ZUI

Známé systémy umělé inteligence

Známé systémy umělé inteligence: DeepBlue, Watson, AlphaGo, Eliza, Shakey, DQN

DeepBlue

- Šachová umělá inteligence, která roku 1997 porazila tehdejšího nejlepšího hráče Garryho Kasparova
- Využívalo mnoho nových metod dvouhráčových her, například Negascout, prohledávání stromu bylo paralelní
- Trénováno na hrách grandmasterů
- Využívalo databázi openingů

Watson

- Počítač schopný odpovídat na otázky položené v přirozeném jazyce
- Zvítězil ve vědomostní soutěži Jeopardy! v roce 2011

AlphaGo

- Umělá inteligence, která dokázala v roce 2015 vyhrát proti mistrovi Evropy v Go
- Umělá inteligence pro Go byla další výzvou po DeepBlue, protože počet her, které jsou v Go možné je ještě řádově větší než v šachu
- AlphaGo bylo založeno na Monte Carlo Tree Search algoritmu

Eliza

- Počítačový program z roku 1966, který byl schopný konverzace v přirozeném jazyce
- Založen na transformačních pravidlech, která využívá v případě, že zachytí klíčová slova
- V případě, že klíčová slova nezachytí, provede generickou odpověď

Shakey

- Jezdící robot vyvýjený na Stanfordské univerzitě v letech 1966-1972
- První využití algoritmu A*
- Naprogramován v jazyce Lisp
- Využívá plánovací jazyk

DQN

- Reinforcement learning metoda, která je vhodná pro MDP s velmi vysokým počtem stavů
- Inspiruje se u supervised learningu a k nalezení optimálního řešení využívá SGD (= Stochastic Gradient Descent)

Formální reprezentace problému AI

Formální reprezentace problému AI: problém mnohorukého bandity, MDP, POMDP, hra v rozšířené formě

Prohledávání

- Jedna z nejzákladnějších metod řešení problémů
- Formální reprezentace prohledávání:
  - Množina stavů $s \in S$
  - Množina akcí $a \in A$
  - Aplikací akce a dojde ke změně ze stavu s do stavu s'
- Některé problémy lze formulovat jako sekvenci akcí, které vedou do cílového stavu

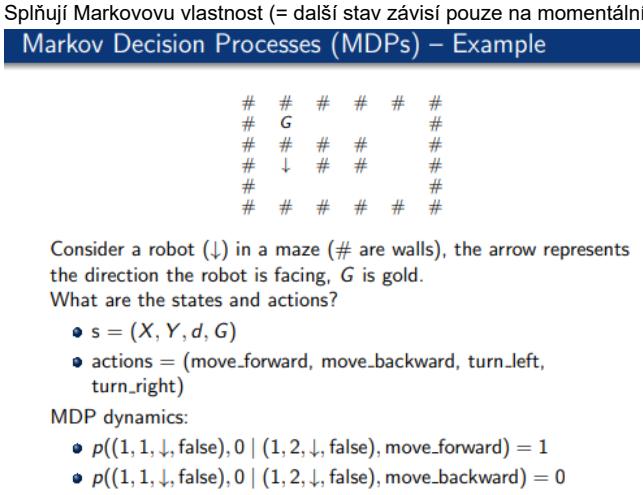
MDP (Markov Decision Process)

- Definuje konečné množiny stavů S, odměn R a akcí A
- Probíhá v diskrétních krocích t
- V každém kroku t provede agent na základě svého aktuálního stavu S_t akci A_t
- V důsledku této akce získá agent odměnu R_{t+1} a dostane se do nového stavu S

- Dynamiku MDP popisuje následující podmíněná pravděpodobnost:

$$p(s', r|s, a) \leftarrow Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

- Pokud hodnotu této podmíněné pravděpodobnosti (v případě MDP 0 nebo 1) vyjádříme pro každou akci odměnu a stav, úplně tím daný MDP popíšeme
- Splňují Markovovu vlastnost (= další stav závisí pouze na momentálním stavu a provedené akci)

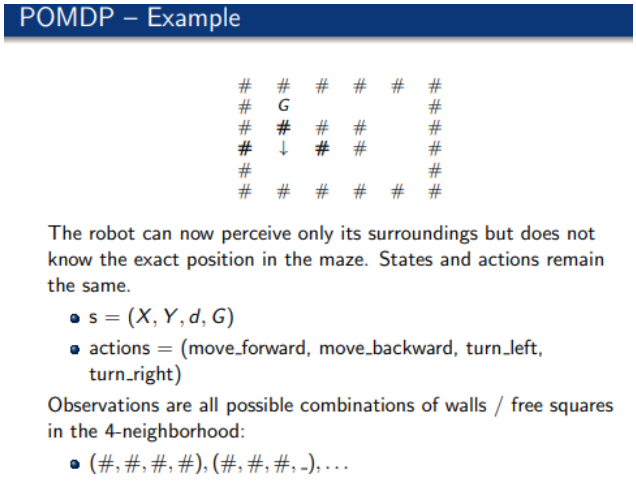


POMDP (Partially Observable Markov Decision Process)

- U POMDP navíc definujeme belief b, což je funkce, která každému stavu přiřazuje věrohodnost (číslo do 0 do 1)
- Dále POMDP definuje konečnou množinu pozorování O
- Na základě pozorování $o \in O$ se mění hodnota beliefu pro každý možný stav s, na počátku je použitý počáteční belief
- Definiční vztah pro update beliefu:

$$b(s') = \mu O(o|s', a) \cdot \sum_{s \in S} Pr(s'|s, a) \cdot b(s)$$

- $O(o|s', a)$ vyjadřuje pravděpodobnost toho, že agent zaznamená pozorování o za předpokladu, že se dostane do stavu s' po vykonání akce a a μ je normalizační konstanta



Extensive-Form Games (EFGs)

- Česky hry v rozšířené formě
- Narozdíl od MDP nebo POMDP jsou konečné
- Často bývají realizovány stromem, kde kořen je začátek hry, hrany jsou akce a listy jsou pak koncové stavy

Problém mnohorukého bandity

- Problém mnohorukého bandity je problém který lze modelovat jedním jediným stavem, ve kterém však lze vykonat několik akcí, s tím že každá akce má jiný reward (ten nemusí být vždy stejný)
- Statisticky pak každou akci a dokážeme vypočítat očekávaný reward, což kde napsat i takhle:

$$q_*(a) = \mathbb{E}[R_t|A_t = a]$$

- V takových problémech řešíme často dilema, zda zkoušet náhodné akce a zjistit více o povaze systému (exploring) nebo s použitím již známých informací maximalizovat reward (exploiting)
- Tomuto problému se říká *Exploration/Exploitation Dillema*
- Takzvanou *greedy akci* definujeme jako akci, která v čase t maximalizuje odhad hodnoty dané akce $Q_t(a)$, což lze popsat pomocí následujícího vztahu:

$$A_t^* = \text{argmax}_a Q_t(a)$$

- Pokud si v daném kroku zvolíme greedy akci, pak dochází k *exploitaci*, pokud nikoliv pak dochází k *exploraci*.

Epsilon-greedy selekce

- Definujeme hyperparameter ϵ, který nám určí pravděpodobnost, že zvolíme náhodnou akci (exploration)
- V případě, že nezvolíme náhodnou akci, tak zvolíme greedy akci (akci a, která maximalizuje $Q(a)$)
- V každém kroku algoritmu pak pro provedenou akci A spočítáme novou hodnotu funkce $Q(A)$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$$

kde $Q(A)$ je odhad hodnoty akce A, $N(A)$ vyjadřuje, kolikrát již byla akce A provedena a R vyjadřuje reward, který byl výsledkem akce A

Nestacionární problém

- Může se stát, že se chování systému v čase mění, což znamená že by se začnou měnit i hodnoty jednotlivých akcí
- V takovém případě není dobrý nápad počítat celkový odhad jako průměr všech dílčích rewardů, ale je lepší dávat vyšší váhu nedávným rewardům
- Pomocí parametru $\alpha \in (0, 1]$ pak spočítáme odhad v kroku $n + 1$ následovně

$$Q_{n+1} = Q_n + \alpha[R_n - Q_n]$$

Upper Confidence Bound (UCB)

- Způsob jak snížit exploraci po vysokém počtu iterací
- Vybíráme vždy akci a, která maximalizuje následující výraz

$$Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}$$

- c je hyperparametr a t je číslo kroku

Prohledávání stavového prostoru

Metody prohledávání stavového prostoru: DFS, BFS, ID-DFS, Dijkstra, A*

Řešení deterministického MDP

- Cílem je najít nejlepší sekvenci akcí, která vede k cílovému stavu
- Metrikou pro určení jaká sekvence akcí je nejlepší jsou typicky akumulované rewardy
- Protože aplikací jedné akce často dostaneme více stavů, musíme si tyto stavy průběžně ukládat do speciálního seznamu (anglicky *fringe*)
- V každé iteraci prohledávání pak podle nějakého pravidla vybereme ze seznamu stav a zkusíme aplikovat další akci v něm
- Podle toho jaký stav zvolíme definujeme několik základních prohledávacích algoritmů
  - **Uniform-cost search** - stavy jsou v seznamu seřazeny obecně podle ohodnocení cesty z počátečního stavu do daného stavu, jde o variantu Dijkstrova algoritmu
  - **Prohledávání do hloubky (DFS)** - všechny stavy, které objevíme při vykonání nějaké akce umístíme na začátku seznamu a vždy pak vybíráme ty stavy, které jsou na začátku seznamu; seznam se tedy chová podobně jako zásobník
  - **Prohledávání do šířky (BFS)** - podobné jako DFS, ale stavy tentokrát umisťujeme na konec; seznam se pak chová jako fronta
- Ohodnocení v uniform cost-search můžeme u MDP realizovat pomocí záporných rewardů (když pak maximalizujeme reward, tak minimalizujeme ohodnocení)

Dijkstrův algoritmus (stručně)

- Algoritmus pro prohledávání grafů, který využívá prioritní frontu k řazení vrcholů, vrcholy jsou tedy seřazeny podle akumulovaného ohodnocení hran v grafu
- Je obecnější nežli uniform-cost search, protože Dijkstrův algoritmus umožňuje najít optimální cestu do libovolného stavu

ID-DFS (Iterative Deepening Depth First Search)

- Jde o kombinaci prohledávání do hloubky a prohledávání do šířky
- Provádíme prohledávání do hloubky, nicméně definujeme maximální délku cesty, kterou chceme při prohledávání procházet
- Pokud nedojdeme do cílového stavu, maximální délku cesty zvýšíme o jedna
- Lze zobecnit, nemusíme používat limit pro délku cesty, ale pro celkové ohodnocení cesty

Heuristika

- Funkce $h : S \to \mathbb{R}^+$, která udává odhad ohodnocení cesty z aktuálního stavu do stavu cílového
- Lze využít pro zvolení dalšího stavu
- Optimální heuristika h^* určuje optimální ohodnocení této cesty
- Při různých problémech je možné používat různé heuristiky, pro nejlepší výsledek by však měly být přípustné
- *Přípustná* heuristika je taková, jejíž ohodnocení bude vždy menší než ohodnocení optimální heuristiky
- *Konzistentní* heuristika je taková, pro kterou platí

  $$h(s) \leq h(s') + c(s, s')$$

  tedy pokud se chceme dostat ze stavu s do stavu s', nikdy se nestane, že by se součet heuristiky a akumulovaného ohodnocení cesty po cestě zmenšoval; $c(s, s')$ je ohodnocení cesty z s do s'
- Lze dokázat, že každá konzistentní heuristika je zároveň i přípustná

A* prohledávání

- Prohledávání, při kterém si zvolíme další stav s na základě akumulovaného ohodnocení z počátečního stavu do stavu s ($g(s)$) a heuristiky v tomto stavu ($h(s)$)

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

- Tato funkce f se potom používá při řazení ve frontě
- A* je optimální (vede k optimálnímu řešení) pokud je použitá heuristika h přípustná
- A *je* optimálně eficientní*, což znamená, že neexistuje jiný optimální algoritmus, který by při prohledávání potřeboval expandovat méně stavů (přidat méně stavů do fronty)

Iterative Deepening A*

- Rozšíření A*, které je inspirované ID-DFS
- Definujeme maximální ohodnocení (*cost cutoff*) c a zavedeme podmínku

  $$g(s) + h(s) > c$$

- Analogicky k ID-DFS, pokud nenajdeme

Jak navrhnout heuristiku

- Nestačí aby heuristika byla přípustná, $\forall s : h(s) = 0$ je taky přípustná heuristika, ale není *informativní* (jde ve výsledku opět o uniform-cost search)
- V některých případech není nutné použít přípustou heuristiku a výrazně tím zrychlit prohledávání, takové prohledávání však někdy nenajde optimální řešení
  - Toho lze dosáhnout například tak, že vezmeme již přípustnou heuristiku a vynásobíme ji $\epsilon > 1$

Posilované učení

Algoritmy posilovaného učení: policy evaluation, policy improvement, policy iteration, value iteration, Q-learning
Metoda, kdy agent provede akci, dostane odpověď od prostředí (reward) a agent si aktualizuje svůj vnitřní stav.

Strategie agenta

- Strategie (anglicky *policy*) π_t v kroku t je funkce, která určuje, jak by se měl agent chovat v nějakém konkrétním stavu
- Lze definovat dvěma způsoby:
  - $\pi_t(a|s)$ popisuje pravděpodobnost, že v kroku t provede agent akci a pokud je ve stavu s (*stochastická strategie*)
  - $\pi_t(s)$ je akce, která bude podle strategie π provedena v kroku t pokud je agent ve stavu s (*deterministická strategie*)
- Cílem strategie je maximalizovat reward pro složité problémy (cílem není maximalizovat reward ihned, ale pro celý problém, dlouhodobě)
- Strategii si můžeme vybrat podle různých kritérií
  - Suma všech rewardů
  - Suma všech rewardů, kde rewardy, které se objevily později mají menší váhu (anglicky *discounted reward*)

- Průměrný reward

Epizodická úloha

- Úloha, u které definujeme koncový stav (v případě, že agent dojde do koncového stavu, dojde k ukončení jedné *epizody*)
- Téměř vždy využívá sumu všech rewardů jako kritérium
- V definicích pro epizodické úlohy se číslo epizody obvykle neuvádí pro zjednodušení značení

Spojitá úloha

- Úloha, která pokračuje donekonečna
- Ve spojité úloze se vždy využívá discounted reward
- Celkový reward v kroku t je tedy vypočten jako

$$G_t = \sum_{k=0}^{\infty} \gamma_k R_{t+k+1}$$

kde $\gamma \in [0, 1]$ je *discount factor* (česky se dá asi přeložit jako diskontní míra)

Ohodnocovací funkce

- Vedle funkce Q definujeme pro strategii π ještě funkci, která ohodnocuje jednotlivé stavy (*state-value funkce*), budeme ji značit v_π
- Obecně se definuje jako očekávaný return (G, které jsme definovali výše), kterého dosáhneme pokud se budeme řídit strategií π
- Na základě této funkce definujeme částečné uspořádání pro strategie:

$$\pi \leq \pi' \Leftrightarrow v_\pi(s) \leq v_{\pi'}(s)$$

- *Optimální strategie* π_* jsou takové, které jsou lepší nebo stejně dobré jako všechny ostatní ve smyslu částečného uspořádání
- *Optimální ohodnocovací funkce pro stavy* $v_\pi(s)$ je pak taková, pro kterou platí

$$\forall s \in \mathcal{S} : v_*(s) = \max_\pi v_\pi(s)$$

- Analogicky definujeme i *optimální ohodnocovací funkci pro akce* (*action-value funkce*)

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{A} : q_*(s, a) = \max_\pi q_\pi(s, a)$$

Bellmannova rovnice

- Pro strategii π můžeme v průběhu algoritmu rekurzivně vypočítat hodnotu ohodnocovací funkce

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')$$

- Takto pak rovnice vypadá pro optimální hodnoty obou ohodnocovacích funkcí

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

- Řešení rovnice je polynomiální vzhledem k počtu stavů, nicméně počet stavů ve stavovém prostoru může někdy být extrémně velký
- γ je hyperparametr

Policy Evaluation algoritmus

- Vychází z bellmanovy rovnice pro state-value funkci
- Pro to abychom mohli algoritmus provést už musíme znát nějakou strategii π
- Na začátku algoritmu je state-value funkce inicializována pro všechny stavy na nulu
- State-value funkce je potom iterativně přepočítávána pomocí bellmanovy rovnice
- V každé iteraci se vypočítá maximální odchylka state-value funkce od předchozí hodnoty, Δ
- Algoritmus se opakuje dokud není Δ menší než předem stanovená hranice θ
- Výpočet state-value funkce je prováděn podle následujícího vztahu

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

Policy Improvement

- Po přepočítání value funkce v_π se může ukázat, že strategie π není optimální, protože existuje stav a a akce s takové, že

$$q_\pi(s, a) > v_\pi(s)$$

- V takovou chvíli lze strategii zlepšit přiřazením $\pi(s) = a$

Policy Iteration algoritmus

- Rozšíření policy evaluation o zlepšování strategie π dokud se zlepšovat nepřestane
- Nejprve dojde k inicializaci state-value funkce a strategie pro všechny stavy
- Pak dochází k iterativnímu zlepšování state-value funkce pomocí následující rovnice

$$V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$$

- Třetí krok je zlepšení strategie na základě nových hodnot state-value funkce

$$\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

- Pokud dojde k nějakému zlepšení strategie, algorimus se začne opět přepočítávat state-value funkci, pokud ne tak končí

Value Iteration algoritmus

- Má podobnou strukturu jako policy-evaluation algoritmus, ale narozdíl od něj nevyžaduje na vstupu existující strategii, ale naopak iterativně strategii vytváří
- Protože není k dispozici strategie, je vždy pro výpočet state-value funkce vybrána akce, která bude hodnotu state-value funkce maximalizovat:

$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

- Výsledná strategie se pak spočítá stejně jako u policy iteration algoritmu a algoritmus končí

Q-Learning algoritmus

- Narozdíl od předchozích tří metod pracuje s action-value funkcí
- Hodnoty action-value funkce se zlepšují tak, že se několikrát za sebou hledá optimální cesta z počátečního do koncového stavu a přitom jsou hodnoty action-value funkce přepočítávány
- V každém kroku jedné epizody dojde ke zvolení akce A ve stavu S na základě action-value funkce Q
- Poté dojde k pozorování rewardu R a nového stavu S'
- Na základě těchto hodnot je action value funkce přepočítána podle následujícího vztahu

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \text{max}_a Q(S', A) - Q(S, A)]$$

- α a γ jsou hyperparametry
- Existuje i varianta zvaná *Double Q-Learning*, která upravuje hodnoty dvou action-value funkcí, které v každé iteraci prohazuje

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha[R + Q_2(S', \text{argmax}_a Q_1(S', a)) - Q(S, A)]$$

Dvouhráčové hry

Algoritmy pro řešení her dvou hráčů: minimax, alpha-beta prořezávání, negamax, negascout, MCTS

Formální definice dvouhráčových her

- $P = \{1, 2\}$ je množina hráčů
- H je množina historií akcí
- A je konečná množina akcí
- $A(h)$ je množina akcí, které jsou dostupné v bodě $h \in H$
- $Z \subseteq H$ je množina historií, které vedou ke konci hry
- $u : Z \rightarrow \mathbb{R}$ je funkce, která ohodnocuje výsledek hry

Minimax algoritmus

- Protože každou hru lze reprezentovat stromem, řešení dvouhráčových her probíhá obvykle jako procházení tímto stromem
- Algoritmus začíná v kořeni stromu, což je začátek hry, všechny hrany, které vedou do dalších vrcholů jsou akce, které může hráč, který je na řadě, aktuálně vykonat

- Algoritmus má pak v nějakém vrcholu $h \in H$ následující chování
  - Pokud $h \in Z$, pak se vrací hodnota $u(h)$
  - Pokud nejde o list tak se volá stejná funkce pro všechny child vrcholy, pokud je v daném vrcholu na řadě první hráč vrací se maximum všech vrácených hodnot, v opačném případě se vrací minimum všech vrácených hodnot

Alpha-beta prořezávání

- Často není nutné procházet celý strom, je možné tedy minimax algoritmus rozšířit tak, že jsou některé vrcholy přeskočeny, pro to si musíme v každém vrcholu navíc držet hodnoty α_h a β_h, jejich počáteční hodnoty (hodnoty pro kořen) jsou vstupními parametry algoritmu
- V případě rozhodovacích vrcholů pak rozlišujeme v závislosti na tom, který hráč je na řadě, následující chování
  - V případě prvního hráče se postupně prochází všechny child vrcholy a vracená hodnota v_h je přepočítána jako maximum sebe sama a vrácené hodnoty child vrcholu. α_h je pak vypočteno jako $\max(\alpha_h, v_h)$. Pokud je splněna podmínka $\beta_h \leq \alpha_h$, pak je iterace přes child vrcholy přerušena a v_h je vráceno
  - V případě druhého hráče je postup podobný, ale v_h je vypočteno jako minimum vrácené hodnoty a sebe sama, dochází analogicky k přepočtení β_h a podmínkou pro přerušení je $\alpha_h \leq \beta_h$
- Nejbezpečnější je vždy nastavit počáteční hodnoty na $-\infty$ pro alfa a na ∞ pro beta
- Inicializace na jiné hodnoty může vést k výrazně rychlejšímu řešení, ale někdy také k nesprávnému řešení

Negamax

- Algoritmus, který funguje naprosto stejně jako alpha-beta prořezávání, ale má kratší zápis
- Pro všechny akce a_h v aktuálním vrcholu se provede
  - $v_h = \max(v_h, -\text{search}(a_h, -\beta_h, -\alpha_h))$
  - $\alpha_h = \max(\alpha_h, v_h)$
  - if $\beta_h \leq \alpha_h$ then break

NegaScout

- Také Principal variation search
- Vylepšení negamaxu
- Vychází z předpokladu, že akce jsou optimálně seřazeny (například podle nějaké heuristiky)
- První akce je vždy vyhodnocena s maximálním intervalem, všechny další již s minimálním intervalem
- Podrobný pseudokód je zde (převzato z přednášky):

For a decision point h (initial values for bounds α_h and β_h are passed as parameters):

1. if h is a leaf ($h \in Z$), then return $u(z)$,
2. $\beta'_h = \beta_h$, $\alpha'_h = \alpha_h$
3. if h is a decision node and $h \in H_1$ then for each $a_h \in A(h)$:
   1. $v_h = \text{search}(a_h, \alpha_h, \beta'_h)$
   2. if $((\alpha_h < v_h < \beta_h)$ and (h is not the first child))
      - $v_h = \text{search}(a_h, v_h, \beta_h)$
   3. $\alpha_h = \max(\alpha_h, v_h)$
   4. if $\beta_h \leq \alpha_h$ then break
   5. $\beta'_h = \alpha_h + 1$
   6. return α_h
4. ... (similarly for player 2) ...

For a decision point h (initial values for bounds α_h and β_h are passed as parameters):

1. ...
2. if h is a decision node and $h \in H_2$ then for each $a_h \in A(h)$:
   1. $v_h = \text{search}(a_h, \alpha'_h, \beta_h)$
   2. if $((\alpha_h < v_h < \beta_h)$ and (h is not the first child))
      - $v_h = \text{search}(a_h, \alpha_h, v_h)$
   3. $\beta_h = \min(\beta_h, v_h)$
   4. if $\beta_h \leq \alpha_h$ then break
   5. $\alpha'_h = \beta_h - 1$
   6. return β_h

Náhodnost ve dvouhráčových hrách

- Pokud ve dvouhráčových hrách může mít jedna akce několik různých výsledků s různou pravděpodobností, vypočtou se hodnoty pro všechny výsledné vrcholy a výsledek bude jejich váženým průměrem

Monte Carlo Tree Search (MCTS)

- Místo jakési evaluační funkce je tato metoda založena na náhodných simulacích hry
- V algoritmu se dokola opakují čtyři fáze *selekce*, *expanze*, *simulace* a *zpětná propagace*

Selekce v MCTS

- Jde opět o dilema explorace a exploitace, nejpopulárnějším řešením je speciální verze Upper Confidence Bound, které se říká *UCT*
- V případě UCT se akce se volí na základě následující podmínky

$$A_t(s) = \text{argmax}_a \left[Q_t(s,a) + c\sqrt{\frac{\log N_t(s)}{N_t(s,a)}} \right]$$

Expanze v MCTS

- Expanze se provádí pomocí *progresivního rozšiřování*
- Na počátku se používá jen malé množství akcí
- Poté co byly všechny akce přidány a dobře prozkoumány jsou přidány další
- Funguje i pro nekonečný počet akcí

Simulace v MCTS

- Tři způsoby jak simulovat:
  - Náhodné akce
  - Akce na základě předem daných pravidel
  - Akce na základě naučených znalostí

Zpětná propagace v MCTS

- Statistiky , které se využívají při selekci, jsou po simulaci dopočítávány
- Tyto statistiky zahrnují hlavně $N(s)$, $N(s,a)$ a $Q(s,a)$

Strukturovaná reprezentace znalostí

Strukturovaná reprezentace znalostí: CSP, Scheduling, Situtation calculus, STRIPS

CSP (Constraint Satisfaction Problem)

- Problémy, ve kterých je cílem najít přiřazení hodnot ke proměnným tak, aby byla splněna všechna definovaná omezení
- U CSP definujeme tři množiny
  - *Proměnné* x_i
  - *Domény* D_i pro každou proměnnou
  - *Omezení* c_i
- Omezení jsou definována jako podmnožina proměnných, které je přiřazeno několik n-tic, které obsahují hodnoty domén
- CSP lze reprezentovat prohledávacím stromem, kde v každém vrcholu se nachází nějaké přiřazení proměnných (některé proměnné nemusí mít přiřazeného nic) a v child vrcholech se pak nachází všechna přiřazení, kde je o jedna více přiřazených proměnných
- Standardně se však CSP reprezentuje tak, že všechna omezení jsou přepsána na *binární omezení*
  - Omezení, které se týká k proměnných lze vždy přepsat na k binárních omezení
- V případě, že definujeme jen binární omezení, můžeme vizualizovat CSP pomocí grafu, kde vrcholy jsou proměnné a hrany jsou omezení
- *Forward checking* - odstranění hodnot proměnných, které nesplňují omeezní pokaždé, když nějaké proměnné přiřadíme hodnotu

Prohledávání CSP

- Algoritmus si udržuje všechny možné hodnoty pro všechny proměnné
- V algoritmu se zvolí vždy jedna proměnná a iteruje se přes všechny hodnoty v její příslušné doméně. Pokud se přiřazení podaří pokračuje se s další proměnnou
- Pokud mají všechny proměnné přiřazenou jednu hodnotu, tak se tato přiřazení vrací jako výsledek
- Pokud po forward checkingu zůstane doména nějaké proměnné prázdná, tak se algoritmus vrací

AC-3 algoritmus

- Algoritmus, který slouží ke kontrole konzistence hran v grafu
- Udržuje si frontu Q všech hran, u kterých je potřeba zkontrolovat konzistenci
- Pokud byla při kontrole konzistence dané hrany upravena doména jejího výchozího vrcholu, přidají se do fronty všechny hrany, jejichž cílový vrchol je počáteční vrchol této hrany
- AC-3 stále negarantuje řešení

Constraint Optimization Problems

- Rozšíření CSP, které může kromě *hard constraints*, které jsou nutné pro nalezení řešení, definovat i *soft constraints*, což může být například nějaká funkce, kterou chceme optimalizovat
- Můžeme také hledat více než jedno řešení a přidat ořezávání (podobně jako alpha beta pruning)

Scheduling

- Lze reprezentovat pomocí CSP (víc jsem k tomu nenašel...)

Situační kalkulus

- Způsob jak reprezentovat změny ve světě pomocí logiky prvního řádu (viz LGR)
- Predikáty mohou být
  - rigidní - nezávislé na situaci
  - fluentní - závisí na situaci
- Situace jsou provázány funkcí result - pokud je vykonána akce a v situaci s, pak je výsledkem situace s'
- Definujeme predikáty pro agenta a pro objekty v modelu
- To, jak se mění fluentní atributy lze popisovat pomocí axiomů, kterých je asymptoticky $O(f \cdot a)$, kde f je počet fluentních predikátů a a je počet akcí, takže uplatňování axiomů v každém kroku je výpočetně náročné (*frame problem*)

Logické plánování

- Cílem plánování je fluidní predikát, který popisuje cílový stav
- Chceme ověřit zda zda je cílový predikát sémantickým důsledkem počátečních znalostí, toho lze dosáhnout například rezoluční metodou

Problémy plánování

- Problémy, kde je cílem najít plán, což je sekvence tahů, které transformují počáteční stav do cílového
- Plánování může mít několik podúkolů:
  - Zjistit jestli existuje řešení

    - Najít jakékoli řešení
    - Najít optimální řešení
    - Najít řešení dostatečně rychle
    - Najít řešení, které splňuje podmínku $\aleph$
- Součástí plánování jsou modely, jazyky a algoritmy
- Řešení problémů plánování jsou cesty z počátečního do koncového stavu
- Akce je často nutné reprezentovat kompaktněji než vztahy (z důvodu velikosti stavového prostoru)

Plánovací jazyky

- Deklarativní jazyky, které se používají k reprezentaci problémů plánování
- Dvě role: *specifikace* a *výpočet*
- Jedním z plánovacích jazyků je jazyk STRIPS

STRIPS

- "Formální definice":
    - P je množina *atomů* (booleovské proměnné)
    - $I \subseteq P$ je *počáteční situace*
    - $G \subseteq P$ je *cílová situace*
    - A je konečná množina akcí a, které jsou specifikované pomocí $\mathrm{pre}(a)$, $\mathrm{add}(a)$, $\mathrm{del}(a)$
- Akci a lze aplikovat ve stavu s pokud platí $\mathrm{pre}(a) \subseteq s$
- Po aplikaci akce a ve stavu s vzniká nový stav $s = (s \backslash \mathrm{del}(a)) \cup \mathrm{add}(a)$
- STRIPS pouze slouží k čitelnější reprezentaci problémů plánování
- K řešení problémů, které jsou napsané v STRIPS lze velmi dobře použít A*
    - Heuristiku můžeme definovat jako $h(s) = |G \backslash s|$
- *Abstrakce* - řešení menšího problému, například odstranění predikátu
- *Relaxace* - řešení problému, který má méně omezení
    - Provádí se prostřednictvím heuristik (například straight-line heuristika, kde ignorujeme fakt, že musíme zůstat na cestě)
    - Specificky pro STRIPS můžeme provést takovou relaxaci, že budeme ignorovat efekty akcí, které odstraňují akce
- Pro akci $a = \langle \mathrm{pre}(a), \mathrm{add}(a), \mathrm{del}(a) \rangle$ definujeme relaxovanou akci $a^+ = \langle \mathrm{pre}(a), \mathrm{add}(a), \emptyset \rangle$

Příklad - Sokoban

- Reprezentace stavů

```
positions a1, ..., a6,...
                 f1, ..., f6
box_at(P), free(P)
player_at(P)
adjacent(P1, P2)
adjacent(P1, P2)
```

- Akce:

```
move(X, Y):
        pre: player_at(X)
                adjacent(X, Y)
                free(Y)
        add: player_at(Y)
        del: player_at(X)
box_at(P), free(P)
player_at(P)
adjacent(P1, P2)
adjacent(P1, P2)
```

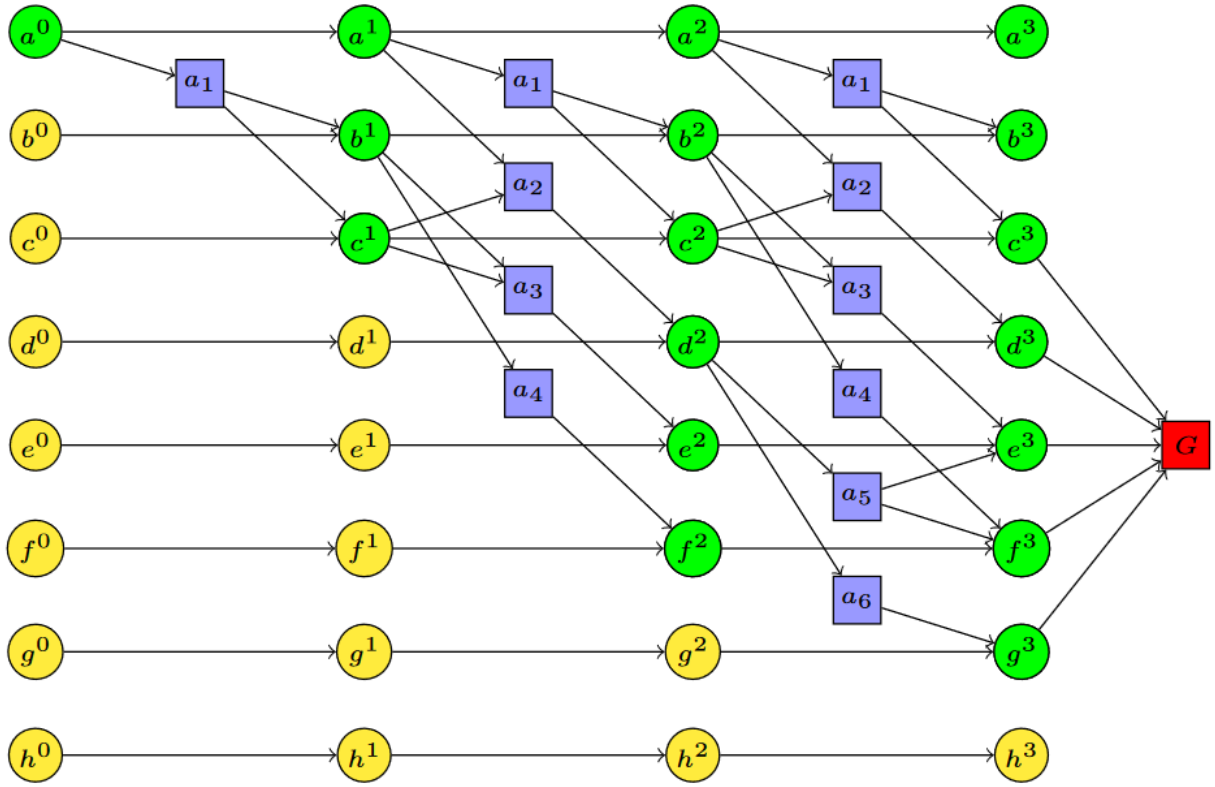
Relaxovaný plánovací graf

- Graf dosažitelnosti, kde se na začátku nacházejí atomy počátečního stavu
- Za sebou se pak nacházejí střídavě akce a stavy

$$A_i = \{a \in A | \text{pre}(a) \subseteq P_i\}$$

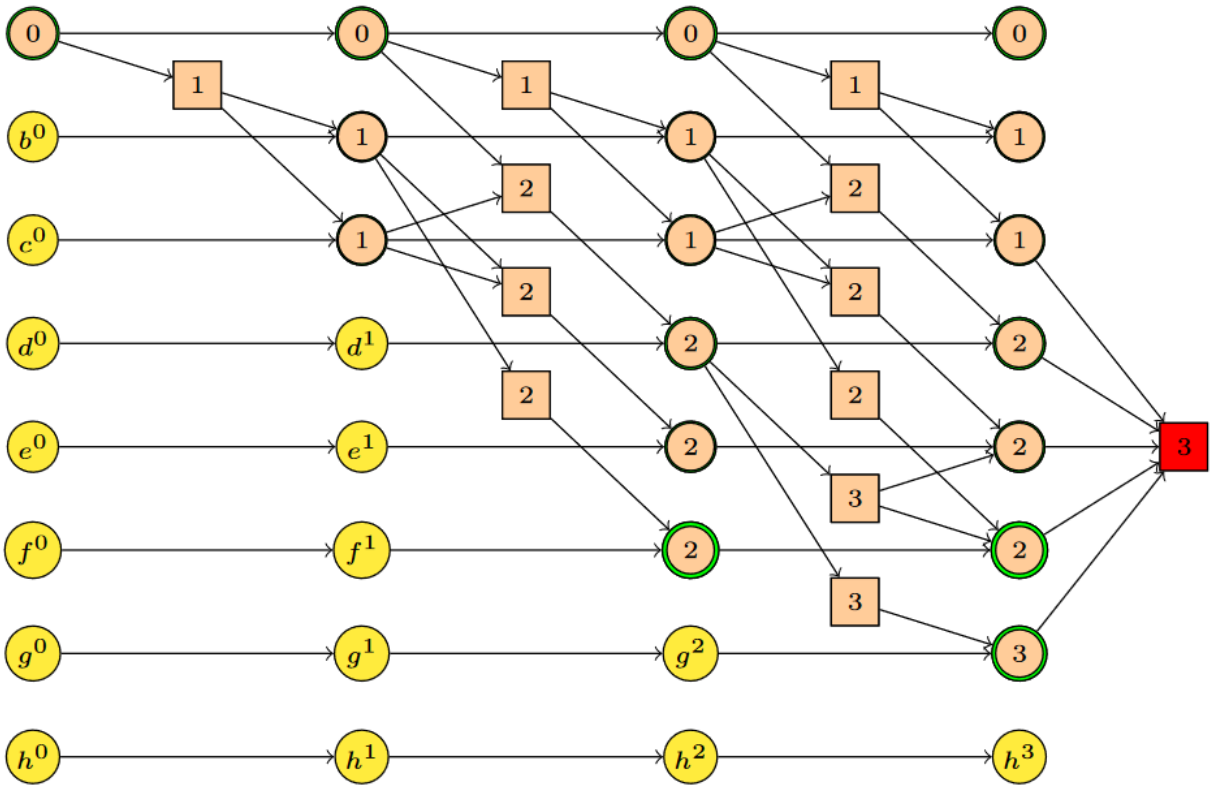
$$P_{i+1} = P_i \cup \{p \in \text{add}(a) | a \in A_i\}$$

- Konstrukce grafu je ukončena ve chvíli kdy $G \in P_i$
- Ilustrační obrázek:



- Stejný graf se dá použít k výpočtu optimální heuristiky h_{max}
  - Vrchol, který náleží počátečnímu stavu se inicializuje na nulu
  - Všechny vrcholy s akcemi jsou pak rovny maximu hodnot z předpokladů dané akce
  - Do vrcholů se stavy (propozicemi) se pak zapíše nejlevnější akce, která do daného stavu vedla

- Ilustrační obrázek:



Neurčitost v AI

Neurčitost v AI: maximalizace očekávané utility, Bayesovo pravidlo, Bayesovské sítě

Racionální rozhodování

- Součástí racionálního rozhodování jsou dvě složky, pravděpodobnost a utilita

Důležité vztahy z pravděpodobnosti

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

$$P(a, b) = P(a|b)P(b)$$

- Bayesovo pravidlo:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

Podmíněná nezávislost

- Proměnné X, Y jsou nezávislé za podmínky Z, pokud platí jakákoli z následujících rovnic:

$$P(X|Y, Z) = P(X|Z)$$

$$P(Y|X, Z) = P(Y|Z)$$

$$P(X, Y|Z) = P(X|Z)P(Y|Z)$$

Bayesovská síť

- Graf (DAG), kde vrcholy jsou náhodné proměnné a šipky popisují pravděpodobnostní závislosti mezi těmito proměnnými, platí pak ještě následující vztahy
  - Vrchol je podmíněně nezavislý na svých předchůdcích za podmínky svých rodičů
  - Vrchol je podmíněně nezávislý na všech vrcholech, které nejsou potomky, za podmínky svých rodičů
  - Vrchol je podmíněně nezávislý na všech ostatních vrcholech za podmínky svých dětí, rodičů a rodičů svých dětí
- Odvozování:

- Chceme spočítat aposteriorní pravděpodobnost množiny *dotazovacích proměnných* X za předpokladu pozorovaných přiřazení proměnných E k hodnotám $\mathbf{e}$
- Označíme množinu náhodných proměnných Y, které nepatří ani do E ani do X, pak platí
- $$P(X|\mathbf{e}) = \frac{P(X,\mathbf{e})}{P(\mathbf{e})} = \alpha \sum_{y \in Y} P(X, \mathbf{e}, y)$$

- U bayesovských sítí pak sdruženou pravděpodobnost můžeme spočítat jako
- $$\prod_i P(X_i|\text{Parents}(X_i))$$

- Výpočet se dá vizualizovat pomocí stromu. Protože se některé výpočty opakují vícekrát, lze využít dynamické programování

Speciální případy Bayesovských sítí

- *Naivní Bayesovský klasifikátor* je síť kde se nachází nepozorovatelná třída Y, ze které vycházejí šipky do jednotlivých proměnných které třídu ovlivňují (použití například u spam filteru)
- *Hidden Markov model* je síť, kde se v čase t nachází skutečná informace H_t a nepřesné měření této informace E_t, přičemž E_t závisí na H_t a H_{t+1} závisí na H_t

Řešení POMDP

Řešení POMDP: PBVI, HSVI

Ohodnocovací funkce pro POMDP

- State-value funkce u POMDP přiřazuje nějakému beliefu hodnotu
- Optimální state-value funkce je definovaná Bellmanovou rovnicí pro POMDP

$$v_*(b) = \max_a \int p(b', r|b, a)[r + \gamma v_*(b')]db'$$

- Tato rovnice je však nepraktická pro výpočet
- Pro každou akci můžeme definovat tzv. α-vektor, což je value funkce pro nějakou specifickou akci
  - Funkce je definovaná na simplexu všech stavů (všechny možné hodnoty beliefu - hodnota od 0 do 1 pro každý stav)
  - Pro dva stavy lze znázornit ve 2D jako interval od 0 do 1

Value iteration pro POMDP

- Přepočet state-value funkce je proveden podle následující rovnice

$$v_{t+1}(b) = \max_a \left\{ \sum_{o \in O} \max_{\alpha' \in v_t} \left[\sum_{r,s,s'} \mu p(s', r|s, a) b(s) O(o|s', a)(r + \gamma \alpha'(s')) \right] \right\}$$

- Konstrukce α-vektorů z v_{t+1}
  - Známe α-vektory v kroku t
  - Na základě pozorování a předchozí hodnoty state-value funkce zvolíme nový α-vektor
- Při plném value iteration algoritmu je potřeba počítat velké množství α-vektorů, což je neefektivní, nemusíme však počítat všechny α-vektory, vzhledem k tomu, že mnoho z nich není relevantní k řešení problému

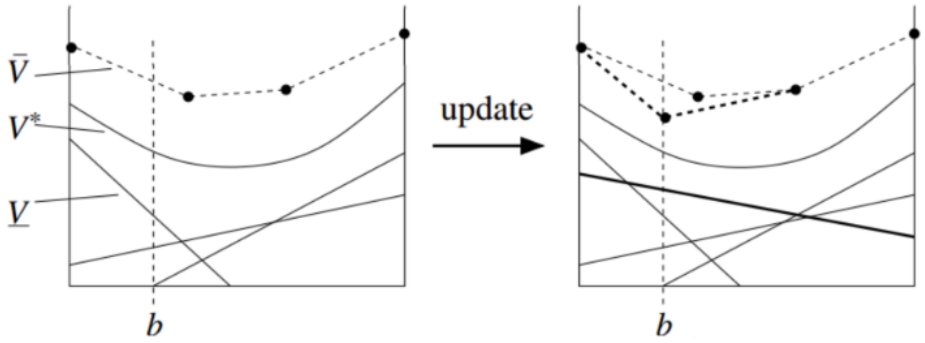
Point-based Value Iteration (PBVI)

- Definujeme množinu $\mathcal{B}$, která obsahuje specifické hodnoty beliefu
- Při výpočtu si uchováváme pouze ty α-vektory, které jsou optimální pro belief pointy z množiny $\mathcal{B}$
- Při vyhodnocování beliefu pomocí Bellmanovy rovnice také používáme pouze tyto belief pointy, což algoritmus zásadně zrychlí
- Tento algoritmus již nemá exponenciální výpočetní náročnost, ale zároveň nedává žádnou informaci o tom, jak daleko je aktuální řešení od optima

Heuristic Search Value Iteration (HSVI)

- V tomto algoritmu aproximujeme state-value funkci pomocí dvou jiných funkcí, které udávají horní a dolní mez
- Kroky
  - Nejprve jsou horní a dolní mez inicializovány
  - Dále je vybrána akce k prozkoumání nejlepšího prostoru belief pointů
  - Pro vybrané belief pointy jsou pak vypočítány hodnoty obou aproximačních funkcí
- Inicializace funkcí:

    - Spodní mez - výběr libovolné akce
    - Horní mez - řešení MDP pro každý stav
- Update funkcí:
    - Spodní mez - update v belief pointech
    - Horní mez - spodní konvexní obálka množiny bodů pro horní mez a potom update v belief pointech
- Ilustrační obrázek:



- Horní mez se využívá k prohledávání, spodní mez udržuje nejlepší nalezené řešení