ start ]
3      known = {}
4      known[ str(start) ] = True
5      while not q.empty():
6          actual = queue.pop()
7          if actual.isGoal():
8              path = traverse(actual)
9              return path[::-1]
10         newItems = False
11         for child in actual.expand():
12             if not str(child) in known:
13                 known[n str(child) ] = True
14                 queue.insert(0, child )
15                 newItems = True
16         if newItems:
17             queue.sort(key=heuristic)
18     return []
19
20 Start = State( [] ) #define start here
21 Goal = State( [] ) #define goal here
22 path = bestfs(Start, Goal)
```

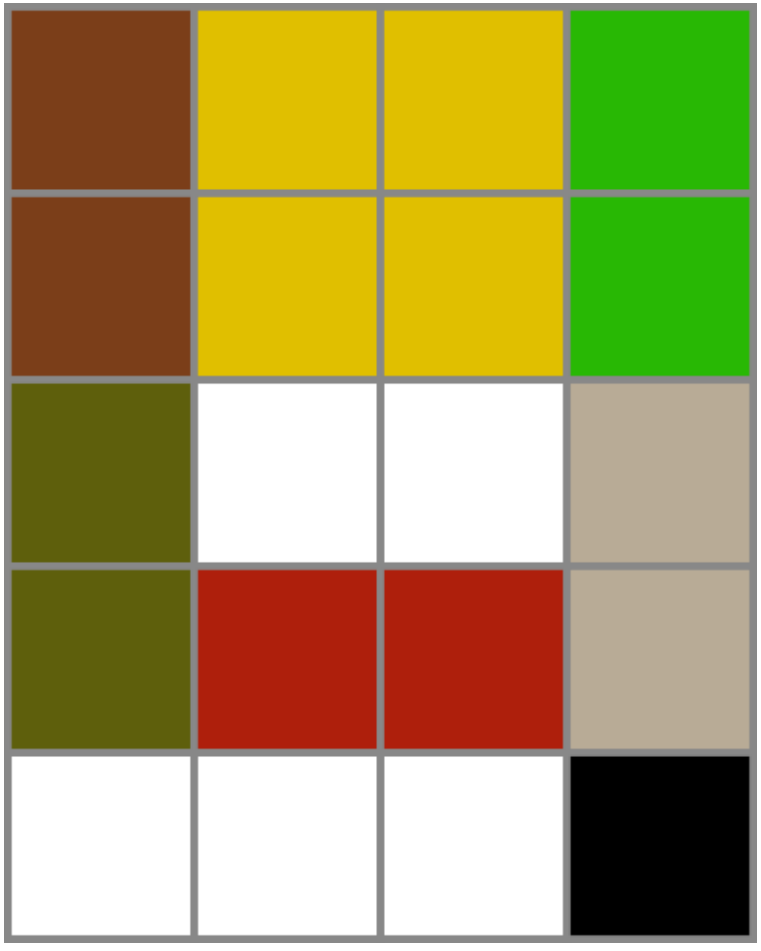
Stavový prostor



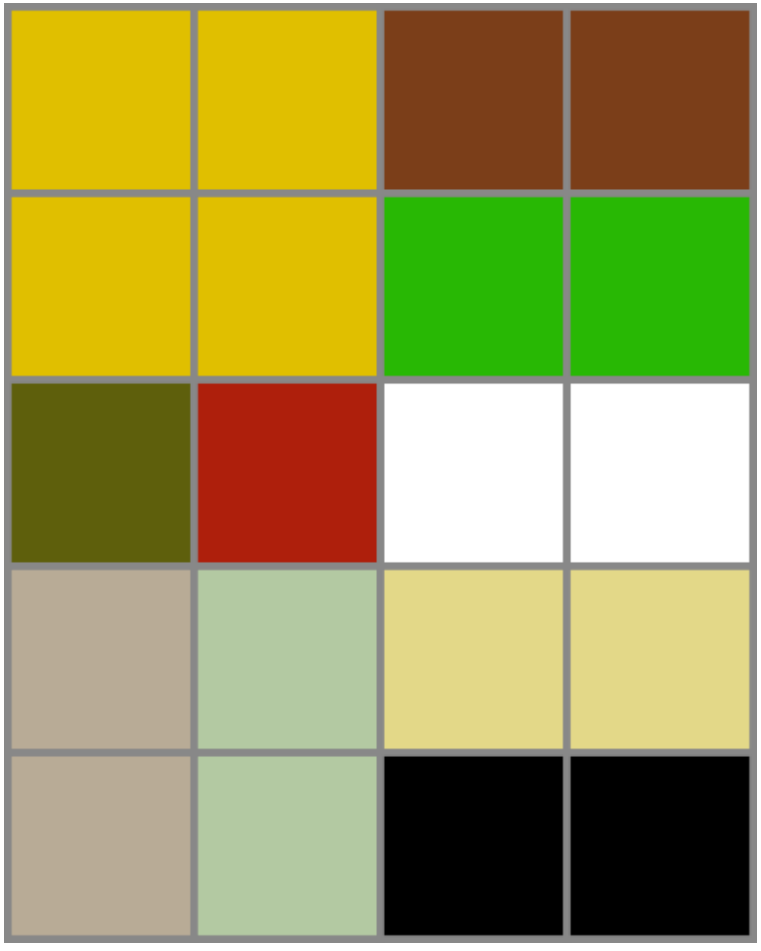
Stavový prostor



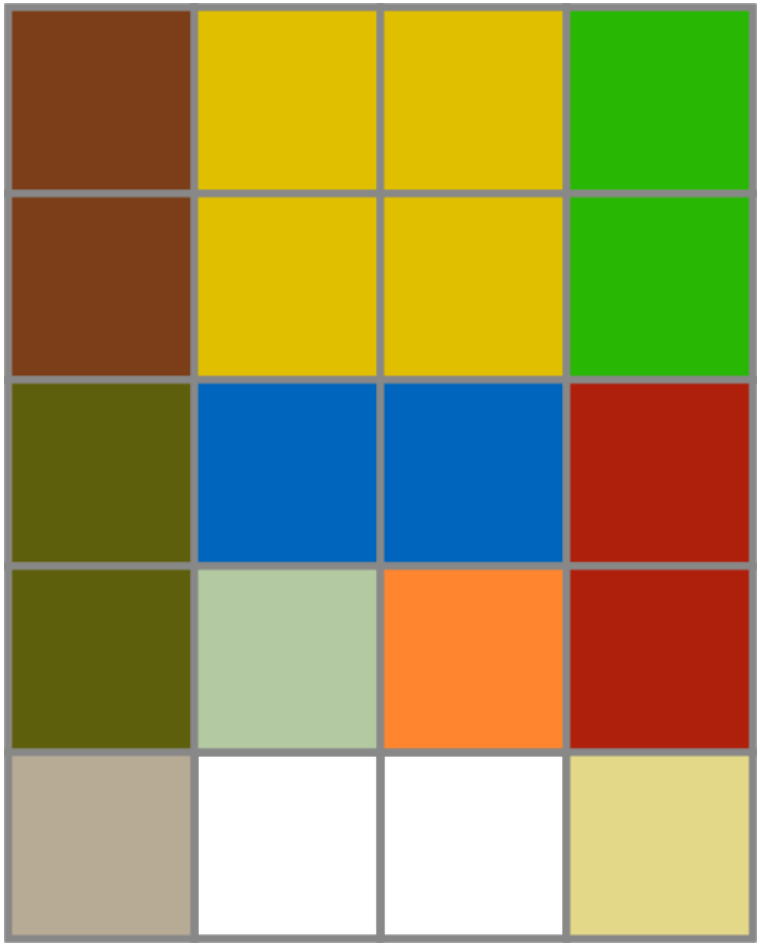
Prohledávání stavového prostoru: hlavolamy



Prohledávání stavového prostoru: hlavolamy



Prohledávání stavového prostoru: hlavolamy



Definice cílového stavu $T \subseteq S$

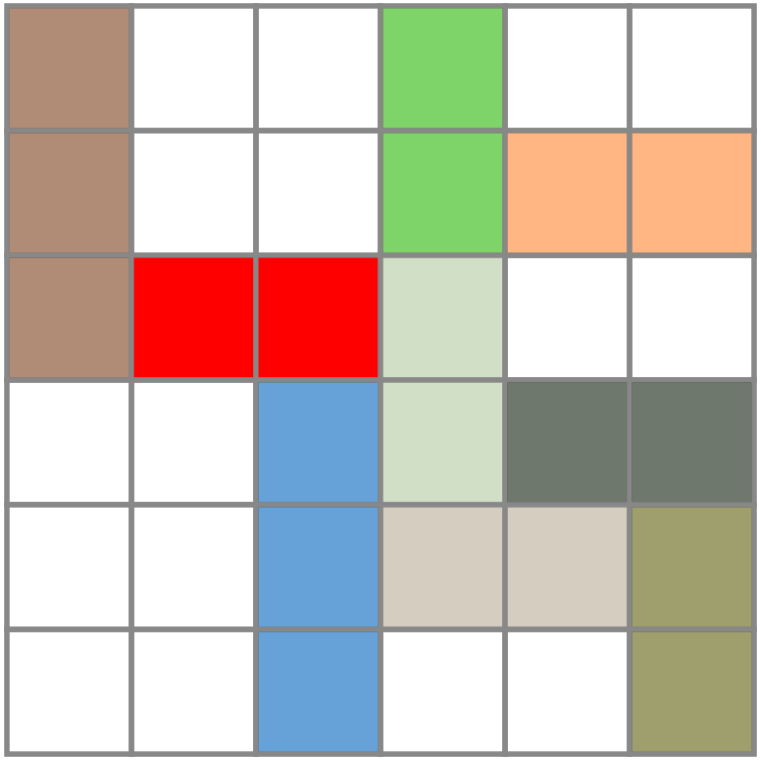
Explicitně

- Cílový stav/stavy $T \subseteq S$ jsou uvedeny explicitně (výčtem prvků)
- Například:
  - Koza/Vlk/Zelí: všechna zvířata jsou na druhém břehu
  - Patnáctka: Výsledné pořadí je $1, 2, 3, \ldots, 15$
  - Hledání cest v bludišti: cíl je (x, y)
- Vhodné pro specifikaci heuristiky
- Dosažení cíle testujeme jako rovnost stavu a cílových stavů

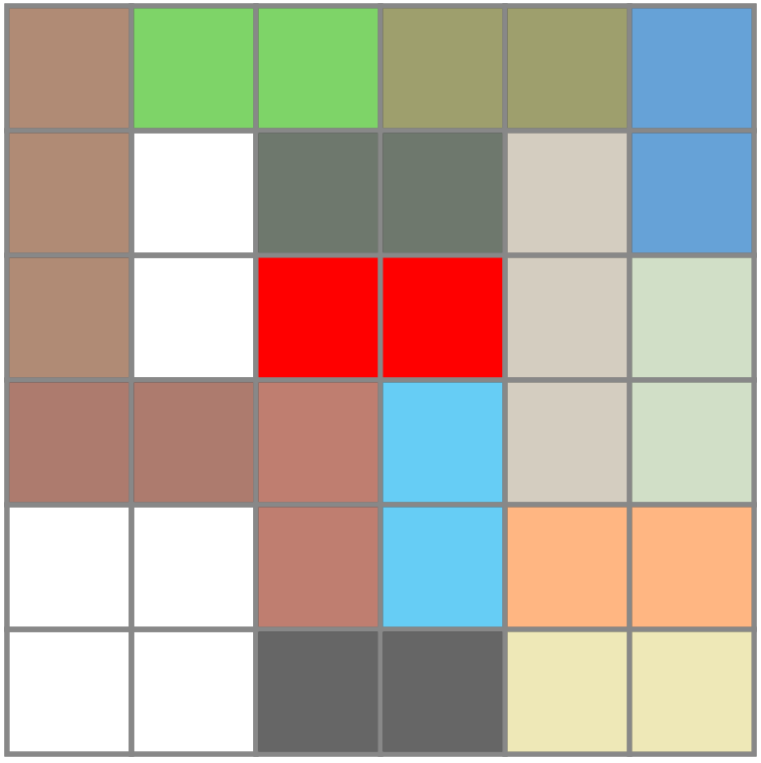
Implicitně

- Cílová množina je zadána funkcí, např. $f(s) = 0$ pokud $s \in T \subseteq S$
  - Klotski: jakékoliv rozložení, kde žlutý kámen je uprostřed dole
  - Hive: jakákoliv situace, kde jedna ze včel je plně obklopena
- Neznáme, kolik stavů je cílových, jak jsou distribuovány
- Složitější na návrh heuristiky

Prohledávání stavového prostoru: hlavolamy



Prohledávání stavového prostoru: hlavolamy



Prohledávání stavového prostoru her

- Teorie her — vědecká disciplína na pomezí matematiky, computer-science, kybernetiky, ekonomie, sociologie
- Studium (konfliktních) problémů, kde účastníci stojí proti sobě (soupeří)
- Řešení konfliktů ve společenosti, ekonomii, války
- Cílem je najít strategii pro jednotlivé hráče
- Nejznámější aplikace: řešení her — šachy, go, Othello, ...

Typy her

- Dělíme podle toho, kolik informací víme (například o hrací desce, zdrojích protihráče, jeho zájmech)
- Náhodné vs. deterministické akce

	Akce	
	Deterministická	Náhodná
Úplná informace	šachy, othello, hive, go, ...	Vrhcáby, člověče nezlob se
Neúplná informace	lodě	bridge, poker, scrabble, válečné konflitky

Předpoklady

V ALP budeme uvažovat pouze tyto hry:

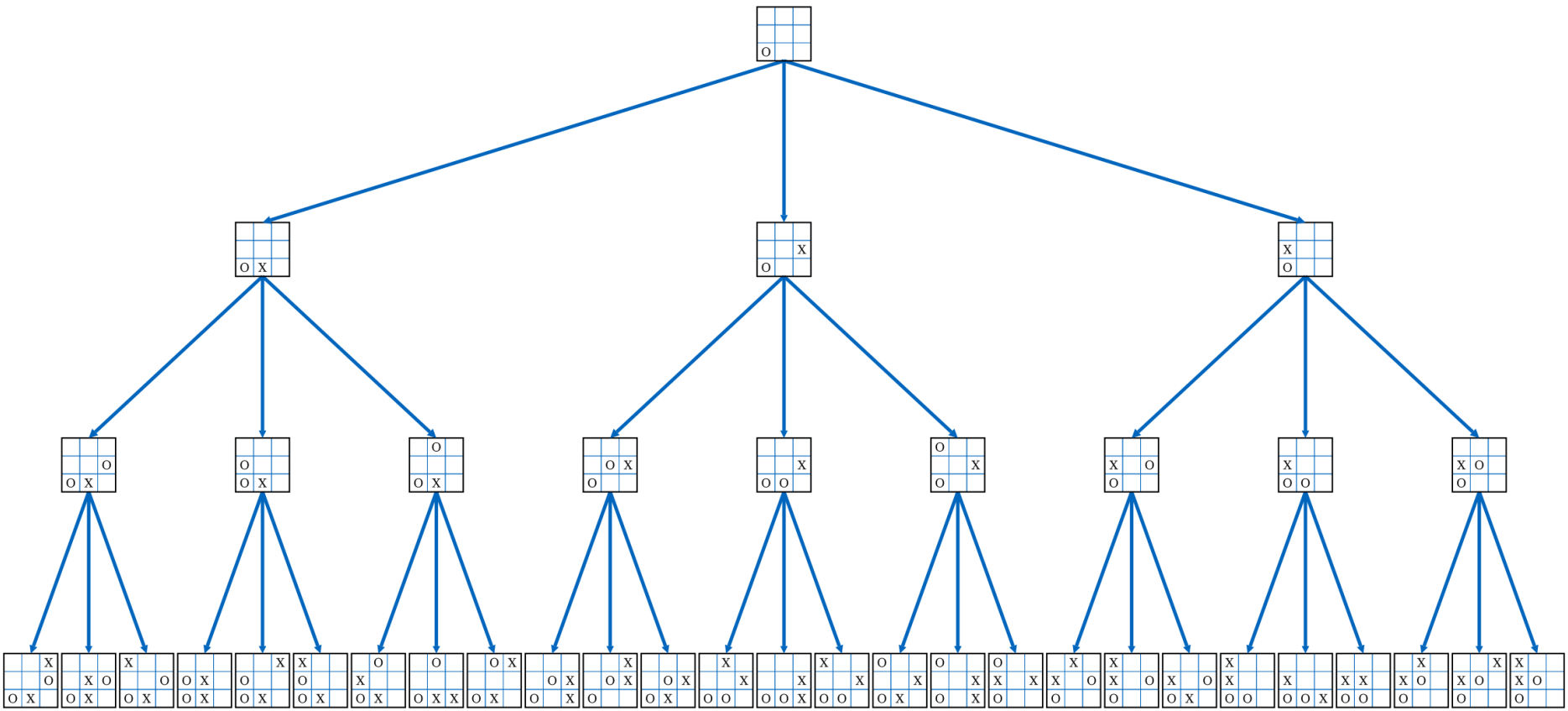
- Hra s nulovým součtem — zisk jednoho hráče je na úkor (ztráty) protihráče
- Deterministická hra s úplnou informací
- Hráči se střídají
- Hry dvou hráčů

Co je cílem

- Známe akce hráčů, známe stav hry (stavový prostor)
- Nyní se ve stavovém prostoru pohybují dva hráči a mají opačné zájmy
- Chceme najít strategii, tj. každému stavu $s \in S$ přiřadit akci $a \in A$ tak, aby to pro hráče na tahu bylo výhodné

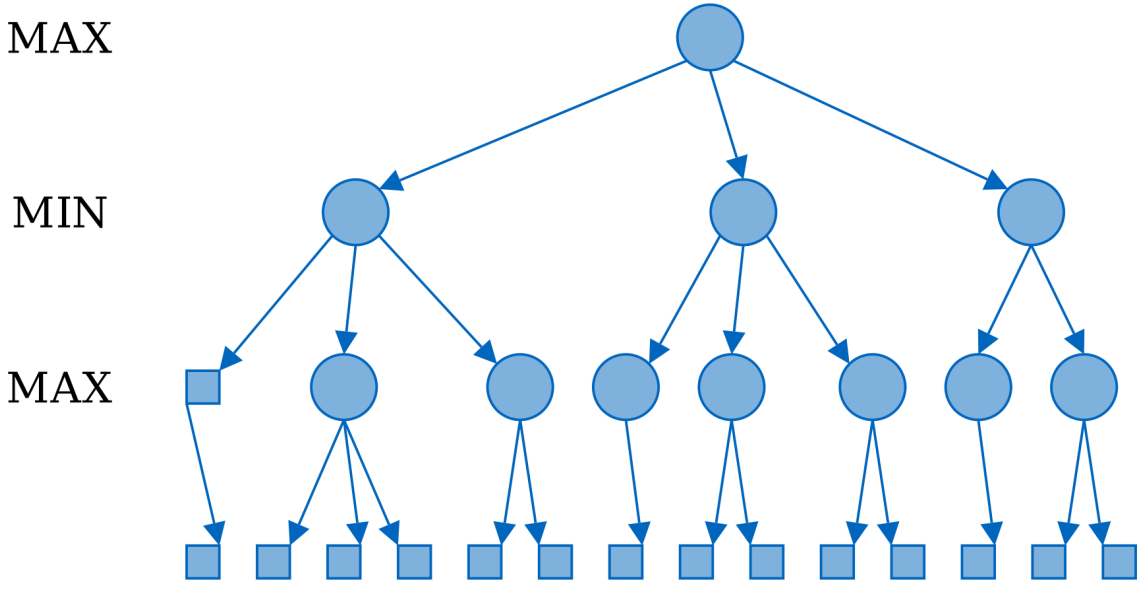
Minimax

- Dva hráči — MIN a MAX (hráč MAX maximalizuje zisk a naopak)
- Začíná hráč MAX, hrají tak dlouho, až je hra u konce
- Hráči se střídají a společně prohledávají stavový prostor hry
- Minimax poskytuje tah pro hráče MAX, který maximalizuje zisk pro nejhorší možný případ

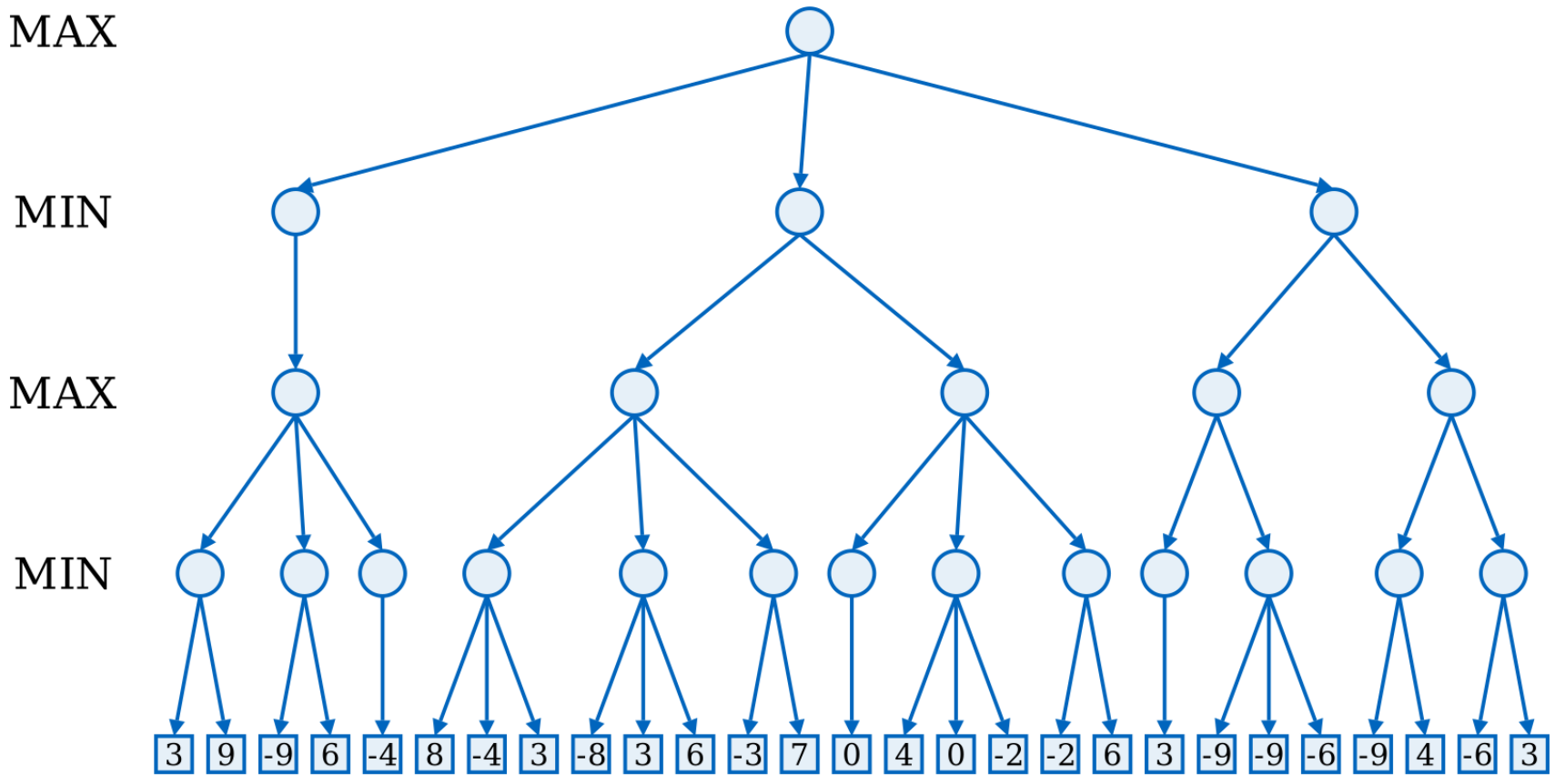


Minimax

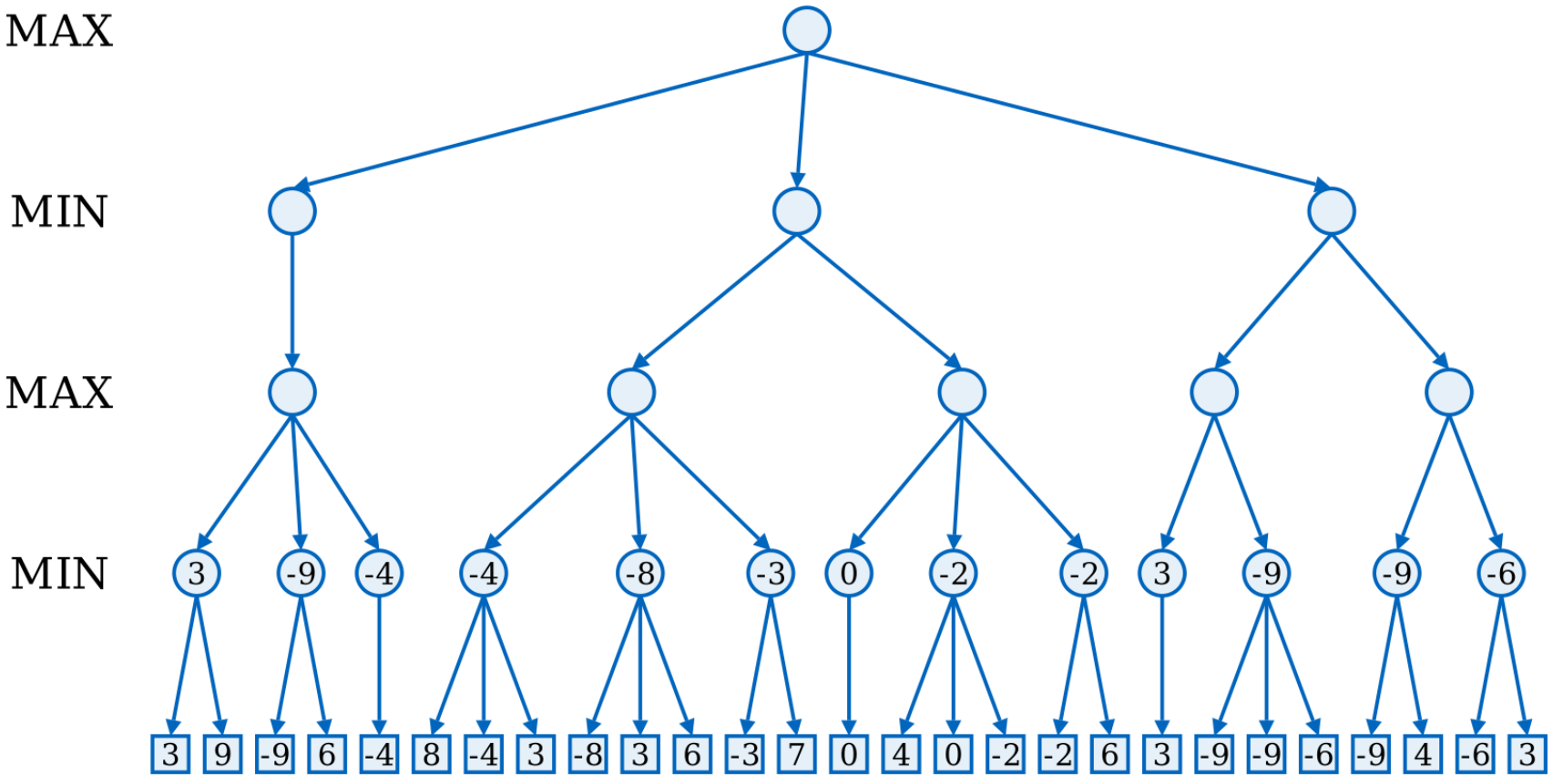
- Sestavit úplný strom hry od počátečního stavu až do každého koncového stavu
- Výpočet hodnoty koncových stavů (reward)
- Propagace hodnot z listů směrem k rodičům
  - Hodnoty stavů v MAX uzlech jsou maxima jejich následníků
  - Hodnoty stavů v MIN uzlech jsou minima jejich následníků
- Kořen je na úrovni MAX, tj. hráč MAX vybírá akci, která je maximum jeho potomků



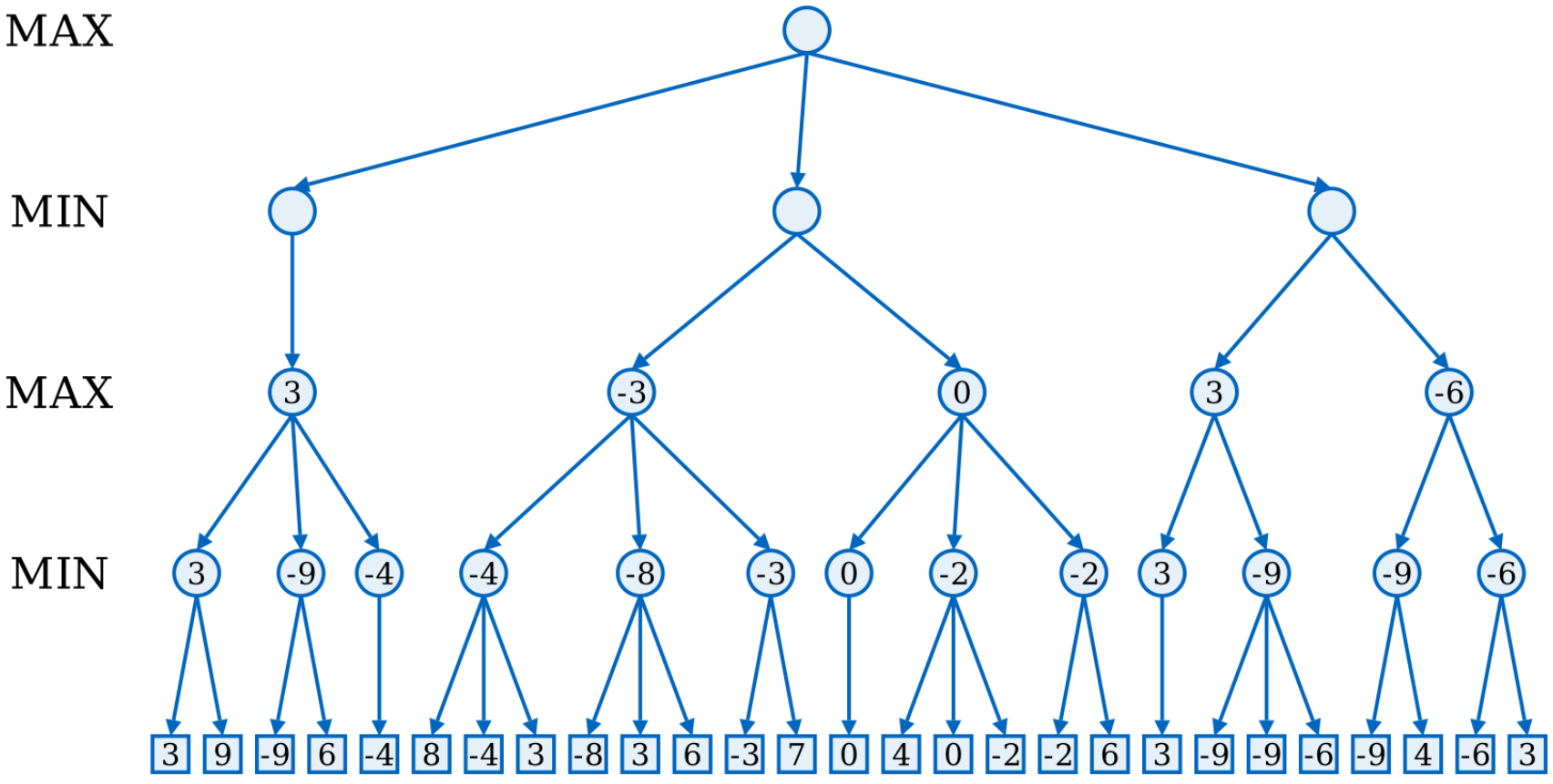
- Propagace hodnot z listů směrem k rodičům
  - Hodnoty stavů v MAX uzlech jsou maxima jejich následníků
  - Hodnoty stavů v MIN uzlech jsou minima jejich následníků



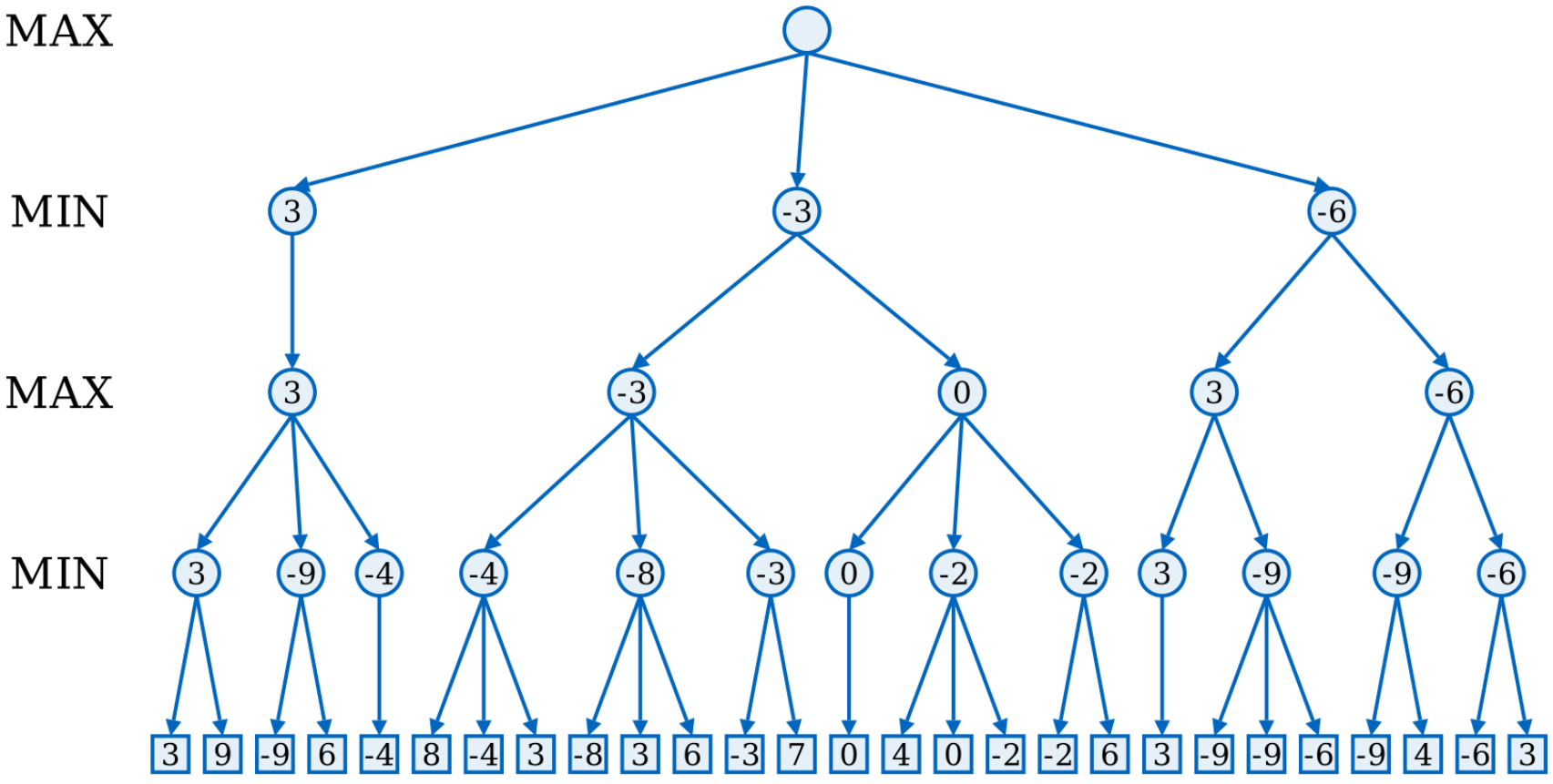
- Propagace hodnot z listů směrem k rodičům
  - Hodnoty stavů v MAX uzlech jsou maxima jejich následníků
  - Hodnoty stavů v MIN uzlech jsou minima jejich následníků



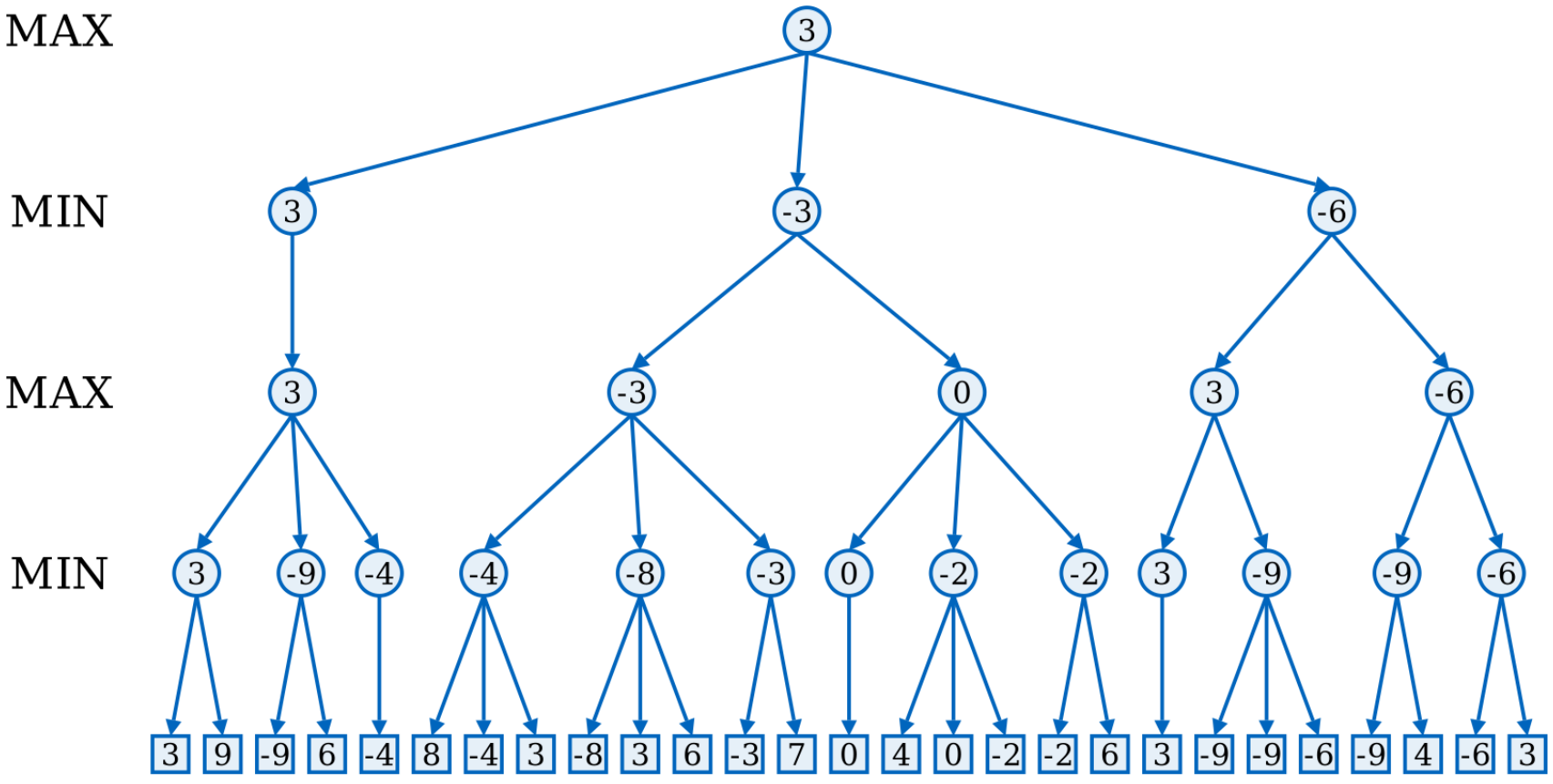
- Propagace hodnot z listů směrem k rodičům
  - Hodnoty stavů v MAX uzlech jsou maxima jejich následníků
  - Hodnoty stavů v MIN uzlech jsou minima jejich následníků



- Propagace hodnot z listů směrem k rodičům
  - Hodnoty stavů v MAX uzlech jsou maxima jejich následníků
  - Hodnoty stavů v MIN uzlech jsou minima jejich následníků



- Propagace hodnot z listů směrem k rodičům
  - Hodnoty stavů v MAX uzlech jsou maxima jejich následníků
  - Hodnoty stavů v MIN uzlech jsou minima jejich následníků



- Rekurzivní varianta Minimaxu

```
1  def minimax(state, depth, maximizingPlayer):
2      if depth == 0 or state.isGoal():
3          return state.heuristic()
4
5      if maximizingPlayer:
6          value = None
7          for child in state.expand():
8              childValue = minimax( child, depth-1, False)
9              if value == None or childValue > value:
10                 value = childValue
11         return value
12     else:
13         value = None
14         for child in state.expand():
15             childValue = minimax( child, depth-1, True)
16             if value == None or childValue < value:
17                 value = childValue
18         return value
```

Minimax

- Použití algoritmu Minimax pro MAX hráče
- Vyzkoušíme všechny akce v určeném stavu x
- Na každý stav (akci) zavoláme Minimax pro MIN hráče
- Vybereme stav (akci) s nejvyšší hodnotou

```
1 def selectAction(state, depth):
2     bestAction = None
3     bestValue = None
4     for child in state.expand():
5         value = minimax(child, depth, False)
6         if bestValue == None or value > bestValue:
7             bestValue = value
8             bestAction = child.action
9     return bestAction, bestValue
```

- Pokud se stejné stavy objevují vícekrát, je dobré uchovávat si jejich hodnotu a při dalším vyhodnocení ji použít
- Vyšší paměťové nároky, v mnoha případech (např. šachy) není možné z důvodu paměti
- Řešení: použít hash stavu

Minimax: vlastnosti

- Kompletní — vždy najde řešení (pro konečný strom)
- Optimální — pouze pokud MIN hráč hraje také optimálně
  - Musí používat stejné kritérium
- Pokud se strom staví rekurzivně nebo stylem DFS:
  - Časová složitost $\mathcal{O}(b^d)$, branching factor b, hloubka d
  - Paměťová složitost $\mathcal{O}(bd)$

Příklady

- Piškvorky 3×3
  - $b \sim 5$ (průměrně), celkově 9 tahů
  - $5^9 = 1\,953\,125$ stavů
  - Lze spočítat optimální řešení
- Šachy
  - $b \sim 35$ (průměrný branching factor)
  - $d \sim 80$ (průměrný počet tahů)[1]
  - $b^d \sim 3.53077 \cdot 10^{123}$ stavů
  - Nalézt optimální řešení je prakticky neupočitatelné

[1] velký rozptyl mezi hrami

Heuristická funkce $h(x)$

- Odhad výhodnosti pozice (stavu hry x) pro určeného hráče
- Lze zahrnout i odhad výhodnosti pozice pro protihráče, např. $h(x) =$ f(hráč,x) - f(soupeř,x)
- Statická heuristika — odhad pouze na základě stavu
- Výsledek je v rozsahu např. $[-1, 1]$ (-1 určitě prohra, 1 určitě výhra)

Příklady

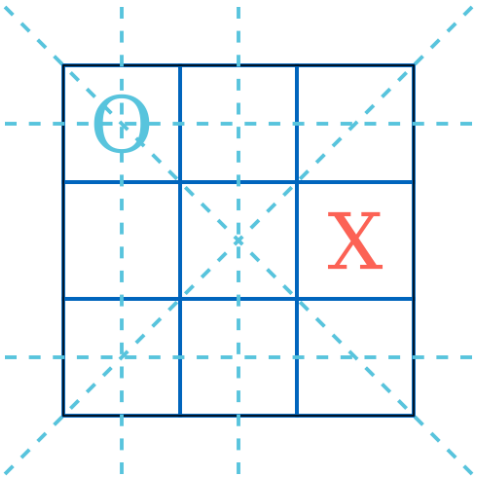
- Reversi/Othello: počet bílých kamenů - počet černých kamenů
- Šachy: vážený součet hodnot figurek, $f_i(x)$ je počet bílých figurek - počet černých figurek daného typu i

$$h(x) = \sum_{i=\{dama,vez,strelec,kun,pesec\}} w_i f_i(x)$$

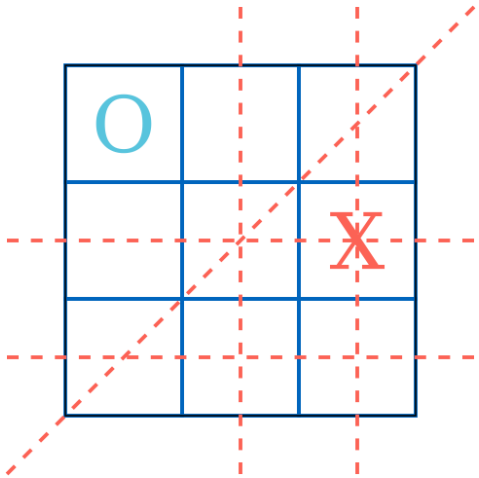
					
w_i	9	5	3	3	1

Příklady

- Piškvorky: $h(x)$ je rozdíl počtu možných výherních přímek



O má celkem 6 možností



X má celkem 5 možností

- Heuristika je $h(x) = 6 - 5 = 1$ (výhoda pro hráče *O*)
- Tato heuristika nemá rozsah hodnot $[-1, 1] \rightarrow +1$ neznamená určitou výhru

Alfa-beta prořezávání

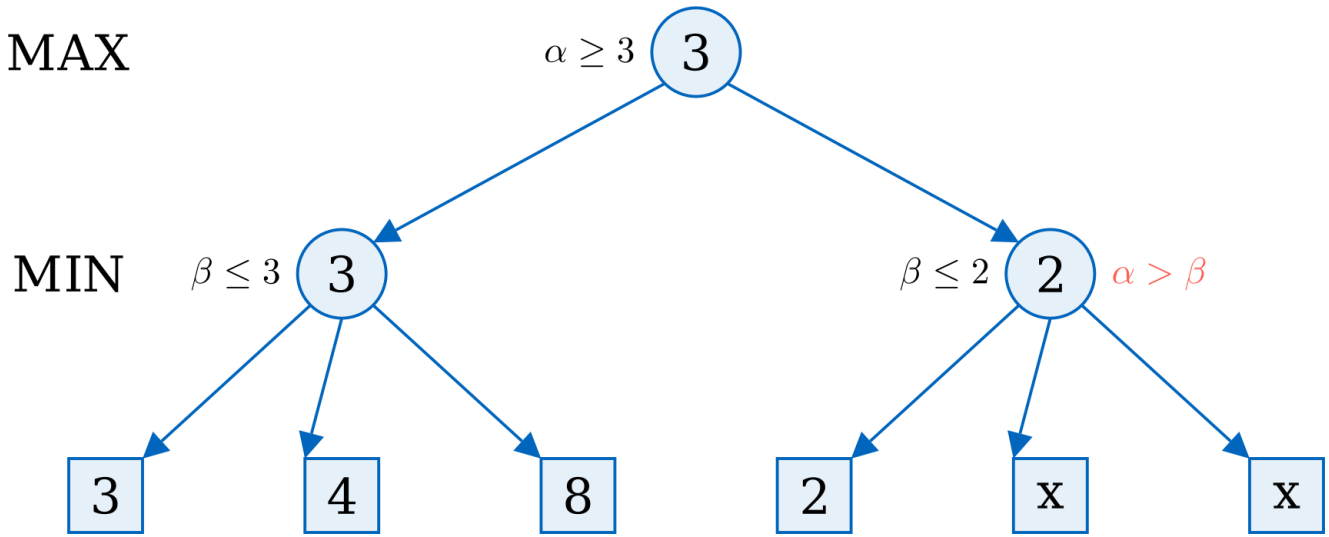
- Verze algoritmu Minimax
- Známe doposud nejlepší hodnotu α pro hráče MAX, a doposud nejlepší hodnotu β pro MIN hráče.
- Tyto hodnoty předáváme do vyhodnocení potomků
- Odřízneme (tj. nevykonáme) vyhodnocení potomků, pokud $\alpha \geq \beta$
- Nevykonávají se expanze uzlů v případě, že určitě máme lepší řešení

Odhady α, β

- α je nejvyšší hodnota pro MAX hráče na cestě z uzlu do jeho potomků
  - α se aktualizuje na úrovni MAX hráče
- β je nejnižší hodnota pro MIN hráče na cestě z uzlu do jeho potomků
  - β se aktualizuje na úrovni MIN hráče
- Na začátku je $\alpha = -\infty$ a $\beta = +\infty$

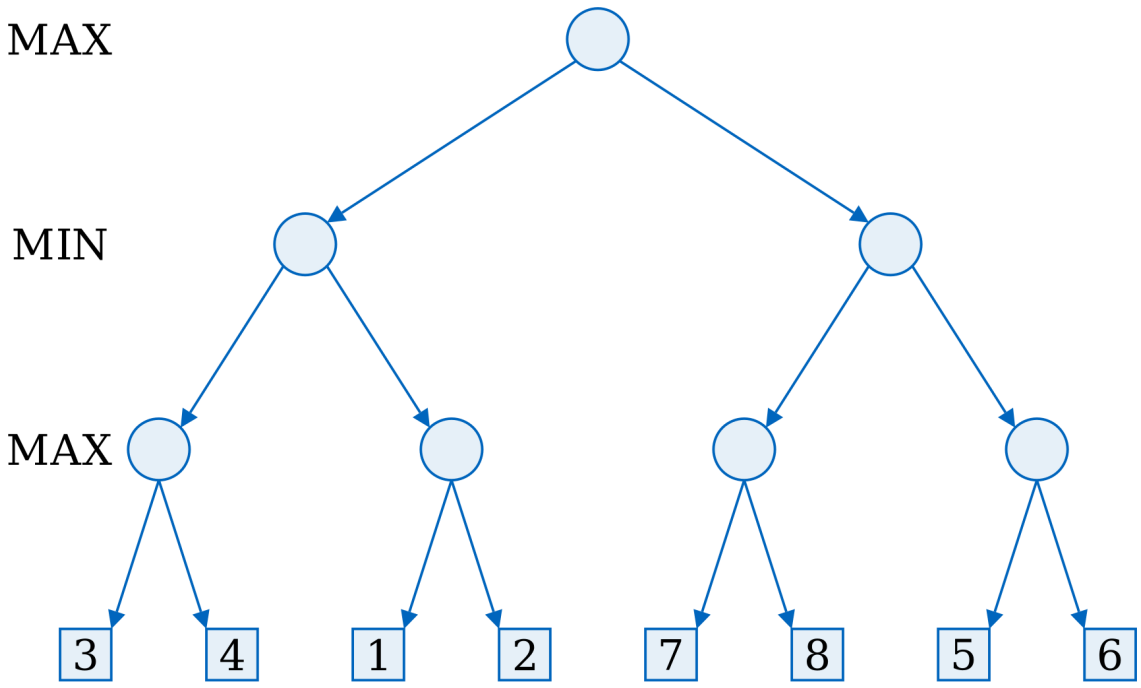
Alfa-beta prořezávání

- Příklad prořezání uzlů “x” (nejsou vyhodnoceny)
- Nejlepší cena levého MIN potomka je 3 (určitě), nejlepší cena pravého MIN potomka je ≤ 2
- Je jedno, jestli ještě klesne cena pravého MIN potomka (pokud by byly vyhodnoceny stavy “x”)
- Stavy “x” mohou zmenšit cenu pravého MIN potomka, ale
- Kořen na úrovni MAX stejně vybere levého MIN potomka bez ohledu na cenu uzlů “x”



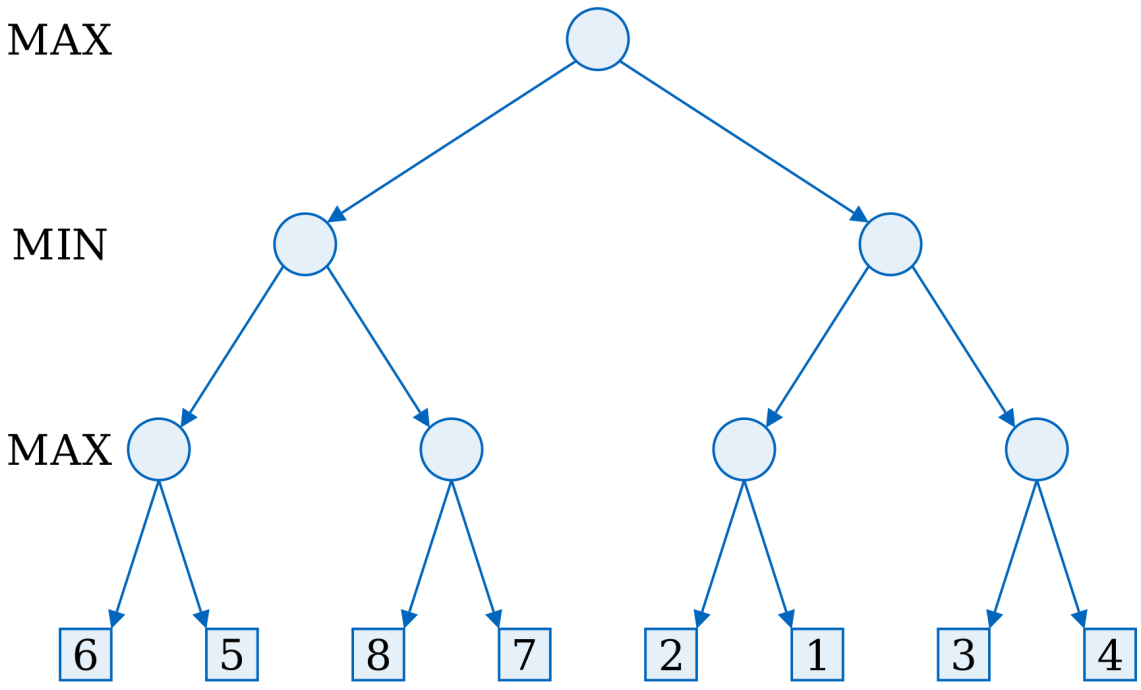
Alfa-Beta Minimax

- Uvažuje následující herní strom, pořadí vyhodnocení uzlů je dáno DFS, na každé úrovni nejdříve zleva
- Které uzly budou (ne)vyhodnoceny v algoritmu Alfa-beta prořezávání?



Alfa-Beta Minimax

- Uvažuje následující herní strom, pořadí vyhodnocení uzlů je dáno DFS, na každé úrovni nejdříve zleva
- Které uzly budou (ne)vyhodnoceny v algoritmu Alfa-beta prořezávání?



Alfa-beta prořezávání

- Kostra Alfa-beta prořezávání, volání stejně jako u klasického Minimaxu

```
1  def alphaBetaMinimax(node, depth, alpha, beta, isMax):
2      if depth == 0 or node.isGoal():
3          return node.h()
4      if isMax:
5          value = float("-inf")
6          for child in node.expand():
7              value = max( value, alphaBetaMinimax(child, depth-1,
                   alpha, beta, False) )
8              if value > beta:
9                  break
10             alpha = max(alpha, value)
11         return value
12     else:
13         value = float("inf")
14         for child in node.expand():
15             value = min( value, alphaBetaMinimax(child, depth-1,
                   alpha, beta, False) )
16             if value < alpha:
17                 break
18             beta = min(beta, value)
19         return value
```

Alfa-Beta Minimax: výpočetní složitost

Minimax vs. Alfa-beta prořezávání

- Klasický minimax prohledává celý herní strom
- Každá akce (uzel) může být výherní
- Uzly se prohledávají bez ohledu na to, jestli už je známé lepší řešení
- **Výsledek obou algoritmů je stejný**

Nejhorší (worst-case)

- Stejná jako u klasického algoritmu Minimax
- Nastane pokud pořadí vyhodnocení akcí (stavů) neumožňuje prořezávání, tj. pokud horší akce (stavy) mají při vyhodnocení přednost

Nejlepší případ (best-case)

- Nejlepší tah (akce) každého hráče je vyhodnocena jako první
- Prakticky se složitost Alfa-beta prořezávání blíží $\mathcal{O}(b^{d/2})$
- tj. jako kdyby byl faktor větení $\sqrt{b}$
- To umožňuje hlubší prohledávání za stejný čas $\rightarrow 2d$

Alfa-beta prořezávání triky

- Pořadí vyhodnocení akcí je velmi důležité, lepší akce (stavy) by měly být vyhodnoceny dříve
- Pro každý uzel lze provést plnou expanzi, vzniklé stavy seřadit, např. dle heuristiky
- Pamatovat si akce, které způsobily výhry v předchozích hrách
- Iterativní DFS
- Minimax + cutoff: ukončit prohledávání na základě heuristiky (pokud jsou uzly horší/lepší než . . .)

Iterativní prohledávání do hloubky

- Iterative depth-first search (IDS)
- Varianta DFS, kdy je omezena hloubka prohledávání
- Poté, co je nalezeno nějaké řešení, spustí se IDS znova s větší hloubkou
- Pokud je limit na provedení tahu, pak se následující běhy IDS mohou ukončit a přesto existuje řešení z předchozího běhu