1. ‚Historie AI

Přednáška 1 20.2.2024

1.1 Dávná historie

- 15. století – **mechanický rytíř** (Leonardo da Vinci) – brnění pohybující se pomocí kabelů atd.
- 18. století – **mechanický Turek** – postavička, která hrála šachy (uvnitř seděl člověk, který vše ovládal)
- 1914 – první stroj, který skutečně sám posouval figurky (uměl jen pár kombinací)
- 1921 – R.U.R. (K. Čapek) – původ slova „robot“

40. léta – první články o neuronových sítích

50. léta

- 1950 – první šachový program (Alan Turing)
  - **TURINGŮV TEST** – „Jak určit, že je stroj inteligentní?“ (testuje kvalitu AI)
    - Stroj je inteligentní, pokud člověk, dostává odpovědi na libovolné otázky od dvou stran = AI a člověka, nedokáže rozlišit, která strana je AI a která člověk.
- 1956 – Logic Theorist – program pro dokazování matematických teorémů
  - **Dartmouth workshop** – v žádosti o dotaci projektu poprvé použit termín „*artificial intelligence“*

60. léta – první článek o problému Superinteligence

- 1966 – **ELIZA** – první populární chatbot
    - princip pattern matchingu (identifikace klíčových slov + transformační pravidla), simuloval konverzaci s psychologem
  - **SHAKEY** – první mobilní robot
    - vytvořen jazyk Lisp (vhodný pro plánování)
    - plánování pomocí STRIPS algoritmu
    - poprvé představen A* algoritmus

70. léta

- 1972 – WATBOT-1, humanoidní japonský robot
  - MYCIN – systém identifikace příčiny infekcí
    - větší úspěšnost než lékaři, ale nepoužíval se – lidé strojům nevěří, přestože mají lepší výsledky než lidé

První AI zima (1974 – 1980) – pokles zájmu a financování AI – nedostavily se očekávané výsledky

80. léta

- 1980 – Chinese room argument – „Turningův test není dostatečný pro otestování skutečné inteligence“
    - silná AI (počítač je mozek, který skutečně chápe) X slabá AI (jen se tváří, jako že rozumí)
- 1986 – Samořiditelná dodávka (na prázdných ulicích)
- 1988 – Bayesovské sítě – pravděpodobnostní model s grafovou reprezentací

Druhá AI zima (1987 – 1993) – mnoho failed start-upů, pokles produkce AI HW (kvůli silnějším obecným HW)

90. léta – důraz na rychlé, reaktivní chování AI

- 1997 – **DeepBlue** poráží Kasparova (programováno přímo proti němu, αβ-pruning)
- 1999 – AIBO – robotický pes (první masově vyráběný robotický systém)

2000s – komerčně úspěšní roboti

- 2005 – DARPA Grand Challenge – poprvé úspěšné týmy (auto ujelo 130 mil v poušti bez zásahu člověka)

2010s – deep learning a NN (výpočty na grafických kartách), bezpečnost a férovost AI (předpojatost AI modelů – genderové stereotypy, sexismus, rasismus atd.)

- 2011 – Watson, Siri – jazykové modely
- 2015 – **DQN v Atari hrách** (DL + RL) – naučil se hrát Atari hry i bez znalosti pravidel
  - **AlphaGo** (MCTS + DL) – hra Go
- 2017 – DeepStack (CFR + DL) – poker
- 2018 – jazykový model BERT a další (hodně dat + DL)
- 2019 – **AlphaStar** (Game theory + RL + DL) – hra StarCraft II

2020s

- 2020 – AlphaFold – predikce 3D tvaru proteinu na základě genetické sekvence (dobré pro tvorbu léku a lepší porozumění biologickým procesům)
- 2021 – Dall-E – generování obrázků z textových popisů
- 2022 – ChatGPT
- 2024 – SORA – generování videí z textu (jeden z kroků k porozumění fyzice světa)

2. Formalizace AI úloh a prohledávání

Přednáška 2 27.2.2024

2.1 Prohledávání v AI

° hledání = jedna z nezákladnějších a nejuniverzálnějších metod řešení úloh

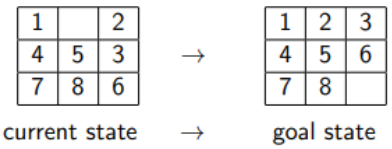
FORMÁLNÍ REPREZENTACE

Umožňuje systematické hledání řešení

° S … stavy – formální reprezentace každé možné situace
  – musí obsahovat všechny potřebné informace pro určení možných akcí, v jakém stavu po provedení akce skončíme, nebo zda bylo dosaženo cíle

° A …akce – jak jsou stavy měněny pomocí algoritmu (agenta)

Příklad 1: 8-puzzle



Možné reprezentace stavů:

- hodnoty políček [1, _, 2, 4, 5, 3, 7, 8, 6] – vhodnější (méně výpočetně náročné, teoreticky více náročné na paměť)
- pozice čísel [1, 3, 6, 4, 5, 9, 7, 8] – výpočetně náročnější – musíme vždy vyhledat prázdné pole

Reprezentace akcí – akci chápeme jako pohyb prázdného políčka

Příklad 2: šachy

Možná reprezentace:

- pozice jednotlivých figurek + doplňující informace (zda se pohnul král nebo věž – kvůli rošádě, opakování pozic – pokud se pozice 3x zopakuje, nastává remíza -> stav může odpovídat historii všech tahů)

Mnoho AI úloh lze formulovat **jako hledání sekvence akcí, které vedou k cílovému stavu**

Hledáme nejlepší možnou sekvenci (na základě daného kritéria)

° minimalizace počtu akcí

° minimalizace celkové ceny/maximalizace odměny (každá akce může mít přiřazenou nějakou hodnotu)

Musíme znát pravidla prostředí – jaké akce můžeme udělat, do jakého stavu se tím dostaneme – tzn. musíme mít formální model nebo přístup k simulátoru

2.2 ZÁKLADNÍ MODELY PRO SEKVENČNÍ ROZHODOVÁNÍ

° **Markovovy rozhodovací procesy** (MDPs)
  - 1 agent, dokonale pozorovatelné prostředí (pokud existuje nejistota, musíme o ní vědět)
  - např. bludiště

° **Částečně pozorovatelné Markovovy rozhodovací procesy** (POMDPs)
  - 1 agent, částečně pozorovatelné prostředí (např. agent neví, kde se nachází)

° (Neperfektně informované) hry v rozvinutém tvaru / (Imperfect Information) **Extensive-Form Games** (EFGs)
  - n hrajících agentů (každý optimalizuje pro vlastní cíl), perfektně (částečně) pozorovatelné prostředí, konečný horizont (hra musí skončit)
  - např. robotický fotbal / šachy

° (Částečně pozorovatelné) **Stochastické hry** (POSGs)
  - n hrajících agentů (každý má vlastní cíl), perfektně (částečně) pozorovatelné prostředí, nekonečný horizont (není dáno, že hra po určitém počtu tahů skončí)

PROČ POTŘEBUJEME OBECNOU FORMÁLNÍ DEFINCI?

- Pokud vymyslíme algoritmus pro MDPs (POMDPs, EFGs, …), bude umět řešit celou třídu problémů
- Formalizace pomáhá se nad problémem zamyslet
- MDPs a POMDPs jsou už dobře prozkoumané – můžeme použít již existující algoritmy
- Jasná definice vstupů a výstupů algoritmu

MDPs

Zatím předpokládáme deterministické prostředí – žádné

° Konečné množiny S – stavy, R – odměny, A - akce

° Diskrétní kroky t = 0, 1, …
  - v každém kroku obdrží aktuální stav S_t, vybere akci z A_t, obdrží odměnu R_{t+1} a přesune se do stavu S_{t+1}

- Odměny a stavy jsou generovány na základě dynamiky daného prostředí **í**

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

  - Pravděpodobnost, se kterou se po zahrání akce dostaneme z původního stavu do dalšího a jakou odměnu tím získáme (pst se nemění) – deterministickém prostředí vždy 0 nebo 1

Další stav závisí pouze na aktuálním a na provedené akci (Markovova vlastnost)

Příklad: bludiště – nejlepší cesta ke zlatu

- Robot (→ šipka odpovídá směru ve kterém robot stojí), zdi (#), zlato (G)
- stavy S = (X, Y, d, G)
  - souřadnice X a Y
  - směr d
  - zda jsme zlato sebrali nebo ne G
- Mapa je fixní, proto ji nemusíme mít v paměti
- akce A = (move_forward, move_backward, turn_left, turn_right)

Dynamika MDP bludiště:

- p((1, 1, down, false), 0 | (1, 2, down, false), move_forward) = 1 … 100% pravděpodobnost, že pokud v pozici (1, 2, down, fasle) provedeme pohyb vpřed, dostaneme se do pozice (1, 1, down, false,) a získáme odměnu 0
- p((1, 1, down, false), 0 | (1, 2, down, false), move_backward) = 0 … naopak s 0% pravděpodobností se do dané pozice ze stejného stavu dostaneme při pohybu zpět

POMDPs

- Konečné množiny stavů S, odměn R a akcí A
- **Nemá definovaný horizont**
- Agent nemůže dokonale pozorovat okolí – ***belief*** – pravděpodobnostní distribuce nad stavy, pravděpodobnost, že je agent v daném stavu při daném pozorování (např. s 50% si myslí, že je ve stavu 1 a s 50%, že ve stavu 3) – ALE **musí vědět, jaké stavy existují**
  - V každém kole pozorování z konečné množiny O, na základě, kterého aktualizuje svůj belief

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

    - μ … normalizační konstanta
    - $O(o|s', a)$ … pravděpodobnost pozorování o za předpokladu, že jsme ve stavu s‘ a byla provedena akce a
    - $\mathrm{P}(s'|s, a)$…pst, že jsme se akcí a ve stavu s dostali do stavu s‘
    - $b(s)$…belief stavu s

Příklad: bludiště

- agent vidí jen 4 okolní políčka
- robot (→…šipka odpovídá směru ve kterém robot stojí), zdi (#), zlato (G)
- stavy S = (X, Y, d, G) … souřadnice X a Y, směr d, zda jsme zlato sebrali nebo ne G
  - Mapa je fixní, proto ji nemusíme mít v paměti
- akce A = (move_forward, move_backward, turn_left, turn_right)
- množina pozorování (#, #, #, #), (#, #, #, _),…

EFGs

- **více hráčů** – v každém stavu je na tahu jen jeden hráč
- typická reprezentace = game-trees:
  - konečné (definovaný horizont)
  - uzly = stavy, hrany = akce
  - odměny pouze v listech (tzn. na konci hry)
  - agent nemusí mít perfektní pozorování (ale v tomto předmětu ano)

POSGs

- multiagentní rozšíření POMDPs
  - každý agent – vlastní akce, pozorování, odměny, belief (o stavu, ostatních hráčích,…)
- jeden z nejobecnějších formálních modelů – těžko algoritmicky řešitelný

2.3 Řešení deterministických MDPs

hledáme nejlepší sekvenci vedoucí k cíli => pro relevantní stavy MDP hledáme jakou akci hrát

- chceme, aby trajektorie $S_0, A_0, R_1, S_1, A_1, R_2, \ldots S_k$ maximalizovalo nahromaděnou odměnu (S_k je cílový stav)
- uvažujeme, že se odměny pouze sčítají

ZÁKLADNÍ ALGORITMUS

1. Začátek v počátečním stavu S_0
2. Aplikace možných akcí na aktuální stav a generování nových možných stavů
3. Výběr jednoho z nově generovaných stavů jako aktuálního
   a. Pokud je to cílový stav -> skonči
   b. jinak jdi do 2.

Který stav vybrat v kroku 3?

Open=list / fringe = seznam dosažených stavů, které můžeme dále prozkoumávat – vyřešíme tím situace, kdy nejlevnější akce vede k dražším akcím, než by byly v jiné cestě

VARIANTY NEINFORMOVANÝCH HLEDÁNÍ (BRUTE-FORCE)

Varianty open=listu:

1. seřazené podle rewardu + stavy k prozkoumání bereme ze začátku … **UNIFORM-COST search** (vybíráme stav s nejnižší naakumulovanou cenou, končí, když je goal-state zpracován pro expanzi)
   a. complete a optimal (pokud jsou odměny čistě záporné)
   b. varianta Dijsktrova algoritmu (ale hledá cestu jen do cílového stavu, ne do všech)
2. halda … **prohledávání DO HLOUBKY (DFS)**
   a. nemusí skončit (not complete) -> limitujeme maximální délku sekvence akcí které můžeme prohledat a limit postupně zvětšujeme = **iterative deepening**
3. fronta … **prohledání DO ŠÍŘKY (BFS)**
   a. vždy skončí (complete)
   b. najde sekvenci s nejmenším počtem akcí bez ohledu na ceny
   c. potřebuje exponenciální paměť a čas

CO KDYŽ VYGENERUJEME STAV, KTERÝ UŽ BYL NAVŠTÍVEN?

Closed-list – seznam již vyhodnocených stavů (tzn. po tom co je vybereme z open-listu a zpracujeme)

- měl by u každého stavu obsahovat předchozí stav + akci – rekonstrukce cesty

ITERATIVE DEEPENING

Kombinuje BFS a DFS (najde nejkratší cestu s paměťovými nároku DFS)

1. Zavoláme DFS s limitem hloubky 0
2. Při neúspěchu zavoláme DFS s limitem 1
3. Při neúspěchu zavoláme DFS s limited 2 atd.

Můžeme použít i k optimalizování ceny – limitujeme místo hloubky cenu a iterativně limit zvětšujeme o 1

BACKWARD / BIDIRECTIONAL SEARCH

V některých případech je lepší začít **prohledávání od konce** (např. pokud do cíle vede jen malý počet akcí, a naopak na začátku je problém komplikovaný – ale musíme umět efektivně generovat předchozí stavy)
Někdy můžeme použít **prohledávání z obou směrů** paralelně – pokud má **nejkratší řešení hloubku d**, stačí nám expandovat $\mathbf{b^{d/2}}$ stavů (b je branching factor = počet dostupných akcí) – pak musíme zajistit, aby se prohledávání setkalo uprostřed (vymyšleno 2016)

° **Algoritmus MM (2016)** = jako A*, ale prioritní řazení v open-listu podle $pr_F(n) = \max(f_F(n), 2g_F(n))$

3. Informované (heuristické) hledání

3.1 A algoritmus*

= uniform-cost prohledávání + kritérium pro expanzi

- co nejnižší **f(n) = g(n) + h(n)** … suma akumulované ceny ve stavu a heuristického odhadu

Vymyšlen jako část Shakeyho (1968) – od té doby mnoho variant

HEURISTIKA

= odhad ceny z aktuálního stavu do cílového (h: S -> R^+)

° optimální heuristika h^* = skutečná cena ze stavu do cíle

- snaha přiblížit se h* (odhad je často zjednodušením problému)

Přípustná heuristika – optimistická (není nikdy vyšší než optimální heuristika neboli skutečná cena)

° $h(n) \leq h^*(n)$ pro všechny stavy n z N

° zaručuje, že A* nalezne optimální řešení

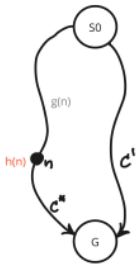
Konzistentní heuristika – heuristika stavu n není nikdy větší než heuristika sousedního uzlu + cena do sousedního uzlu

° $h(n) \leq h(n') + c(n, n')$ neboli $g(n) + h(n) <= g(n') + h(n')$ … n‘ je sousední stav následující po n

° každá konzistentní heuristika je i přípustná

° Zaručuje, že stav je vybrán pro exploraci až ve chvíli, kdy už jsme do něj nalezli optimální cestu

OPTIMALITA A*



DŮKAZ:

C* … cena optimální cesty do cíle G

A* algoritmus najde suboptimální cestu s cenou C‘ > C*, v open listu je určitě node, který je na optimální cestě z počátečního stavy do cíle (pojmenujme n)

$$f(n) = h(n) + g(n) \leq h^*(n) + g(n) \leq C^* < C'$$

Tzn. node n bude určitě expandován dříve, než bude dosaženo stavu G přes suboptimální cestu

Sol. length	Iterative Deepening	$A^*(h_1)$	$A^*(h_2)$
8	6384	39	25
12	3644035	227	73
16	—	1301	211
20	—	7276	676

Věta: A* je **optimally efficient** = žádný jiný optimální algoritmus nemůže navštívit méně uzlů než A*

° A* expanduje na základě kritéria $f(n) = g(n) + h(n) < C^*$ … každý jiný algoritmus musí také navštívit tyto uzly, jinak by nemusel najít optimální řešení

ITERATIVE DEEPENING A* (IDA*)

Snižuje paměťové nároky A*

Zavedeme limit pro cenu c, který iterativně zvětšujeme – nejsou expandovány uzly g(n) + h(n) > c

DESIGNOVÁNÍ PŘÍPUSTNÉ HEURISTIKY

° Zjednodušený (relaxovaný) problém – např. v bludišti nejsou překážky

° Vyřešíme jen jeden podproblém

° Rozdělíme problém do několika podproblémů

Heuristika by měla být informativní

° Přípustné heuristiky h_1 a h_2: $h_1(s) > h_2(s)$ pro všechny stavy … h_1 je informativnější

° **Informativnější heuristika expanduje méně uzlů**

PŘÍKLAD: 8-PUZZLE:

1		2
4	5	3
7	8	6

Některé přípustné heuristiky:

- počet špatně umístěných políček (h_1)
- manhattanská vzdálenost čísel od jejich správných pozic (h_2)

Počet expandovaných uzlů

PŘÍKLAD: BLUDIŠTĚ – heuristika – manhattanská vzdálenost

NEPŘÍPUSTNÉ HEURISTIKY

° snížení počtu expandovaných uzlů (zvyšují rychlost a snižují paměťovou náročnost)

° není garance optimálního řešení.

Takovou heuristiku můžeme naimplementovat např. jako h(s) = epsilon * přípustná heuristika (epsilon > 1) => nalezené řešení má cenu maximálně epsilon-krát větší než optimální řešení

Hodí se, pokud nepotřebujeme nutně úplně nejlepší cestu, ale chceme ji co nejrychleji nebo u hodně velkých problémů

4. Reinforcement learning I

ZPĚTNOVAZEBNÍ UČENÍ = učení výběru akcí optimalizujících odměny na základě zkušenosti – pokus omyl

° Je potřeba méně lidských vstupů

4.1 Multi-armed bandit problém – EXPLORACE vs. EXPLOITACE

Mějme sadu automatů s různými (nám neznámými) pravděpodobnostmi výhry a její výše. **Jak najít strategii, která maximalizuje naši celkovou výhru?**

OBECNĚ:

◦ V každém kroku t = 1, 2, …, T vybereme možnou akci A_t a reálnou odměnu R_t

- Odměna záleží pouze na vykonané akci a je nezávisle identicky distribuovaná – očekávaná odměna:

$$q*(a) = E[Rt \mid At = a], pro\ všechna\ a\ z\ \{1, \dots, k\}$$

- skutečné hodnoty ani distribuce neznáme.

Musíme **maximalizovat celkovou odměnu**.

Musíme jednak zkoušet různé akce, abychom zjistili jejich hodnotu (**explorace**) a současně preferovat ty, které se nám zdají nejlepší (**exploatace**).

Utvoříme si odhad $Q_t(a)$ … např. průměrná odměna dané akce => **greedy akce** je $\mathbf{A^*_t = argmaxQ_t(a)}$

° výběr $A_t = A^*_t$ … *exploitace*

° výběr A_t != A^*_t … *explorace*

Jak dělat oboje ve správném poměru?

ε – GREEDY ACTION SELECTION

° Nejjednodušší způsob balancování explorace a exploitace

- P(nejlepší akce) = 1 – ε … exploitace
- P(random akce) = ε … explorace

ALGORITMUS:

Q(A) .. očekávaná odměna

N(A) … kolikrát byla zahrána akce A

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

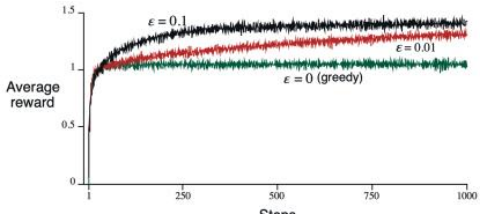
Repeat forever:

$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

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



VÝPOČET OČEKÁVANÉ HODNOTY

$$Q_n = \frac{R_1 + R_2 + \dots + R_{n-1}}{n-1} \dots \text{očekávaná hodnota před akcí n}$$

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) = \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) = \frac{1}{n}(R_n + (n-1)Q_n) = \frac{1}{n}(R_n + nQ_n - Q_n)$$

$$= Q_n + \alpha_n[R_n - Q_n]$$

Zajištění konvergence $\sum_{n=1}^{\infty} \alpha_n(a) = \infty$ and $\sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$

NovýOdhad = StarýOdhad + StepSize [NováHodnota – StarýOdhad]

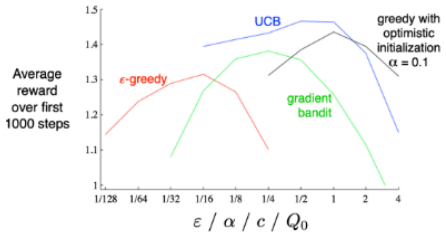
° Step-size je výhodný v situacích, kdy je problém **NONSTATIONARY** (skutečné hodnoty akci se pomalu mění v čase)

- starší hodnoty jsou méně relevantní než novější (nastavení step-size toto zohlední)

OPTIMISTICKÁ INICIALIZACE HODNOTY

Je lepší na začátku udělat nadhodnocenou inicializaci – na začátku budeme explodovat, protože více vyzkoušeným akcím se bude hodnota snižovat, pak narazíme na optimálnější.

UPPER CONFIDENCE BOUND



= vybíráme hodnoty s nejvyšší horní hranicí (čím vyšší je poměr hraní dané akce, tím víc že zmenšuje rozptyl)

Výpočet horní hranice:

$$A_t = arg\max_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$$

4.2 Aplikace bandit problému na MDPs

POLICY v kroku t = udává vektor pravděpodobností se kterými bude v daném stavu zahrána akce a

$\boldsymbol{\pi_t(a|\, s)}$ = pravděpodobnost akce a ve stavu s

- Speciální případ – deterministické policies (psti 0 nebo 1)

MAXIMALIZACE OČEKÁVANÉHO NÁVRATU (EXPECTED RETURN)

Agent maximalizuje expected return = střední hodnota G_t

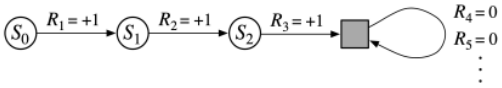
- **Total reward** … $G_t = R_{t+1} + R_{t+2} + \dots + R_T$ (součet všech následujících odměn v epizodě)
- **Discounted reward** $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \dots$ $0 \leq \gamma \leq 1$ je **discount rate** (typicky **0.9**)
  - Důraz na krátkodobější odměny $\gamma \to 0$
  - Důraz na dlouhodobější odměny $\gamma \to 1$
- **Average reward** = průměrná odměna za krok
  - používá se v problémech, které nekončí ale všechny odměny jsou stejně důležité (brzké i pozdější)
    - např. kontrola vibrací letadla, hlídání objektu atd…
  - v AI není časté

Episodic tasks = interakce je přirozeně rozdělena do epizod (např. kola hry, průchody bludištěm atd.)

- Většinou **total reward**

Continuing tasks = interakce nemá přirozené epizody, ale stále pokračuje

- Vždy **discounted reward**
  - ⇨ **SJEDNOCENÍ zápisu**

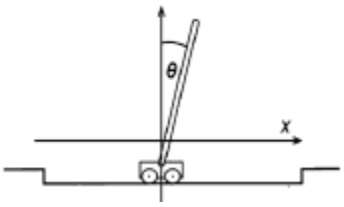


- Konec epizody = absorbující stav, které vždy produkuje 0
  - mžeme použít vzoreček pro continuing tasks:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$, kde γ je 1, jen pokud je absorbující stav vždy dosažen

PŘÍKLAD: BALANCUJÍCÍ TYČ

Cíl: vyhnout se selhání = pád tyče / nabourání vozíku



- **Episodic task** – epizoda končí selháním
  - Reward = +1 za každý krok před selháním
  - return = počet kroků před selháním
- **Continuing task**
  - reward = -1 při selhání, jinak 0
  - return = - γ^k … k = počet kroků před selháním

5. Reinforcement learning II (MDPs)

Přednáška 5 | 19.3.2024

5.1 Hodnotová funkce

hodnotová funkce stavu (state-value function) pro MDP a policy π = funkce, která každému stavu přiřadí očekávaný návrat za celkovou cestu z daného stavu do koncového stavu při dodržování dané policy

$$v_\pi: S \to \mathbb{R} \qquad v_\pi(s) = \mathbb{E}_\pi G_0$$

hodnota funkce akce (action-value function) pro MDP a policy π = mapuje pro každou dvojici (stav, akce) očekávaný návrat při zahrání akce a ve stavu s a následně dodržování dané policy

$$q_\pi: S \times A \to \mathbb{R}$$

OPTIMÁLNÍ POLICY

částečně seřazené policies:

$$\pi \leq \pi' \quad \Leftrightarrow \quad v_\pi(s) \leq v_{\pi'}(s) \quad \text{pro všechny stavy s}$$

optimální policy… π* = policy, která je v částečném řazení stejná nebo lepší než všechny ostatní

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

° Optimálních policies může být víc

- Všechny mají stejnou **optimální state-value funkci**

$$v_*(s) = \max_\pi v_\pi(s) \text{ ... pro všechny stavy s z S}$$

- Všechny mají stejnou **optimální action-value funkci**

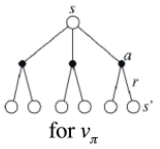
$$q_*(s, a) = \max_\pi q_\pi(s, a) \qquad \forall s \in S\,, a \in A$$

5.2 Bellmanovy rovnice

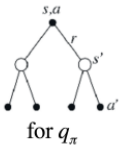
BELLMANOVA ROVNICE PRO POLICY

Umožňuje jednoduchý výpočet hodnot stavu za dané policy

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



for v_π

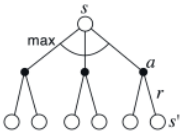


for q_π

BELLMANOVY OPTIMALITNÍ ROVNICE

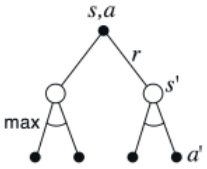
Hodnota stavu při optimální policy = maximum z očekávaných návratů (return) pro nejlepší akci ze stavu

$$v_*(s) = \max_a q_{\pi*}(s, a) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]$$
$$= \max_a \sum_{s',r} p(s', r\,|s, a)[r + \gamma v_*(s')]$$



Hodnota stavu + akce při optimální policy

$$q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a\right]$$
$$= \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right].$$



Souvislost s A* - hodnota stavu při optimální policy odpovídá cca optimální heuristice daného stavu

$$v_*(s) \approx -h^*(s)$$

5.3 Policy evaluation

= výpočet state-value funkce pro danou policy

ITERATIVE POLICY EVALUATION

$$v_0 \to (\text{sweep}) \to v_1 \to \ldots \to v_k \to (\text{sweep}) \to v_{k+1} \to \ldots \to v_\pi$$

° sweep = aplikování **backup operace** na každý stav (to je ta následující rovnice)

- back=up operace vychází z bellmanovy rovnice, ALE! state-values sousedních stavů bereme z předchozí iterace

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_k(s')\right]$$

ALGORITMUS:

Input: policy, která má být ohodnocena

Průběh:

- Inicializujeme state=value všech stavů na 0 → tzn. pole V(s) = 0
- Opakujeme, dokud Δ (= maximální změna state-value) není menší než zadaná hranice (malé positivní číslo)
    - Δ ← 0
    - Pro všechna s:
        - v ← V(s)
        - V(s) ← backup operation
        - Δ ← max(Δ, |v-V(s)|)

Output: V … blíží se skutečné state-value funkci pro danou policy

γ-KONTRAKCE

Definice: Libovolná funkce F : R^n -> R^n je γ-kontrakce pro $0 < \gamma < 1$ ⇔ pro nějakou normu || a všechna x, y z R^n platí:

$$\| F(x) - F(y) \| \leq \gamma \| x - y \|$$

Věta: Pro γ-kontrakci F platí:

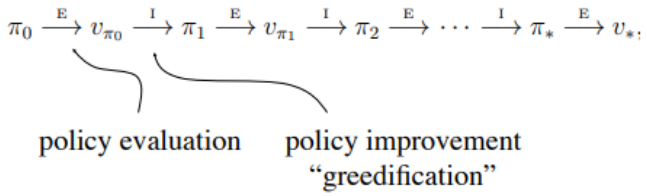
- Iterativní aplikace F konverguje do **jediného** fixního bodu nezávisle na bodu počátečním
- Konverguje lineární rychlostí určenou γ

ZLEPŠOVÁNÍ POLICY

Pokud existuje stav *s* a akce *a* takové, že: $q_\pi(s, a) > v_\pi(s)$ pak nastavení π(s) = a zlepší policy

° Strategii to nerozbije protože:
- Hodnota v *s* se zlepší
- Hodnoty ve stavech, které vedou do *s* se tím také zlepší
- Žádné jiné hodnoty se nezmění

° Nerozbije se ani pokud upravíme více hodnot najednou



ALGORITMUS:

1. Inicializujeme V(s) ϵ R, π(s) ϵ A(s)
2. **Evaluace policy**
   Opakujeme, dokud Δ (= maximální změna state-value) není menší než zadaná hranice (malé positivní číslo)
    a. Δ ← 0
    b. Pro všechna s z S
        i. v ← V(s)
        ii. V(s) ← backup opeartion
        iii. Δ ← max(Δ, |v-V(s)|)
3. **Zlepšování policy**
    a. policy-stable ← true
    b. Pro všechna s z S
        i. a ← π(s)
        ii. π(s) ← argmax přes a z backup operation
        iii. pokud a != π(s) pak policy-stable ← false
    c. pokud policy-stable true, vrátíme V a π, jinak opakujeme 2.-4.

GENERALIZOVANÁ POLICY ITERATION

Nemusíme střídat evaluaci všech stavů a následně zlepšování policy pro všechny stavy -> **konvergence je zaručena** i pokud kombinujeme **libovolné** konzistentní **zlepšení odhadu hodnoty** s libovolným konzistentním **zlepšením policy na základě dané hodnoty**

° podset stavů (i jeden)

° malé zlepšení ve správném směru atd.

VALUE ITERATION

Full value-iteration backup

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_k(s')\Big] \qquad \forall s \in \mathcal{S}$$

Algoritmus:

Vstup: V(s) = 0

Opakujeme, dokud Δ > θ (malé kladné číslo)

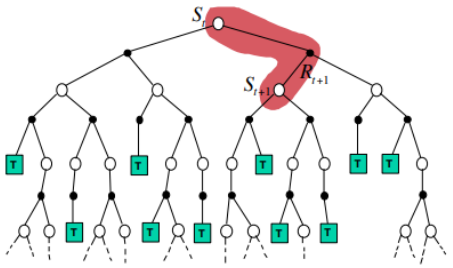
- Δ ← 0
- Pro každé s ϵ S:
  - v ← V(s)
  - V(s) ← full value-iteration backup
  - Δ ← max(Δ, | v – V(s) |)

Výstup: policy π, taková, že:

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

6. Q-learning a DQN

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

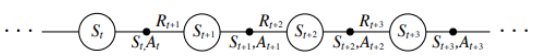


Ve skutečném světě nemůže robot prozkoumávat při RL vždy všechny možné akce daného stavu (může v daném běhu udělat jen jednu akci) a nezná dokonale dynamiku prostředí – proto počítá state-value funkci přes několik běhů = **temporal difference learning** (rozdíl mezi tím, jak očekáváme, že se svět bude chovat při daném stavu a akci vs. jak se skutečně chová)

° Dokážeme zjistit value funkci bez toho, abychom znali dynamiku světa (vzoreček u stromu)

Jak vědět, kterou akci udělat, aby nás dostala do stavů s vysokou value funkcí? => **action-value funkce**

° Po každém přechodu ze stavu S_t updatujeme jeho action value funkci (opět **iterativní odhad**, ale jen pro danou policy, ne pro nejlepší)



$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

- Pokud je S_{t+1} terminální, pak $Q(S_{t+1}, A_{t+1}) = 0$
- A_{t+1} je akce kterou skutečně v dalším kroku použijeme

6.1 Q-learning

° Zjistí nám action-value funkce optimální policy, aniž bychom tu optimální policy znali

° K updatování používáme hodnotu nejlepší možné akce v dalším kroku (ale pak můžeme prozkoumávat libovolnou akci)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

ALGORITMUS:

° **Inicializujeme:** Q(S, A) pro všechny s a A(s) libovolně, Q(terminal_state, *) = 0

° **Opakujeme** pro každou epizodu, dokud S není terminální stav
- Inicializujeme S
- Opakujeme v každém kroku epizody
  - Vybereme A z S za pomoci policy z Q (ε-greedy, nebo jiná bandit policy – abychom exploitovali i explorovali)
  - Zahrajeme akci A, pozorujeme R, S‘
  - $Q(S,A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
  - $S \leftarrow S'$

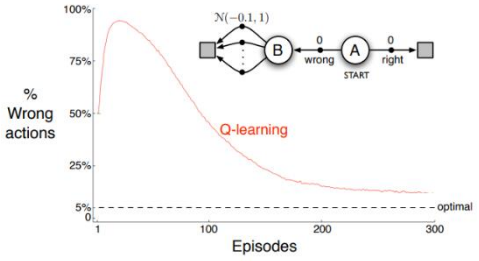
MAXIMIZATION BIAS

° Wrong action dává reward do B 0 a pak v normálním rozdělení N(-0,1 ; 1)

° Right action dává reward 0 (terminální stav)

Průběh Q-learningu

- $A_1 = r \rightarrow R = 0 \rightarrow Q(A, r) = 0$
- $A_2 = w \rightarrow R_1 = 0 \rightarrow Q(A, w) = 0$
  $R_2 = -0.5 \rightarrow Q(B, b_1) = -0,05$
- $A_3 = w \rightarrow R_1 = 0 \rightarrow Q(A, w) = 0$
  $R_2 = 0,5 \rightarrow Q(B, b_2) = 0,05$
- ....
- $A_k = w \rightarrow R_1 = 0 \rightarrow Q(A, w) = 0,05$

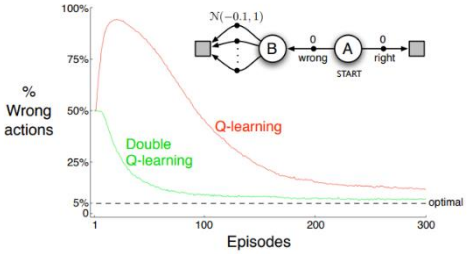


Protože Q-learning vybírá vždy u wrong akce maximum z Q(S‘, a), vybere vždy to nejlepší (i když to bude náhodou)! Tzn. hodnota wrong akce bude stoupat, ačkoliv průměr akcí by měl být -0,1 – pokud bude akcí hodně, bude trvat algoritmu několik epizod, než zjistí, že lepší je akce right (čím více akcí vede z B, tím déle bude algoritmu to trvat)

DOUBLE Q=LEARNING (2010)

Je řešením maximization bias.

° Trénujeme pomocí Q-learningu action-value funkce Q_1 a Q_2
- Trénujeme vždy na odlišných time-steps => Q_1 a Q_2 jsou nezávislé
- Náhodně vybíráme, která bude v daném kroku updatovaná
- Pro update používáme vždy **kombinaci funkcí** (vezmeme nejlepší akci jedné funkce a vyhodnotíme uloženou hodnotu druhé funkce)



$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\left(R_{t+1} + Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)\right)$$

6.2 MDP reprezentace Atari her

° 1 stav = 4 po sobě jdoucí framy (1 – známe pozici, 2 – známe rychlost, 3 – známe zrychlení, 4 - abychom si byli jistí)
  - Redukujeme velikost obrázku na 84 x 84 x 1
° Akce: 8 směrů joysticku + tlačítko (2 x 8 – každý směr má verzi s tlačítkem a bez něj)
° Odměny: každá hra má jiné => sjednocení: +1 za každé zvýšení skóre, -1 za každé snížení skóre

Velmi paměťově náročné => s černobílými obrázky stále $2^{84x84x4}$
Q funkce je vysoko-dimenzionální funkce, kterou aproximujeme pomocí NN (méně paměťově náročné) – na vstupu vektor 28224 čísel + akce -> vrátí reálné číslo

$$\hat{l}(\tilde{D}, w) = \frac{1}{|\tilde{D}|} \sum_i \| f_w(\vec{x_i}) - \vec{y_i} \|$$

- Pro trénování se používá minibatch SGD s loss funkcí
  - x_i je jeden stav (4 framy)
  - y_i je návratová hodnota value-action funkce
  - D je minibatch (malá podmnožina datasetu)
- minibatch verze v průměru cca odpovídá verzi ze všech bodů

6.3 DQN algoritmus – použité na řešení Atari her

° ve 20 hrách horší než člověk, ve 29 lepší než člověk

ALORITMUS 1: deep Q-learning with experience replay

Inicializace:

° Replay memory/experience buffer D velikosti N (uchováváme v ní posledních N-zkušeností z minulosti – tzn. co jsme v daném stavu zahráli a jaký byl reward – i přes několik epizod)
° Action-value funkce Q s náhodnými váhami θ (NN s náhodnými váhami -> náhodná Q-funkce)
° Target Action-value funkce Q^- se stejnými váhami $\theta^- = \theta$ (budeme dělat double Q-learning)

Průběh

° Pro epizodu 1, .. M opakuj:
  - Inicializuj sekvenci $s_1 = \{x1\}$ a preprocessed sekvenci $\Phi_1 = \Phi(s1)$ – tzn. převedeme na 84x84x1 (greyscale)
  - Pro t = 1, .. T opakuj:
    - S pravděpodobností ε vyber náhodnou akci a_t, jinak vyber $a_t = \text{argmax } Q(\Phi(st), a; \theta)$ (je to jako ε-greedy)
    - Provedeme akci a_t a obdržíme reward r_t a obrázek x_{t+1}
    - Nastavíme další stav $s_{t+1} = s_t, a_t, x_{t+1}$ a předzpracujeme $\Phi_{t+1} = \Phi(s_{t+1})$
    - Uložíme přechod do D $(\Phi_t, a_t, r_t, \Phi_{t+1})$
    - Vysamplujeme náhodný minibatch přechodů z D $(\Phi_j, a_j, r_j, \Phi_{j+1})$
      - Pokud epizoda končí v kroku j + 1, pak $y_j = r_j$
      - Jinak $y_j = r_j + \gamma \max_{a'} Q^-(\Phi j+1, a'; \theta^-)$
    - Provedeme gradient descent na $(y_j - Q(\Phi j, a'; \theta))^2$ vzhledem k parametrům θ
    - V každých C krocích znovu nastavíme $Q^- = Q$

7. Dvou-hráčové hry

Přednáška 7 | 26.3.2024

Případy, kdy jiný hráč libovolně vybírá akce – **jak najít optimální strategii?**
TEORIE HER - definuje optimální chování hráče vzhledem k tomu, co hrají ostatní (nebo co si myslí, že budou hrát)

7.1 EFGs s perfektní informací

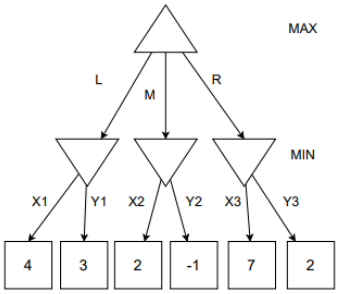
EXTENSIVE-FORM REPREZENTACE (sekvenční hry s konečným horizontem; generalizuje search tree)

- **Množina hráčů** P = {1, 2} = {MAX, MIN}
- **Množina historie akci od počáteční pozice** H (H_i = množina rozhodovacích bodů hráče i)
- **Konečná množina akcí** A … A(h) je množina možných akcí v uzlu h
- **Podmnožina konečných stavů** … Z (podmnožina H)
- **Utility funkce u**: Z -> R … přiřazuje konečnému stavu hry výsledek (reálné číslo)

Zero-sum hry … utility hráče 1 = - utility hráče 2
- minimalizace utility hráče 1 = maximalizace záporné utility funkce hráče 1

7.2 Řešení dvouhráčových her

MINIMAX ALGORITMUS (upravené DFS) = backward induction



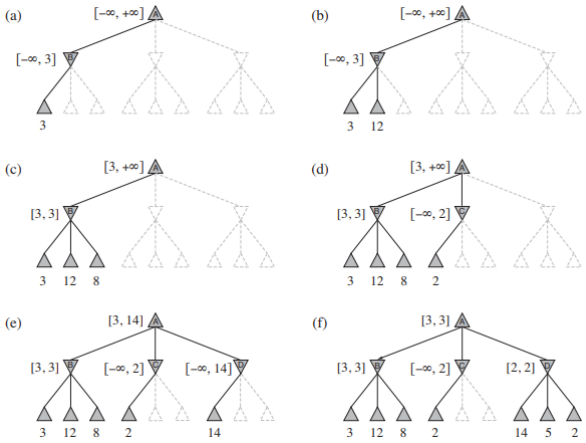
- h je terminální historie => vrať u(z)
- h je rozhodovací uzel => ohodnoť všechny potomky: v_a = search(ha), a ϵ A(h)
  - h ϵ H_1 (maximalizují hráč), vrať $\max_a v_a$
  - h ϵ H_2 (minimalizující hráč) vrať $\min_a v_a$

Hodnota hry = utility funkce hráče 1, pokud oba hráči hrají optimálně

ALPHA-BETA PROŽEZÁVÁNÍ (pruning)

Pomáhá snížit náročnost prohledávání v minimax algoritmu

- K prohledávání přidáme spodní hranici (α) a horní hranici (β) odhadu hodnoty hry
- Search space = $O(b^{d/2})$ pro optimální seřazení akcí (minimax měl $O(b^d)$)



Část algoritmu pro rozhodovací uzel h (α_h a β_h jsou předány jako parametry – v prvním kole inicializujeme na +- nekonečno))

- Pokud h je list, vrať u(z)
- Pokud h je rozhodovací uzel a na tahu je MAX hráč, pak pro každou akci a_h
  - $v_h = \max(v_h, \text{search}(a_h, \alpha_h, \beta_h)$ $\quad \alpha_h = \max(\alpha_h, v_h)$
  - pokud $\beta_h \leq \alpha_h$ => break
- Pokud h je rozhodovací uzel a na tahu je MIN hráč, pak pro každou akci a_h
  - $v_h = \min(v_h, \text{search}(a_h, \alpha_h, \beta_h)$ $\quad \beta_h = \min(\beta_h, v_h)$
  - pokud $\beta_h \leq \alpha_h$ => break
- vrať v_h

NEGAMAX

= Zjednodušení pro hry, kde se hráči střídají pravidelně

1. Pokud h je list, vrať u(z)
2. Pokud h je rozhodovací uzel pak pro každou akci a_h
   - $v_h = \max(v_h, -\text{search}(a_h, -\beta_h, -\alpha_h,)$ $\quad \alpha_h = \max(\alpha_h, v_h)$
   - pokud $\beta_h \leq$ a α_h -> break
3. vrať v_h

ZMĚNA BOUNDS α a β

Pokud na začátku inicializujeme bounds na nějakou naši hodnotu (např. [9, 11]) můžeme **evaluovat méně uzlů**

- Pokud se nám vrátí hodnota mimo tento interval -> náš odhad byl špatný
  - Pokud je menší než interval (např. 7) => nebyly správně prozkoumány uzly MIN hráče => spustíme znovu s intervalem [- nekonečno, 7]
  - Pro vyšší hodnotu se to dělá naopak

Další možností je systematicky posouvat okna podle toho, jaká hodnota se vrátila

NEGA-SCOUT

- Zavedení heuristiky – pomůže seřadit akce před každou evaluací nodu
- První akci celou evaluujeme => druhou akci evaluujeme s minimálním intervalem
  - Pokud vrácená hodnota stromu je stejná nebo horší, pokračujeme s další akci
  - Pokud je vrácená hodnota lepší, musíme tuto akci správně evaluovat

ALGORITMUS

- Pokud je h list => vrať u(z)
- $\beta'_h = \beta_h$, $\alpha'_h = \alpha_h$
- h je rozhodovací uzel MAX hráče – pro všechny možné akce proveď
  - v_h = search(a_h , α_h, β'_h)
  - if $\alpha_h < v_h < \beta_h$ a h není první potomek
    - v_h = search(a_h , v_h, β_h)
  - α_h = max(α_h, v_h)
  - if $\beta_h \leq \alpha_h$ => break
  - $\beta'_h = \alpha_h + 1$
  - return α_h
- h je rozhodovací uzel MIN hráče – pro všechny možné akce proveď
  - v_h = search(a_h , α'_h, β_h)
  - if ($\alpha_h < v_h < \beta_h$ a h není první potomek)
    - v_h = search(a_h , α_h, v_h)
  - β_h = min(β_h, v_h)
  - pokud $\beta_h \leq \alpha_h$ => break
  - $\alpha'_h = \beta_h - 1$
  - return β_h

NEGASCOUT vs. ALPHA-BETA

Negascout může evaluovat některé uzly několikrát, ale nikdy neevaluuje několik různých historií
Je o 10 – 20% rychlejší než alpha-beta

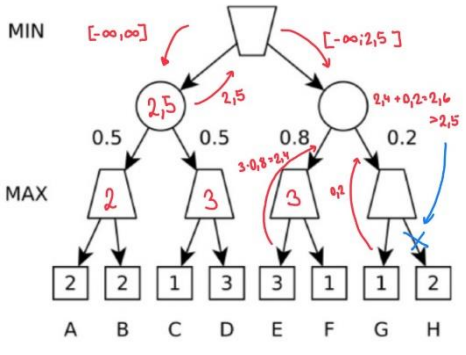
7.3 Game solving vs. game playing

- Algoritmy, které jsme probírali jsou tzv. **offline algoritmy**
  - Nejsou použitelné pro většinu skutečných praktických her, vzhledem k velikosti herního stromu
- Algoritmy skutečně hrající hry prohledávají jen do **limitované hloubky**, místo utility funkce vrací **heuristickou evaluation funkci** (tu je těžké nadesignovat)
  - Strategie hledaná do limitované hloubky nemusí odpovídat skutečné optimální strategii
- Možností je **transposiční tabulka** – cache tabulka dříve evaluovaných pozic, minimalizuje negativní efekty herního stromu

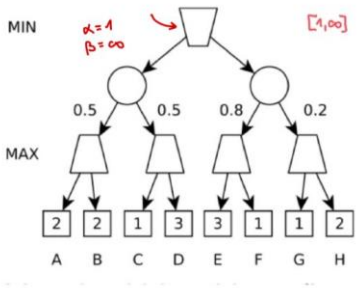
7.4 Hry s náhodou

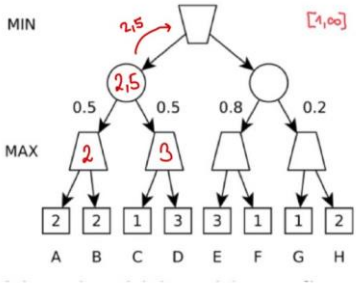
Dvouhráčové hry můžeme lehce rozšířit o stochastické události – přidáme 3. hráče - **náhodu**

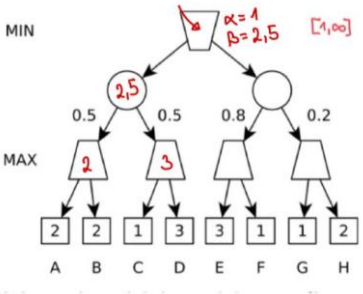
Vyřešit, jestli alpha-beta něco vyprunuje! - myslím, že když utility values nejsou omezeny, **tak se nic nemůže vyprunovat**, druhá množnost je, že takhle

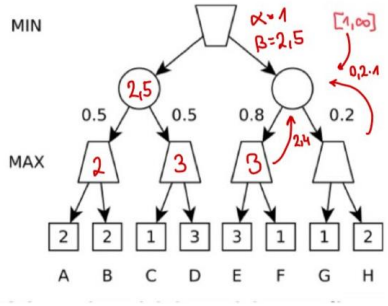


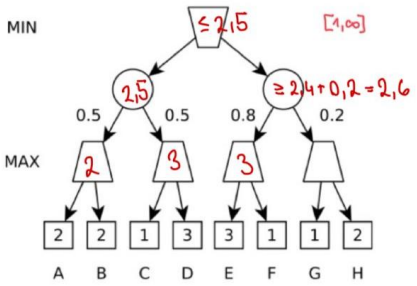
Co se stane, pokud possible utility values jsou z intervalu [1, nekonečno]? – asi takto

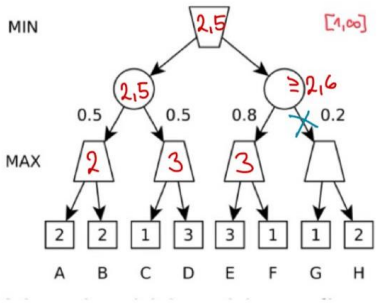










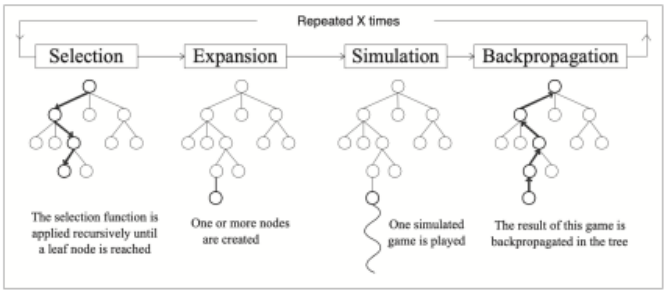


8. MCTS a AlphaGo

- Skutečné hry mají obrovské množství možných stavů (šachy - 10^{45}, go – 10^{170})
- Minimax
  - Není reálné abychom to minimaxem zvládli
  - nejúspěšnější αβ-search prohledá v šachách 10^{23}, v go 10^{85} stavů
  - navíc by byla potřeba extrémní paměť na uložení strategie
- Depth-limited search – prohledávání utneme v hloubce h a cenu uzlů v této hloubce ohodnotíme pomocí heuristiky
  - Pořád extrémně velké množství stavů
    - ⇨ Monte Carlo se s tímto vypořádává

8.1 Monte Carlo Tree Search

- Místo evaluační funkce používá náhodné simulace k dohrání hry
  - Průměrně u špatných pozic bude víc prohrávat a u lepších víc vyhrávat i při náhodných hrách
- Prohledává víc statisticky lepší tahy, než statisticky horší tahy
- Ačkoliv byl vymyšlen pro Go, dá se použít v libovolném prohledávání stavového prostoru (i jednohráčové problémy – PROST, POMCP,…, plánovaní atd.)



(Image from Chaslot et al. 2007)

- **Základní fáze algoritmu**:
  - **Selekce** – pomocí nejlepších tahů se dostaneme do listu našeho uloženého podstromu
  - **Expanse** – vybraný nejlepší list rozšíříme o jednoho potomka (nebo o všechny potomky), toho uložíme do našeho podstromu v paměti
  - **Simulace** – z nově přidaného uzlu provedeme jednu náhodnou simulaci hry
  - **Backpropagace** – podle toho, jak hra dopadla updatujeme statistiky ve všech uzlech našeho podstromu, které jsme v tomto uzlu navštívili

MCTS SELEKCE

- Dilema exploitace vs. explorace => můžeme použít jakýkoliv algoritmus pro multi-armed bandit problém
- **UCT**
  - UCT = MCTS + UCB (Upper Confidence Bound)
  - Nejpoužívanější verze selekce
  - **Výběr akce** při selekci v uzlu s

$$A_t(s) \doteq \arg\max_a \left[Q_t(s,a) + c\sqrt{\frac{\log N_t(s)}{N_t(s,a)}} \right]$$

MCTS EXPANSE

- Možnosti expanse
  - Přidání všech potomků
  - Přidání jedné akce (dobré, pokud je velký branching factor) – pokud se dostaneme do uzlu, kde ještě nejsou přidané všechny děti, tak preferujeme ty, které jsme ještě nepřidali
  - Přidání uzlu až po několikátém navštívení (ještě více omezuje růst stromu)
  - **Progressive widening** – někdy je tolik akcí, že ani MCTS by to nestihlo
- Progressive widening
  - Heuristiky si zvolíme jen pár akcí (heuristicky nebo náhodně) a nejprve pořádně prozkoumáme tyto vybrané
    - Sice nebudeme mít prozkoumané všechno, ale budeme mít představu o tom, co prozkoumané máme, což je lepší, než mít hodně akcí ale téměř nic o nich nevědět
  - Další akce přidáme až když předchozí jsou dostatečně dobře prozkoumané
  - Počet akcí, které máme přidané v jednom stavu doporučeno $k = \lceil C * N(s)^{\alpha} \rceil$
  - Funguje i pro hry s nekonečným množstvím akcí (= **infinitely many armed bandits**)

MCTS SIMULACE

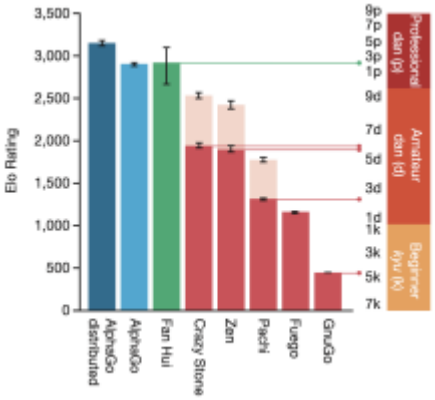
- Důležité je, aby byly hodně rychlé
- Možnosti – u všech se přidává nějaká míra náhody, kdybychom náhodou měli chybnou simulation policy
  - Čistě náhodné
    - Potřeba hodně velké množství simulací

        - Hodně rychlé
    - Ručně naimplementovaná policy – např. 60% policy, 40% náhoda
    - Naučené znalosti – např. použít supervised learning na nějakých reálných hrách atd., ale opět přidat náhodu

MCTS BACKPROPAGATION

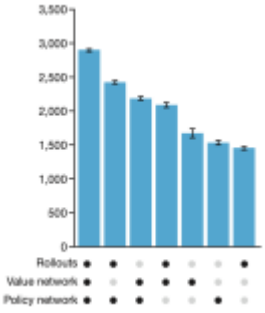
° Chceme udržovat statistiky
° Můžeme si pamatovat cokoliv, ale většinou:
    - N(s) … počet navštívení uzlu
    - N(s, a) … kolikrát jsme ve stavu zahráli danou akci
    - Q(s, a) … hodnota akce ve stavu
° Další možnosti jsou např. reward range, RAVE = Q(a) – hodnota akce nezávisle na tahu, …

8.2 AlphaGo



° Go vybráno, protože je to náročné – branching cca 350, délka hry cca 300 tahů

° **Hlavní myšlenky**:
    - Základní algoritmus – MCTS
    - Selekce a simulace používají kombinaci naučených policies
        - Lidská znalost - $\mathbf{p_\sigma(a \mid s)}$
            - Data ze serveru, kde amatéři hrají Go
            - Naučené pomocí ML (NN) – SGD maximalizace očekávané hodnoty pravděpodobnosti zahrání akce v daném stavu
            - Přesnost jen cca 57%
            - Ale vyhodnocení trvá dlouho
        - Velmi rychlá policy pro simulace - $\mathbf{p_\pi(a \mid s)}$
            - Naučeno na stejných datech jako předchozí policy
            - Mnohem jednodušší NN – cca 1000x rychlejší
            - Přesnost 24%
        - RL policy naučená při hraní proti sobě - $\mathbf{p_\rho(a \mid s)}$
            - Nejprve nainicializovaná pomocí p_σ
            - Použit SGA (SG ascent) – pokud hra vyhrála, maximalizovala se pst akce, pokud prohrála, minimalizovala se pst akce
            - Nakonec nad p_σ vítězila ve více než 80%
        - Value funkce naučená pomocí RL
            - Spočítaná pro policy $p_\rho(a \mid s)$
            - Z každé hry se vybral jen jeden náhodný stav a minimalizovala se MSE mezi v(s) tohoto stavu a hodnotou hry {-1,1}
    - Fáze MCTS
        - Selekce – něco jako UCB, ale kombinováno s lidskou znalostí
        - Expanse – nejspíš se neexpanduje vždy, ale list musí být několikrát navštíven
        - Simulace – výsledná value funkce je kombinace naučené value funkce a simulace pomocí p_π
        - Backpropagace N(s, a) a Q(s, a)



° AlphaGo vyhrálo 494/496 her proti jiným programům
° Všechny komponenty důležité (viz obrázek)
° Další vylepšení:
    - AlphaGO Zero – bez lidských znalostí, trénink jen self-play
    - AlphaZero – obecný algoritmus (šachy, shogi, Go), nepoužíval simulace
    - µZero(2020) – nepotřebuje ani znát pravidla hry, jen dostává skóre výhra/prohra/ilegální tah

9. CSP a Scheduling

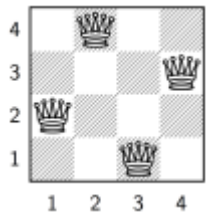
- Důležitý je jen cílový stav (cesta ani počáteční stav ne)
- Úlohy, které můžeme reprezentovat jako CSP
  - Hlavolamy – sudoku, cryptarithmetic
  - NP problémy – SAT, barvení grafu
  - Rozvrhování (scheduling)
  - …

Reprezentace

- 3 konečné množiny:
  - **Proměnné** (variables) – $x_1, x_2, \ldots, x_n$
  - **Ohodnocení** (domains) – D_i pro každou proměnnou x_i (hodnoty, kterých mohou proměnné nabývat)
  - **Omezení** (constraints) – $c_1, c_2, \ldots, c_m$
    - Každé omezení se týká podmnožiny proměnných a udává nám množinu povolených přiřazení jednotlivých domén (ohodnocení)
    - Všechna omezení reprezentujeme binárně – pokud nejsou binární, můžeme je jako binární vyjádřit zavedením nové proměnné (jako domény bude mít všechny k-tice, které původní podmínku splňují, vytvoříme k binárních omezení)
- **Cíl**: najít přiřazení tak, aby žádné omezení nebylo porušeno

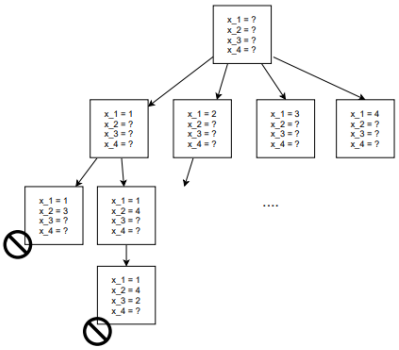
Příklad: Problém N-dam

- Na šachovnici NxN chceme rozmístit N dam tak, aby se vzájemně neomezovaly



- **Reprezentace:**
  - Proměnné: $x_1, \ldots, x_n$ (jedna proměnná pro každou dámu)
  - Ohodnocení: $D_i = \{1, \ldots, N\}$ (řádek, na kterém se dáma nachází, sloupce neřešíme – i-tá dáma je v i-tém sloupci)
  - Omezení:
    - $x_i != x_j$ pro všechny i, j ϵ D, kde i != j … dvě dámy nejsou v jednom řádku
      - Některé solvery podporují globální podmínku alldiferent($x_1,\ldots,x_n$)
    - $|x_i – x_j| != |i – j|$ pro všechny i, j ϵ D, kde i != j … dvě dámy nejsou ve stejné diagonále

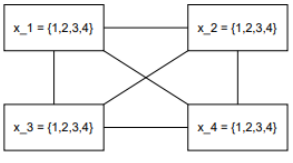
Prohledávací stromy pro CSP



- Reprezentace:
  - Uzel = (částečné) přiřazení hodnot proměnným
  - Hrana = zatím nepřiřazené hodnotě přiřadíme ohodnocení
- Neinformované prohledávání – kontrola podmínek, pokud porušujeme, tak už nemá v dané větvi smysl prohledávat dál – vracíme se stromem zpět (backtracking)

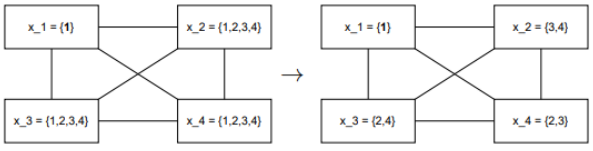
Standardní reprezentace CSP

- Díky pouze binárním omezením můžeme reprezentovat jako graf
  - Vrcholy = proměnné
  - Hrana – pokud je mezi dannými proměnnými nějaká podmínka



CSP Prohledávání s propagací podmínek

- Díky grafu můžeme omezit domény nepřiřazených proměnných dopředu (=**forward checking**)
  - Pokud se nějaká doména bude prázdná množina, víme, že tudy řešení nevede (zmenšíme prohledávanou část stavového prostoru)



Zapojení do algoritmu

- V algoritmu si uchováváme přípustné hodnoty pro jednotlivé proměnné
- Algoritmus přiřadí hodnotu proměnné x_i – pro každou proměnnou x_j, pro kterou existuje omezení c_{ij} mezi těmito dvěma proměnnými zkontrolujeme aktuální domény z D_j a ponecháme jen ty, které splňují c_{ij}
- Pokud je pro nějakou proměnnou doména prázdná algoritmus okamžitě backtrackuje

- **Heuristika** – minimal remaining value (MRV) – přiřazujeme nejprve proměnné, u kterých je méně možností (tzv. fail-fast heuristika)

Algoritmus

1. If všechny proměnné jsou přiřazené => vrátíme aktuální přiřazení
2. Jinak $x_i \leftarrow$ ChooseVariable(X, D)
3. for each $v \in D_i$:
    a. Přiřadíme $x_i = v$
    b. valid = ForwardChecking(X, D, i, v)
    c. if valid => search(X,D)
    d. undo local assignments
4. return false

Konsistenční algoritmus

- Chceme pro každou podmínku zvlášť vyhodnotit, zda může být splněná (zda je konzistentní)

Arc consistency

Děláme jednu hranu (arc) c_{ij} konzistentní (pokud bylo něco z D_i smazáno, vrátí true)

1. deleted = false
2. for each $v \in D_i$:
    a. supported = false
    b. for each $v' \in D_j$:
        i. if $c_{ij}(v,v')$ then supported = true
    c. if not supported then
        i. remove v from D_i
        ii. deleted = true
3. return deleted

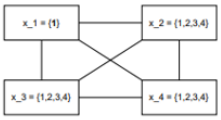
AC-3 algoritmus

Po přiřazení proměnné x_i chceme udělat všechny hrany vedoucí do uzlu i konzistentní – pokud smažeme nějakou proměnnou z x_j musí pro uzel j udělat to stejné

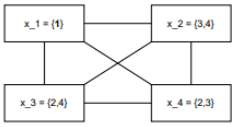
1. $Q = \{(j,i) \mid c_{ij} \in C, i \neq j\}$ … fronta všech hran, které potřebujeme zkontrolovat
2. While Q is not empty
    a. (a, b) = pop(Q)
    b. If MakeConsistent(a,b) then
        i. append(Q, $\{(k,a) \mid c_{ka} \in C, k \neq a\}$)

Příklad N-dam

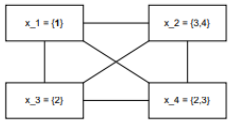
1. Přiřadíme $x_1 = 1$



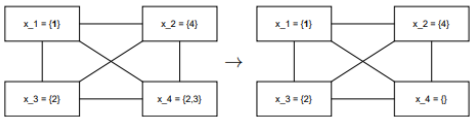
2. AC3-algoritmus – ve frontě všechny hrany (n, 1) = všechny hrany vedoucí do uzlu 1



3. AC3-algoritmus – odstranili jsme hodnoty z D_2, tudíž ve frontě všechny hrany (n, 2)



4. Odstranili jsme z D3 (to teda už kvůli kroku 1)



5. D_4 je prázdné => pro $x_1 = 1$ neexistuje řešení

- AC3 negarantuje, že pokud nic neořeže, tak existuje řešení, stále musíme nechat prohledávací algoritmus doběhnou
- Možná vylepšení
  - Least Constraining Value
    - Heuristika – vybíráme přiřazení, která podporují nejvíce dalších hodnot
  - Backjumping
    - Místo obyčejného backtrackingu najde, která hodnota mu vadí a skočí až k přiřazení této proměnné
  - Dynmic Backtracking
    - Strom nemá pevnou strukturu – pro změnu přiřazení proměnné nemusíme skákat zpět, ale jen změníme přiřazení

Constraint Optimization Problems (COPs)

- Hard constraints – musí být splněny
- Soft constraints – např. že ještě chceme za daným podmínek něco minimalizovat nebo tak

10. Logičtí agenti, plánování

10.1 Logika v AI

- **Výhody**
  - dokáže zaznamenat znalosti o světě
  - dokáže reprezentovat logické uvažování
- **Nevýhody**
  - Svět je neurčitý – čistě logické uvažovaní ohledně reálného světa je hodně komplikované (věci často ne 0 nebo 1, ale pst distribuce atd)
  - Často velmi výpočetně náročné – trade off mezi expresivitou, úplností a efektivitou
- Oblasti kde se nyní využívá
  - Interpretovatelná AI (někdy potřebujeme vědět proč a jak systém dospěl k rozhodnutí)
  - Relační ML/RL
  - Dokazování teorémů
  - Model checking (dokazování, že se nemůže někde něco stát)
  - Znalostní grafy (např. google tak někdy vyhledává, když hledáme nějaké jednoduché fakty)
  - Automatické plánování

10.2 Reprezentace logických problémů

- Klíčové charakteristiky úloh, které můžeme jednoduše řešit pomocí logiky
  - Deterministické (žádná náhoda)
  - Existuje úplný popis problému
    - Nepotřebujeme se nic učit z interakce s prostředím
  - Plánovací úlohy

Jazyk predikátové logiky (logika 1. řádu)

Jazyk

Jazyk predikátové logiky obsahuje tyto symboly:

1. logické symboly
   - proměnné; Var je množina všech proměnných
   - logické spojky: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$, popř. též $\mathrm{tt}, \mathrm{ff}, |, \downarrow, \oplus$
   - kvantifikátory $\forall$ (obecný) a $\exists$ (existenční)
   - symbol rovnosti: $=$
2. speciální symboly
   - predikátové, kde každý má svou aritu $n \geq 0$; Pred je množina predikátových symbolů
   - funkční, kde každý má svou aritu $n > 0$; Func je množina funkčních symbolů
   - konstantní; Kons je množina konstantních symbolů
3. pomocné symboly, jako jsou závorky (,) a čárka ,

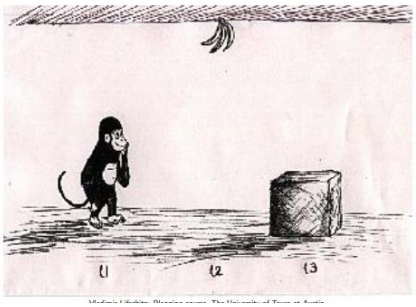
- Postupy by šly použít i se složitějšími, silnějšími logikami

10.3 Situační kalkulus

= způsob reprezentace měnícího se světa v Logice 1. řádu (FOL)

- Predikáty
  - Rigidní predikáty (platí vždy, bez ohledu na situaci)
    - Walks(monkey), moveable(box)
  - Fluentní – vážou se na určitou situaci (dána historií světa)
    - Mají navíc argument času – agent(monkey, at_ban, now)
- Situace odvozujeme pomocí predikátu *result*
  - Nová situace po provedení akce s v situaci a = result(s, a)

Příklad: Opice a banány



- Opice potřebuje posunout box pod banány, aby je mohla sníst

Predikáty

  - agent(agent name, agent position, stands on, situation)
  - object(object name, object position, who stands, situation)
- Iniciální pozice: agent(monkey, L1, ground, init), object(box, L3, none, init)
- Opice se přesune k opici: agent(monkey, L3, ground, result(init,walk)),

object(box, L3, none, result(init, walk))

Reprezentace akcí

° Pro každou akci vytvoříme axiom:
  - Efekt **walk(X, P_1, P_2)**
    - Pro všechny X, P_1, P_2, Z: (agent(X, P_1, ground, Z) ∧ walks(X))
      => agent(X, P_2, ground, result(Z, walk(X, P_1, P_2)))
  - Efekt **climb(X)**
    - Pro všechny X, P, Z: (agent(X, P, ground, Z) ∧ object(box, P, none, Z))
      => agent(X, P, box, result(Z, climb(X))) ∧ object(box, P, X, result(X, climb(X))

° **Problém rámce** = jak uchovávat fluenty, které nejsou danou akcí používány (tzn. nemění se při dané akci) – nechceme muset pro každý objekt, agenta pro každou akci říkat i to, že se s ním nic neděje
  - nedá se nikdy úplně odstranit, jen zmenšit

Logické plánování

° Máme reprezentaci stavů a akcí.

° **Cíl** = logická reprezentace požadovaného stavu
  - Případ s opicí: G … existuje Z takové, že: agent(monkey, middle, box, Z)

° **Logické uvažování** – říká, jestli ze sady axiomů vyplývá tvrzení => dokazujeme, že z axiomů vyplývá cílový stav

° Požadavky na dokazovací metody
  - **Soundness/korektnost** = pokud něco dokážeme, tak to skutečně ze znalostní báze vyplývá
  - **Completeness/úplnost** = pokud něco vyplývá, umíme to dokázat
  - **Efficinecy/Efektivnost** = chceme to dokázat v rozumném čase

° Pokud najdeme Z, které splňuje cílový stav, můžeme ho použít jako plán

° Pokud neskončí, nevíme, jestli je to proto, že algoritmus nedoběhl, nebo protože důkaz neexistuje

10.4 Doménově nezávislé automatické plánování

° Zabývá se vytvářením jazyků s vhodnými vlastnostmi pro reprezentaci plánovacích problémů

° Vytváří algoritmy pro vyřešení libovolného problému popsaného v daném jazyce

° Často extrémně výpočetně náročné

Plánovací problémy

° Charakteristiky:
  - Zabývají se výběrem akcí
  - Z iniciálního stavu má být dosaženo cílového stavu
  - Stavy se mění pomocí aplikování legálních akcí

° Plán = posloupnost akcí, které vedou z iniciálního stavu do cílového stavu
    - Mezi plány existuje uspořádání podle toho, jak dobré plány jsou – např. délka akcí, cena akcí atd.
  - **Možné cíle:**
    - Zjistit, zda řešení existuje
    - Najít jakékoliv řešení
    - Najít optimální řešení
    - Najít co nejlepší řešení v omezeném čase
    - Najít řešení, které splňuje nějakou libovolnou vlastnost, kterou si vymyslíme

Části plánování

1. Modely – definují dynamiku problému (model je např. MDP)
2. Jazyky – slouží k reprezentaci problému
3. Algoritmy – např. MCTS, A*,…

° Proč nepoužívat např. Dijsktru a hotovo? Extrémně moc různých stavů => obrovská výpočetní náročnost

10.5 Jazyk STRIPS

Klasické plánování

= plánování, kde dynamika prostředí je deterministická, nemáme žádné pozorování prostředí, úloha má konečný horizont

- Stavové proměnné => stav světa je ohodnocení stavových proměnných (booleovské)
- Akce => změny hodnot stavových proměnných

Jazyk STRIPS

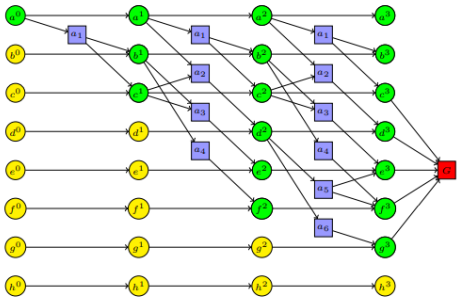
= <P, A, I, G>

- P … konečná množina **atomů** (booleovské proměnné)
- I ⊆ P … iniciální stav
- G ⊆ P … cílový stav
- A … konečná množina akcí – akce a reprezentovaná trojící
  - pre(a) ⊆ P …preconditions = předpoklady, které musí platit, aby daná akce mohla být vykonána
  - add(a) ⊆ P … add efekty = množina proměnných, které budou po aplikaci akce pravdivé
  - del(a) ⊆ P … delete efekty = množina proměnných, které budou po aplikaci akce nepravdivé

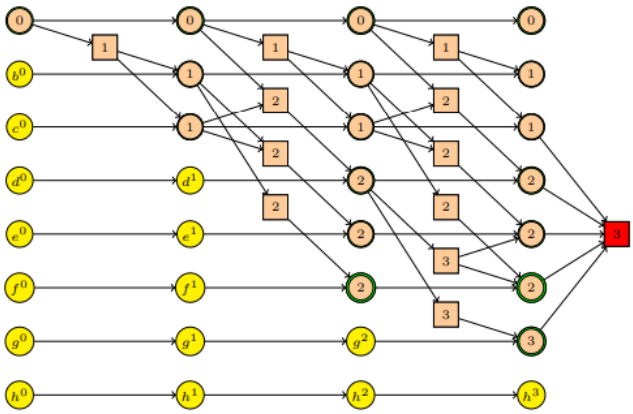
Plánovaní v jazyce STRIPS

- Můžeme použít A*
  - Stav = podmnožina pravdivých atomů
  - Aplikovatelné akce = všechny akce, které mají jako precondition podmnožinu aktuálního stavu
  - Aplikace akce = přidáme add atomy, odebereme del atomy
- Možné heuristiky
  - h(s) = |G \ s| … počet atomů, které je ještě potřeba splnit pro cílový stav
    - není přípustná, ale můžeme použít
  - zjednodušení problému
    - Abstrakce = řešení menší varianty problému (např. smažeme nějaký predikát atd.)
    - Relaxace - ve STRIPSU např. **ignorujeme delete efekty akcí**
  - Landmarks – uděláme nějaké landmarky, které musíme projít a počítáme, kolik ještě jich musíme projít

Relaxovaný plánovací graf



- Samotná relaxace nestačí – chceme i z relaxace dostat nějakou heuristiku => reachability graph (to je na tom obrázku)
  - Vezmeme množinu všech atomů, které platí v iniciálním stavu … P_0
  - Aplikujeme veškeré akce, které můžeme (s vynecháním del efektů) … A_i
  - Získáme novou množinu axiomů = P_i sjednoceno se všemi novými axiomy
  - Pokračujeme až do chvíle, kdy G ⊆ P_i
- Jak z toho získat heuristiku? Můžeme získat různé
  - $\mathbf{h_{max}}$ heuristika – v každém patře pro každý axiom vezmeme maximální hodnotu axiomu, ze kterého nám tam vede akce a k ní přičteme 1 – na konci vezmeme axiom s maximální hodnotou



11. Nejistota v AI

11.1 Racionální agenti

Racionální agent – vybírá akce, které maximalizují očekávaný užitek (přes všechny možné výsledky)

- Racionální rozhodování = utility theory + probability theory

° Použití:

- Pomoc při diagnostice: medicína, IT support, …
- Lokalizace robota: máme zašuměné senzory, nepřesné akce,…
- Zpracování přirozeného jazyka: např. určení téma textu, …

11.2 Pravděpodobnost

Základní pojmy z pravděpodobnosti

° **Náhodná proměnná** – proměnná, která nabývá různých hodnot na základě nějaké pst distribuce

° **Sdružená distribuce** = pst, že náhodné proměnné budou mít v daný okamžik zároveň dané hodnoty

° **Marginální distribuce** = pst, že jedna náhodná proměnná bude mít v daný okamžik danou hodnotu (nezávisle na ostatních proměnných)

° **Podmíněná distribuce** = pst, že proměnná S bude mít danou hodnotu v případě, že proměnná R má nějakou danou hodnotu

Joint distribution:

$P(S,R) =$

s	r	$P(S=s, R=r)$
0	0	0.20
0	1	0.08
1	0	0.70
1	1	0.02

Marginal distribution:

$P(S) =$

s	$P(S=s)$
0	0.28
1	0.72

(sum rows)

Conditional distribution:

$P(S|R=1) =$

s	$P(S=s\|R=1)$
0	0.8
1	0.2

(select rows + normalize)

Random variables are exhaustive and mutually exclusive:

$$\sum_{a \in A} P(A=a) = 1$$

Inclusion-exclusion principle:

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

Product rule:

$$P(A,B) = P(A|B)P(B)$$

Bayes' rule:

$$P(A,B) = P(B,A)$$
$$P(A|B)P(B) = P(B|A)P(A)$$
$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Paměťová náročnost

° V realitě ale jedna věc může záviset na velkém množství dalších jevů – vše musíme uchovávat v paměti

- Velikost tabulky pro sdruženou pst s proměnnými $X_1,\ldots,X_n$: $\prod_{i=1}^{n}|X_i| \geq 2^n$ … exponenciální růst
- To je paměťově i výpočetně náročné + náročné na získání jednotlivých dat (jednotlivých pstí)

° **Absolutní nezávislost -** pro nezávislé jevy můžeme zjednodušovat a tím zmenšit paměťovou náročnost

- mějme jevy X, Y, Z a víme že X nezávisí na Y a Z (např. počasí nezávisí na tom, co jsme si dali ke snídani a jestli jsme šli ven, ale naopak to záviset může)
- P(X, Y, Z) = P(X | Y, Z) * P(Y, Z) … Bayesovo pravidlo
- P(X | Y, Z) = P(X) … díky tomu, že X na Y ani Z nezávisí
- P(X, Y, Z) = P(X) * P(Y, Z) … toto má v paměti menší velikost

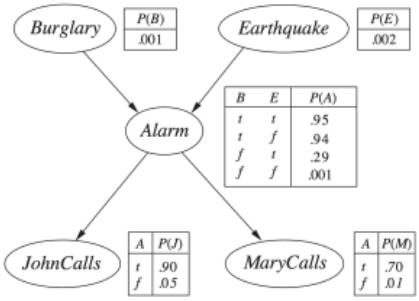
° **Podmíněná nezávislost** – některé jevy mohou být nezávislé, pokud k nim máme důkaz (tzn. taky můžeme zmenšit tabulku)

- Proměnné X a Y jsou nezávislé za podmínky Z, pokud platí něco z následujícího
  - P(X | Y, Z) = P(X | Z)
  - P(Y | X, Z) = P(Y | Z)
  - P(X, Y | Z) = P(X | Z) * P(Y | Z)
- P (X, Y, Z) = P(X | Y, Z) * P(Y | Z) * P(Z) = P(X | Z) * P(Y | Z) * P(Z) … v některých případech tohle zmenší paměťovou náročnost

11.3 Bayesovské sítě

- Slouží ke kompaktní reprezentaci
- Bayesovská síť je orientovaný nezávislý graf (DAG):
  - Uzly = náhodné proměnné
  - Hrany = závislosti mezi uzly

  - Pro každý uzel X_i máme tabulku pravděpodobností $P(X_i \mid Parents(X_i))$ … závislost jen na přímých rodičích (tzn. jen na uzlech ze kterých do daného uzlu vede hrana)



Příklad:

- Na domě máme alarm, který se rozezní v případě vloupání nebo zemětřesení
- Sousedi John a Mary nám zavolají, pokud slyší alarm

- Znění alarmu závisí na vloupání a zemětřesení, a ne na tom, zda Mary a John volají,…
- Pro každý uzel příslušná tabulka

- P(J, M, A, B, E) = P(J | M, A, B, E) * P(M | A, B, E) * P(A | B, E) * P(B | E) * P(E)
  - ⇨ P(J, M, A, B, E) = P(J | A) * P(M | A) * P(A | B, E) * P(B) * P(E)
  - Takové seřazení (takové, že ty psti máme v tabulkách) vždy najdeme, protože graf je acyklický, tzn. existuje topologické uspořádání

- Hrany neznamenají kauzalitu, ale pravděpodobnostní uspořádání => mohli bychom klidně udělat hrany jinak (např. pst že Mary volá zvyšuje pravděpodobnost, že je zemětřesení atd.)

INFERENCE V BAYESOVSKÝCH SÍTÍCH

- **Úkol**: Jak spočítat posteriorní pst distribuci přes sadu proměnných (dotaz) X, za předpokladu že máme hodnoty nějakých evidenčních proměnných (E = e). Y je set všech proměnných, které nejsou v X ani E

$$P(\mathbf{X}|\mathbf{e}) = \frac{P(\mathbf{X},\mathbf{e})}{P(\mathbf{e})} = \alpha P(\mathbf{X},\mathbf{e}) = \alpha \sum_{\mathbf{y}\in\mathbf{Y}} P(\mathbf{X},\mathbf{e},y),$$

Příklad:

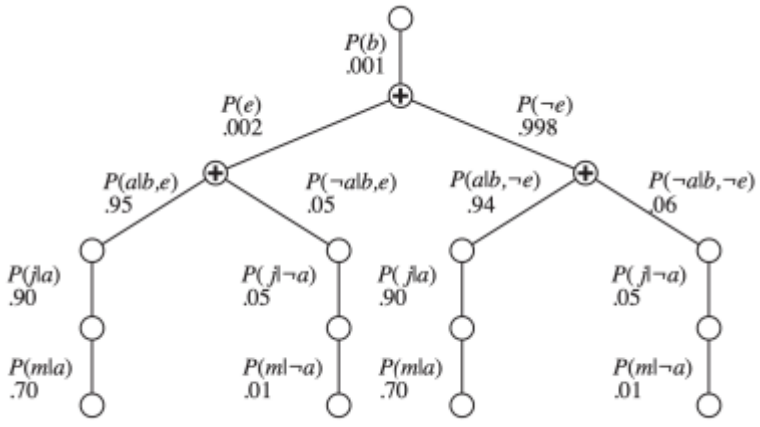
- X = {vloupání}, E = {maryVolá, johnVolá}, Y = {Alarm, Zemětřesení}

$$P(B|j,m) = \alpha P(B,j,m) = \alpha \sum_{e\in E}\sum_{a\in A} P(B,j,m,e,a)$$

For $Burglary = true$

$$P(b|j,m) = \alpha \sum_{e\in E}\sum_{a\in A} P(b)P(e)P(a|b,e)P(j|a)P(m|a) = \alpha P(b)\sum_{e\in E} P(e)\sum_{a\in A} P(a|b,e)P(j|a)P(m|a)$$

- Výpočet můžeme vizualizovat stromem – některé podstromy jsou identické, tzn. stačí je spočítat jednou a tím snížit výpočetní náročnost



Naivní Bayesovský klasifikátor

= speciální typ BN, používá se pro klasifikaci spamů a hamů

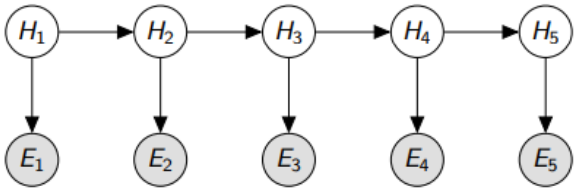
° **w** jsou jednotlivá slova vyskytující se v text

$$P(spam|\mathbf{w}) = \alpha P(spam) \prod_{i=1}^{n} P(w_i|spam)$$

Skrytý Markovův model

= speciální případ dynamické BN, používá se např. pro lokalizaci robota, rozpoznávání řeči

An unobservable true position H_t in time step t. Noisy sensor reading about the position E_t.

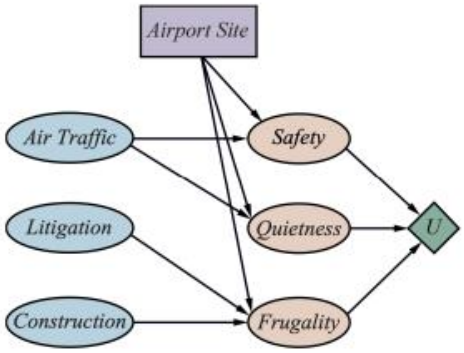


- šedé jsou pozorovatelné proměnné
- v každém kroku získáme čtení
- ptáme se jaká je pravděpodobnost reálné pozice robota v kroku 5 za evidence přechozích 5 pozorování

11.4 Rozhodovací sítě

= diagramy, které kombinují BN s dalšími akcemi a utilitami

- utilita je kosočtverec
- akce je obdélník



° Ptáme se jakou utilitu máme, pro jakou akci

° Počítá se podobně jako BN (ale to moc nebylo vysvětleno)

12. Sekvenční rozhodování s částečným pozorováním (POMDPs) I

- Realističtější – ve skutečnosti máme často jen částečnou informaci
- Používá se např. v robotice k lokalizaci robota – senzory atd. jsou zašuměné, pohyb je trochu nepřesný,… tzn. robot neví přesně kde je
- Bezpečnostní scénáře – předpokládáme, že oponent má zafixovanou strategii, takže ho můžeme brát jako část prostředí, ale nevíme úplně jistě, co dělá

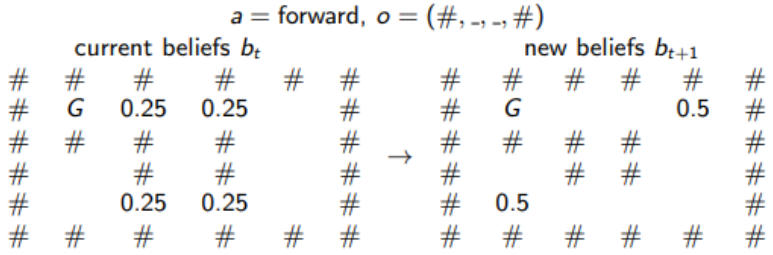
DEFINICE POMDPs

- Konečné množiny: stavy S, odměny R, akce A
- Agent interaguje s prostředím v diskrétních krocích t = 0, 1, …
  - V každém kroku obdrží pozorování o z množiny O (teď budeme předpokládat deterministická pozorování, tzn. jejich pst bude vždy 1 nebo 0, ale jinak obecně bychom mohli předpokládat i např. nepřesná měření a tak)
  - V každém kroku má **belief** (= pst distribuce přes stavy – vyjadřuje subjektivní pst, že agent je v daném stavu)
    - Z iniciálního beliefu na základě pozorování si mění v každém kroku svůj belief

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

Příklad: Bludiště

- Příklad je zadefinovaný už někde na začátku u POMDPs, tak to platí pořád stejně



12.1 Optimální chování v POMDPs

- Jak určit optimální akci, když nevíme, kde jsme?
  - value funkce závisí i na hodnotě beliefu (pro každý stav je optimální jiná akce a my nevíme, kde jsme)
- **Value funkce** pro policy π

$$v_\pi : \Delta(\mathcal{S}) \to \mathbb{R}$$

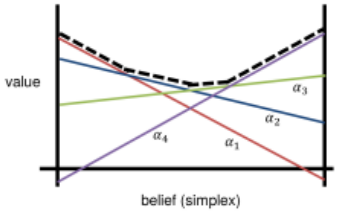
- Bellmanova rovnice pro POMDP

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

  - b je spojitá veličina => výpočet v této formě není praktický
    - Můžeme to chápat jako **MDP se spojitou množinou stavů** (tzn. je jich nekonečně mnoho) -> tzn. nemůžeme použít iterační metodu

α-vektory

- V každém stavu chceme nalézt nejlepší akci – **počet akcí** je **konečný**
  - Hledáme, kde všude je akce nejlepší, jakou má očekávanou hodnotu v jednotlivých beliefech
- Zafixujeme akci a, kterou chceme zkoumat a měníme jen belief point – value funkce (funkce očekávaného zisku) je **lineární** v závislosti na beliefu
  - Tyto lineární funkce = **α-vektory**
- Nejlepší akci pro každou funkci najdeme maximalizací přes α-vektory



$$v(b) = \max_\alpha \sum_{s \in \mathcal{S}} \alpha(s) \cdot b(s)$$

    - α(s) … hodnota lineární funkce ve stavu s
  - Toto je jen naše aproximace optimální value funkce
  - (Obrázky jsou 2D jen pro jednoduchost, jinak belief může mít větší dimenzionalitu, tohle je totiž jen belief mezi 2 stavy, ale mi můžeme uvažovat víc stavů)
- α-vektory reprezentují očekávanou hodnoty pro nějakou policy

Výpočet α-vektorů

- Výpočet hodnotové funkce pro belief b

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

- Jak vygenerovat potřebné α-vektory pro iteraci v_{t+1} (to je prý nahrazení toho vzorečku nahoře) – v každém kroku iterace generujeme nové vektory, které odpovídají n-krokové policy
  1. Předpokládejme, že máme sadu α-vektorů α‘ pro krok t
  2. V kroku t+1 vybereme akci - pak na základě pozorování pokračujeme s policy odpovídající α-vektoru z v_t (různá pozorování vedou k různým hodnotám belief funkce)
     - Např. vybereme akci a3:
       i. Pokud obdržíme pozorování o_2, použijeme hodnotu α'_4
       ii. Pokud obdržíme pozorování o_1, použijeme hodnotu α'_2

- Při plné iteraci se v každém kroku generuje $|V_t| = |A| * |V_{t-1}|^{|O|}$
  - V_t konverguje k optimální value funkci
- Tzn. generovalo by se extrémně moc α-vektorů
  - Není potřeba generovat všechny – většiny belief stavů není vůbec možné dosáhnout
  - **Řešení** = nasamplujeme belief prostor a pro každý belief point si udržujeme jen 1 nejlepší α-vektor

13. POMDPSs II

13.1 Point-based value iteration

Point-based updates a point-based value iteration

- Mějme množinu belief points … B = {b^1, b^2,…}
  - α-vektory uchováváme jen pokud jsou pro nějaký belief point optimální
- Jak samplovat belief points?
  - Existuje mnoho různých strategií – např. uniformně, na základě psti dosažitelnosti atd.

Charakteristiky PBVI

- Výhody:
  - Nevzniká exponenciální množství α-vektorů
  - Praktický způsob, jak řešit POMDPs
- Nevýhody:
  - Nemáme garantováno, jak blízko jsme k optimálnímu řešení
  - Množina belief pointů by měla být v průběhu updatována

13.2 Heuristic Search Value Iteration (HSVI)

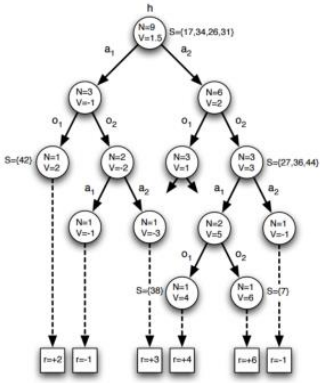
- Algoritmus aproximuje value funkci pomocí 2 aproximativních value funkcí (když budou dost blízko sebe, můžeme říct, že jsme dostatečně blízko optimálnímu řešení)
  - Funkce - dolní odhad … množina α-vektorů odpovídajícím policies o nekonečně mnoha krocích
    - Nekonečně mnoho kroků, protože za pohyby atd. jsou záporné odměny, tzn. že to bude lower bound
  - Funkce – horní odhad …množina bodů nadhodnocujících hodnotu pro každý belief point

Algoritmus

1. Inicializujeme lower a upper bound aproximující funkce
   a. Lower bound – výběr nějaké akce v každém belief-pointu donekonečna
   b. Upper bound – řešíme zjednodušení problému => vyřešíme MDP pro každý stav z S
2. Vybíráme belief points
   a. Algoritmus exploruje nejlepší akce na základě upper bound
3. Point-based update obou aproximujících funkcí
   a. Lower bound – prostě jako v PBVI, ale dostaneme konkrétní belief point
   b. Upper bound – chceme konvexní obal bodů a pak použijeme PBVI

13.3 POMCP – Monte Carlo pro POMDPs

- **Jak postavit strom?**
  - Místo stavů používáme **action-observation**
    - Máme nějaký základní belief
    - Akce vede k několika možným observacím (s pst distribucí)
    - Observace updatuje belief
- Problém – update belief pointů může být příliš náročný – řešení = **particle filtering**
  - Spustíme K náhodných pokusů (s náhodným výběrem pravdivého stavu, aplikací akce a zjištěním následného stavu)
  - Takto aproximujeme další belief pointy



HSVI vs POMCP

- HSVI (exact/approximate algoritmus) algoritmus je lepší pro malé problémy
- HSVI špatně škáluje pro větší domény
- POMCP (sampling-based algoritmus) zvládá i velké problémy, ale často potřeba vylepšení jako particle filtering

- Na POMDPs se dá použít i Reinforcement learning

13.4 Hry s nedokonalou informovaností

- Rozdíly oproti single-agent nebo perfect-information problémům
  - Aby hráč hrál optimálně, musí občas vybírat náhodné akce (musí umět dobře randomizovat, aby druhý hráč nemohl přesně odhadnout strategii)
- Náš belief neovlivňují jen naše akce, ale i akce protivníka – tzn. míra nejistoty se zvyšuje
- RL jde použít u her, kde není tak velká a důležitá randomizace

Deepstack

- Algoritmus pro hraní pokeru
- Klíčové komponenty
  - Místo jednoho belief pointu jednoho hráče, algoritmus uvažuje o všech možných stavech, které může kdokoliv z hráčů považovat za možné
  - Pro prohledávání je potřeba mnohem komplexnější pravděpodobnosti
  - Místo evaluace jednoho stavu, evaluujeme množinu možných stavů (s pravděpodobnostmi)
- Proč zrovna poker jde?
  - Je tam omezená míra neznámé informace (protihráčovy karty)
  - V real-world hrách roste množství nejistoty exponenciálně