27. Prohledávání stavového prostoru (informované a neinformované metody, lokální prohledávání, prohledávání v nejistém prostředí, hraní sekvenčních her, CSP úlohy).

Základním přístupem k umělé inteligenci je tzv. prohledávání stavového prostoru, v rámci kterého se snažíme najít řešení nějakého problému typicky zapsáním přesného deterministického algoritmu, který se daný problém snaží řešit. Narozdíl od tohoto přístupu v metodách strojového učení je typicky řešení problému nějak odvozeno z dat, samotný způsob řešení není přímo explicitně programován. Formálně můžeme problém definovat pomocí následujících složek:

- počátečního stavu
- množiny akcí, které je z jednotlivých stavů možné provést
- přechodovou relací určující pro každý stav a akci, jaký je stav následující
- test na dosažení cíle, resp. množinu cílových stavů
- volitelně také cenu akcí

Řešením problému je pak posloupnost akcí, kterou když agent z počátečního stavu provede, dostane se do cílového stavu (to platí samozřejmě za předpokladu, že agent pracuje ve známém, pozorovatelném a deterministickém prostředí – toto bude rozebráno později).

Algoritmy prohledávání stavového prostoru můžeme klasifikovat podle několika základních kritérií:

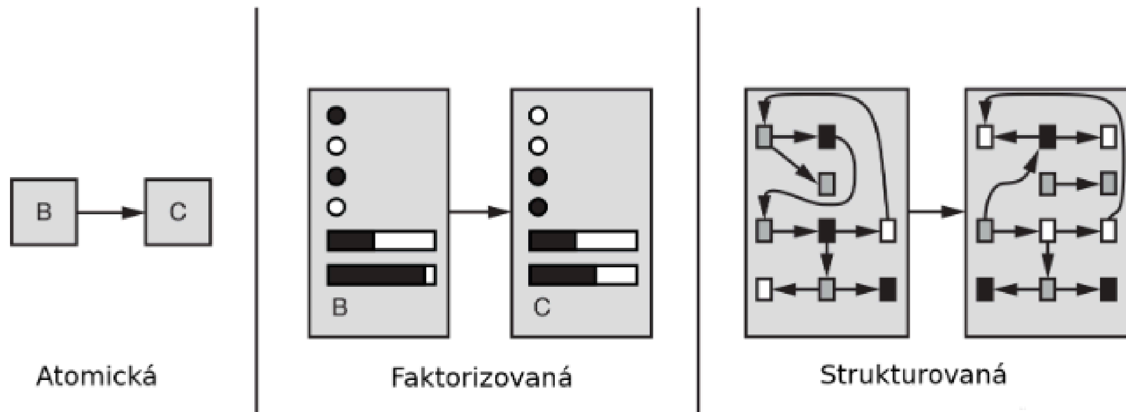
- Úplnost – algoritmus je úplný právě tehdy, když pokud cíl existuje, jistě jej najde
- Optimálnost – algoritmus je optimální právě tehdy, když najde nejlevnější posloupnost akcí k cíli, pokud cesta k cíli existuje
- Časová a paměťová složitost

Dále také rozlišujeme informované a neinformované metody. V rámci neinformovaných metod probíhá slepé procházení stavového prostoru, bez využití konkrétních vlastností zkoumaného problému (např. BFS, DFS). Informované metody jsou opakem, tzn. využívají známé vlastnosti zkoumaných problémů typicky pro nalezení lepšího řešení (např. A*, kde zavádíme heuristiku).

Stav v algoritmu je možné reprezentovat různě, rozlišujeme 3 základní způsoby:

- Atomické stavy – nedělitelné (například současné umístění dam na šachovnici v případě 8 queens problému v rámci základního DFS)
- Faktorizované stavy – stavy jsou rozděleny na dílčí proměnné, které mají své vlastní dílčí domény (např. u CSP, v rámci 8 queens si můžeme uchovávat pro jednotlivé sloupce (proměnné), ve kterých řádcích může být dáma)

- Strukturované stavy – v rámci stavu jsou nějaké složitější vztahy mezi dílčími objekty (např. predikátová logika):



Neinformované metody

Breadth First Search

Nové stavy/uzly k prohledání vkládáme do FIFO fronty, tj. probíhá procházení "po patrech" od počátečního uzlu. V základní verzi by tedy algoritmus mohl vypadat asi takto:

```
queue = {s0}
while queue:
    first = queue.dequeue()
    if isSolution(first):
        Vrať řešení (např. posloupnost kroků, kterou si někde ukládáme)
    for action in actions:
        queue.enqueue(action(first)) # proveď přechod akcí do nového stavu
```

Pro zamezení cyklů je potřeba sledovat již prozkoumané uzly a nepřidávat je znovu do fronty, používáme tzv. seznam/množina CLOSED pro výčet již zpracovaných uzlů. BFS je úplné i optimální (vzhledem k procházení po patrech najdeme nejbližší = nejoptimálnější řešení). Ukládání celých pater spotřebovává hodně paměti – $O(b^d)$, kde b je míra větvení a d hloubka řešení.

Depth First Search

Podobné jako BFS, ale prohledáváme do hloubky namísto po patrech. Toho dosáhneme vkládáním nových uzlů na LIFO zásobník. Může také jednoduše docházet k zacyklení, proto se také využívá seznam CLOSED. Není úplný ani optimální v nekonečně stavových prostorech (může pokračovat hlouběji a hlouběji ve větvi, která nikam nevede). V konečně stavových prostorech tohle však nevadí (je úplný, ale není optimální). Dále je možné omezit hloubku, do které se zanoří, v tom případě je úplný, pokud je řešení v dané hloubce. Spotřebovává narozdíl

od BFS výrazně méně paměti, v základní verzi O(bd), vylepšením pak je backtracking, kdy akce máme seřazené (procházíme je vždy ve stejném pořadí například pomocí iterátoru) a tak v zásobníku budeme mít vždy maximálně O(d) uzlů.

Možnou aplikací DFS s omezením hloubky je **iterativní depth-limited search (IDS)**. Principem je, že začneme od omezení na hloubku d = 1, spustíme DFS s omezením hloubky. Pokud není řešení nalezeno, zvýšíme hloubku a tento proces opakujeme. Tato metoda je úplná a optimální. Jedná se víceméně o pomalejší verzi BFS (když spustíme DFS znovu, procházíme už uzly, co jsme prošli dříve), ovšem šetrnější k paměti.

Bidirectional search

Principem je, že souběžně prohledáváme od startu i od cíle a sledujeme, jestli se nám prohledané prostory někde neprotnou. Z principu by však například DFS z obou směrů nefungovalo dobře, protože pravděpodobnost, že se prozkoumané prostory protnou, by byla minimální. Víceméně je tedy nutné, aby alespoň v jednom směru se prohledávalo ve stylu BFS.

Uniform Cost Search

Narozdíl od předchozích metod zde již zavádíme cenu akcí, přičemž předpokládáme nezápornou cenu. Stavy jsou vkládány do prioritní fronty. Jedná se víceméně o Dijkstrův algoritmus, ovšem s tím, že končí okamžitě, jak najde nějaký cíl. Je optimální a úplný, víceméně funguje podobně jako BFS, ovšem bere v úvahu ceny akcí (pokud by cena všech akcí byla stejná, pak je UCS s BFS ekvivalentní).

Informované metody

Jak bylo naznačeno výše, v rámci informovaných metod už nerozbalujeme uzly slepě, ale využíváme již nějaké vlastnosti řešené úlohy. Nejtypičtěji zavádíme nějaký odhad, kterým směrem je řešení a s jakou cenou jej můžeme dosáhnout. Zavádí se tedy funkce f(n), která se skládá ze dvou složek f(n) = g(n) + h(n). g(n) je zatím zaplacená cena k dosažení aktuálního stavu. h(n) je heuristika, tj. odhad zbývající ceny do cíle od aktuálního stavu. V rámci informovaných metod pak podobně jako v UCS vkládáme uzly do prioritní fronty, kterou řadíme podle f(n) a vybíráme uzly s nejmenší f(n). Pokud bychom nastavili h(n) = 0, pak dostáváme UCS.

Greedy search

Hladové vyhledávání, které není optimální. Úplně ignoruje zatím zaplacenou cenu, tzn. g(n) = 0 a pouze se hladově řídí heuristikou, tzn. agresivně jde za odhadem řešení. Typicky najde rychle nějaké řešení, které však nemusí být to nejlepší.

A*

V rámci A* využíváme funkci f(n) ve smyslu popsaném v úvodu této kapitoly, ovšem klademe speciální požadavky na heuristiku. Při splnění těchto vlastností je pak metoda A* úplná a optimální. Požadujeme, aby heuristika dávala optimistický odhad ceny řešení. Pokud toto heuristika nesplňuje, pak metoda není optimální. Stále využíváme prioritní frontu, jedná se tedy o variantu BFS, ovšem snažíme se zmenšit šířku větvení a jít přesněji za cílem.

Pokud máme dvě heuristiky, které jsou přípustné, tzn. obě dávají optimistický odhad, pak víme, že když jedna heuristika je větší než druhá, je přesnější, dává menší chybu, tzn. je lepší. Jestliže jedna heuristika dává lepší (vyšší) odhad než druhá, říkáme, že ji dominuje. Heuristiky lze také komponovat, pokud máme více přípustných heuristik, můžeme vždy vzít maximální z nich pro dosažení nejpřesnějšího odhadu.

Prostředí

Zatím jsme uvažovali prostředí, které známe, je plně pozorovatelné (víme, ve kterém stavu přesně jsme) a je deterministické, tzn. víme, do jakého stavu nás akce přivede. Existuje ale mnoho problémů, kdy to tak není, podívejme se nyní na různé varianty.

Částečně pozorovatelné prostředí

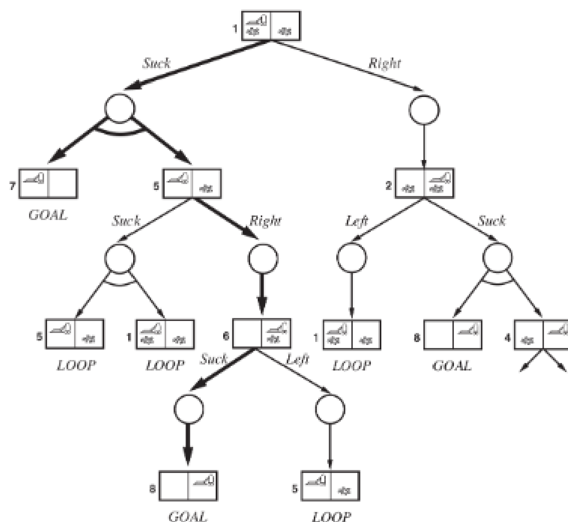
Pokud prostředí není plně pozorovatelné, hovoříme o částečně pozorovatelném prostředí, v extrémním případě nemusíme mít žádná pozorování. V tomto případě získává agent informace o stavu prostřednictvím senzorů. Pro řešení takových problémů se používají tzv. **belief stavy.** Belief stav je podmnožina všech možných stavů, ve kterých agent v daný moment může být. Provedením akce a získáním nového pozorování senzory si pak agent může zpřesnit svůj belief stav. Protože belief stav je podmnožinou stavů, je belief stavů celkově exponenciálně mnoho (jedná se o potenční množinu), což není explicitně vyčíslitelné pro větší problémy, je tedy nutné belief stavy budovat inkrementálně.

Nedeterministické prostředí

V nedeterministickém prostředí není výsledkem akce jediný stav, ale množina stavů, agent tedy výsledek nezná dopředu – musí akci provést a podívat se na výsledek (potřebuje nějaké senzory, aby zjistil, jak akce dopadla). Řešení problému už tedy není sekvence akcí, ale plán ve formě stromu if-then. Tento plán je možné reprezentovat pomocí AND-OR stromu:

- OR uzel reprezentuje možnost volby, stačí vyřešit pouze jeden z podproblémů
- AND uzel reprezentuje nutnost být schopen vyřešit všechny podproblémy

V plánu pro řešení v nedeterministickém prostředí se typicky po vrstvách střídají OR a AND uzly – začínáme OR uzlem (můžeme si vybrat akci), ovšem musíme být schopni řešit všechny stavy, do kterých nás akce může dostat (AND uzel):



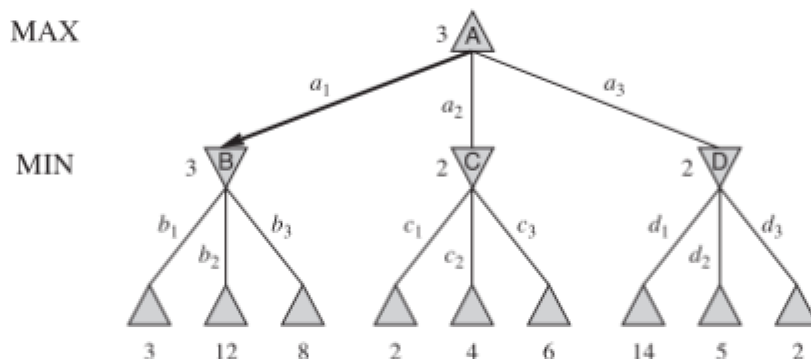
Podproblémy se mohou opakovat, proto je vhodné si zapamatovat řešení -> dynamické programování. Může také nastat cyklus, který je vhodné ignorovat a pokračovat jinou větví.

Hraní her

V rámci hraní her řešíme konfliktní situaci dvou hráčů, kdy každý se snaží maximalizovat svůj zisk (vyhrát) a my chceme zanalyzovat, jak hra proběhne. V rámci této otázky ze SUI se zaměříme pouze na tahové hry dvou hráčů s nulovým součtem (zisk jednoho hráče je ztrátou druhého hráče), více viz otázky 28 a 29 z THE.

Minimax

Metoda minimax staví na principu, že racionální hráč maximalizuje svůj užitek. Protože uvažujeme hry s nulovým součtem, stačí modelovat zisky jednoho hráče (zisky druhého hráče jsou implicitně to stejné, ale záporně). Hru lze modelovat stromem možných tahů, kdy každá úroveň stromu odpovídá jednomu hráči:

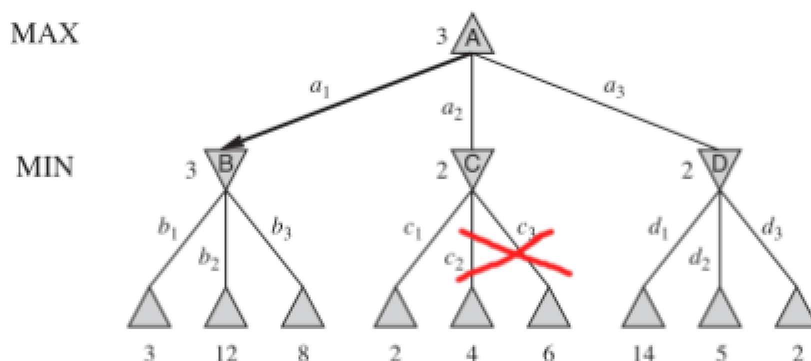


Analyzujme situaci na obrázku výše z pohledu prvního hráče, ten chce maximalizovat svůj užitek. Ví však, že druhý hráč bude minimalizovat svoji ztrátu, bude brát tedy minimum ze svých

možných tahů. Například pokud bychom zahráli akci a_2, dostaneme se do uzlu C a druhý hráč si bude chtít vybrat pro sebe nejvýhodnější variantu (minimalizuje ztrátu), vybere tedy c_1 a bude mít ztrátu 2. Lepší je tedy z pohledu prvního hráče zahrát akci a_1, protože tam nejhůře dostane zisk 3. To ovšem předpokládá racionalitu, pokud by druhý hráč nebyl racionální a hrál například náhodně (nebo dokonce "spolupracoval" a tím zvyšoval svoji ztrátu), situace je samozřejmě značně odlišná.

Alfa-beta prořezání

V rámci základní verze minimaxu se mnohdy dělá zbytečné prohledávání stavového prostoru, i když už je jasné, že daná větev nepovede ke zlepšení. Ve výše uvedeném příkladu jakmile zanalyzujeme akci a_1 a zjístíme, že můžeme dostat zisk 3, pak při analýze akce a_2 když zjistíme, že druhý hráč může zahrát c_1 a zaručit si ztrátu pouze 2, víme, že je zbytečné již dále tento podstrom analyzovat, protože druhý hráč s jistotou může zahrát akci se ziskem pouze 2, což ale pro nás není zajímavé, protože již máme možnost získat zisk 3, můžeme tedy část stromu prořezat:



Další prořezávání

Kromě výše zmíněné alfa-beta optimalizace minimaxu (tyto dvě metody dávají vždy stejné řešení, ovšem alfabeta jej typicky najde rychleji díky prořezávání) můžeme dělat další prořezávání, což už ovšem vede na ztrátu optimálnosti, protože se využívá různých heuristik, které nemusí být nutně přesně. Můžeme například omezit hloubku zanořování nebo provádět dopředné prořezávání podle heuristiky a brát jen několik málo nejzajímavějších potomků.

Hry s náhodou

Pokud hra není plně deterministická a například jeden hráče hraje zcela náhodně, musíme brát v potaz jeho možnost a s jakou pravděpodobností je hraje. Namísto ohodnocení uzlu metodou minimax pak třeba spočítáme očekávaný užitek, tzn. váhujeme zisky pravděpodobností, že takový zisk dostaneme.

Lokální prohledávání

Dosud jsme uvažovali pouze metody globálního prohledávání (BFS, DFS, IDS, A*, …), jejichž výstupem je cesta k cíli (řešení problému). Jejich problémem často byla vysoká spotřeba paměti. To se snaží řešit metody lokálního prohledávání, kdy se díváme pouze na nějaký malý podprostor lokálně a v něm hledáme nejlepší řešení. Výstupem tedy už není posloupnost akcí, ale stav (řešení), snažíme se maximalizovat kvalitu stavu.

Hill-climb

Základní přístup analogický k hladovému prohledávání. Udržujeme v paměti jediný stav, nahrazujeme jej nejlepším sousedem. Pokud žádný soused není lepší, končí a vrací současný stav. Problémem je, že trpí na lokální extrémy, tzn. se může zaseknout v lokálním maximu, přestože někde je výrazně lepší řešení. Možným řešením tohoto problému je opakované restartování s různými počátečními stavy, abychom pokryli větší podprostor.

Local Beam Search

Paralelní hill-climbing, udržujeme konstantní počet nejlepších stavů (označme k) více tedy diverzifikuje řešení a netrpí tolik na lokální extrémy. Jsou různé varianty:

- základní, kdy následníky vybíráme ze všech sousedů (vybereme k nejlepších)
- vybíráme k nejlepších, ale omezíme maximální počet lokálních sousedů
- stochastický výběr k sousedů

Simulované žíhání

Rozšiřuje hill-climbing o princip, že lepšího souseda bereme vždy, ale pokud je horší, bereme jej pouze s nějakou pravděpodobností, která v průběhu běhu algoritmu klesá. Pro současný stav s a souseda s' spočítáme $\Delta E = E(s') - E(s)$ a horšího přijmeme s pravděpodobností $e^{-\Delta E/T}$, kde T je současná teplota, která se průběžně zmenšuje.

Genetické algoritmy

Algoritmy inspirované přírodou a principem přirozeného výběru (nejsilnější přežije). Máme nějakou populaci, jedinci mají nějaké smysluplné kódování genů, algoritmus provede konstantní počet iterací, v rámci kterých provede následující:

1. Vybere rodiče (typicky členům populace přiřazuje fitness funkci popisující kvalitu řešení, výběr se pak provádí například náhodným turnajem na základě fitness funkce)
2. Zkříží rodiče pro vytvoření potomků
3. Zmutuje s nějakou pravděpodobností potomky (větší diverzifikace řešení)

Constraint Satisfaction Problems

Jak bylo naznačeno v rámci části o reprezentaci stavů, některé problémy je možné prakticky reprezentovat s pomocí proměnných X_i a domén proměnných D_i. Následně pak na proměnné a

jejich hodnoty z domén můžeme klást omezení. Například omezením v rámci problému 8 dam je, že se dámy nesmí ohrožovat, umístění jedné dámy tedy omezí doménu ostatních proměnných (dam v dalších sloupcích). Podobně pokud chceme provést barvení grafu a přiřadíme barvu jednomu uzlu, vyloučíme tuto barvu z možných barev všech sousedů. Řešení takových problémů se říká Constraint Satisfaction Problem, kdy chceme najít přiřazení proměnným, které je konzistentní se všemi podmínkami.

Rozlišujeme následující druhy konzistence uzlů:

- unární – proměnná X_i je konzistentní v uzlu, pokud žádná hodnota v D_i neporušuje žádnou unární podmínku
- binární (konzistence hran) – X_i je hranově konzistentní s X_j pokud pro každou $v_i \in D_i$ existuje $v_j \in D_j$ taková, že přiřazení $X_i = v_i, X_j = v_j$ neporušuje žádnou podmínku
- podobně pak dále obecně K-konzistence (např. ternární)

Základním přístupem k řešení problémů CSP je DFS (backtracking), kdy však v průběhu při přiřazení hodnoty nějaké proměnné vynutíme konzistenci s podmínkami, tzn. prořežeme domény ostatním proměnným. Například pokud máme jako omezení soustavu rovnic, kdy možnou doménou je množina {1, 2, …, 9} a máme podmínku $X_1 \cdot X_1 = X_2$ můžeme již na začátku prořezat doménu D_2 na {1, 4, 9}. Pokud pak přiřadíme v rámci backtrackingu například $X_1 = 1$, provedeme prořezání ostatních domén a dojdeme také k $X_2 = 1$. Obecně tento princip prořezávání popisuje algoritmus AC3, který slouží k vynucení hranové (binární) konzistence a pracuje v kubické složitosti vzhledem k velikosti domén.

Řešení s pomocí backtrackingu lze obecně popsat takto:

1. Vyber si volnou (nepřiřazenou) proměnnou
2. Pro každou její hodnotu:
    a. Zkus, jestli je konzistentní, pokud ano, přiřaď ji a
    b. zuž domény s pomocí inference
    c. zavolej se znova s rozšířeným přiřazením a zúženými doménami, v případě neúspěchu zkus další hodnotu. Pokud tento krok byl úspěšný, vrať řešení
3. Pokud žádná hodnota nevedla k úspěchu, úloha nemá řešení splňující všechna omezení

Jednotlivé kroky lze implementovat s pomocí různých heuristik. Například v kroku 1 můžeme vybírat proměnnou, která je nejvíce omezena, tzn. má nejméně možných hodnot. Nebo naopak hodnoty v kroku 2 můžeme procházet v nějakém pořadí, typicky například zkusit přiřadit nejméně omezující hodnotu.

Backtracking se vždy vrací k poslední přiřazené proměnné, která ovšem nemusela mít na selhání zásadní vliv. Jednou z možných optimalizací je tedy určit množinu proměnných, které způsobují konflikt a vracet se až k přiřazení do těchto proměnných.

Min-conflict

Pracuje na principu, že vytvoříme nějaké náhodné řešení problému, které ovšem není úplně korektní. Poté vybereme nějakou proměnnou a nastavíme ji tak, abychom minimalizovali počet konfliktů (= chyb v řešení). Opakujeme, dokud všechny konflikty neodstraníme. Například v rámci 8 queens problému za konflikt považujeme dámy, které se vzájemně ohrožují.

Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele Fifinas.