Řešení problémů

Řešení problémů zde—z hlediska inteligentních agentů—znamená *jak může agent dosáhnout* ***cílů*** a jaké by měly být sekvence akcí, které mohou k daným cílům vést.

Cíl a *soubor prostředků* pro dosažení cíle je ***řešený problém***. Proces prozkoumávání prostředků (tj. co mohou prostředky, které jsou k dispozici, dosáhnout), se nazývá *vyhledávání (*či *prohledávání, pátrání)*.

Agent řešící problémy—typ agenta zaměřeného na cíle. Např. jednoduchý reflexní agent je příliš omezen (kvůli svým vlastnostem nemůže uvažovat směrem do budoucna).

Agenti řešící problémy rozhodují o své činnosti tím, že se snaží nalézt sekvence akcí, které vedou k požadovaným stavům.

Obecně se o inteligentních agentech předpokládá, že provádějí akce tak, aby prostředí procházelo posloupností stavů z hlediska maximalizace měřené výkonnosti (méně obecně: účinnosti agenta splnit vybraný cíl).

Prvním krokem k řešení problému je **formulace cíle**, založená na okamžité situaci.

Za cíl lze např. považovat soubor stavů světa—a to takové stavy, v nichž je cíl splněn. Na akce pak lze hledět jako na činnosti, které způsobí přechody mezi jednotlivými stavy světa, takže úkolem agenta je nalézt vhodné akce k dosažení cílového stavu.

Obecně nelze uvažovat o akcích na určité úrovni, např. při požadavku vyjet z parkoviště není pro agenta vhodné usuzovat o akcích typu “natoč volant 6 stupňů doleva” apod., protože generace řešení na takovéto úrovni detailů by mohla být ve většině případů nezvládnutelným problémem (člověk také neusuzuje tímto způsobem).

Formulace problému je proces rozhodnutí, jaké akce a stavy uvažovat; tato formulace následuje po formulaci cíle.

Dostatečná znalost: lze dojet z jednoho města do druhého, výběr z více cest...

Pokud není k dispozici dostatek informací a k cíli se lze dostat více možnostmi, pak agent nemá nic lepšího než náhodně zvolit.

Obecně platí, že pokud má agent několik možností neznámé hodnoty, pak k rozhodnutí o své činnosti potřebuje napřed prozkoumat *různé možné posloupnosti akcí* vedoucích do *stavů o známé hodnotě*, a pak prostě vybrat tu nejlepší sekvenci. Tento proces se nazývá **vyhledávání**.

Algoritmus vyhledávání má jako vstup řešený problém a na výstupu poskytuje **řešení** ve formě sekvence akcí. Jakmile je řešení nalezeno, je doporučeno k provedení—tzv. **výkonná fáze**.

Z toho plyne jedna z možností, jak navrhnout agenta na základě trojice pojmů "formuluj, vyhledej, proveď". Po provedení zvolené sekvence akcí pak agent zvolí nový cíl a vše se pro aktuální situaci opakuje znovu.

Jednoduchý agent řešící problém:

```
function Simple-Problem-Solving-Agent (p) returns action
     inputs: p       // vjem (percepce)
     static: s       // sekvence akcí, na počátku prázdná
             state   // popis stavu současného světa
             g       // cíl, na počátku prázdný (žádný)
             problem // formulace problému

     state ← Update-State(state, p)

     if s is empty then
          g ← Formulate-Goal(state)
          problem ← Formulate-Problem(state, g)
          s ← Search(problem)

     action ← Recommendation(s, state)
     s ← Remainder(s, state)

     return action
```

(Funkce `Update-State` a `Formulate-Goal`, tj. aktualizce stavu a formulace cíle jsou popsány dále.)

Formulace problémů

Formulace problémů závisí na množství znalostí, které má agent k dispozici, aby bylo možno uvažovat jeho akce a stavy, v nichž se ocitne. To závisí na propojení agenta s prostředím pomocí jeho vjemů a akcí.

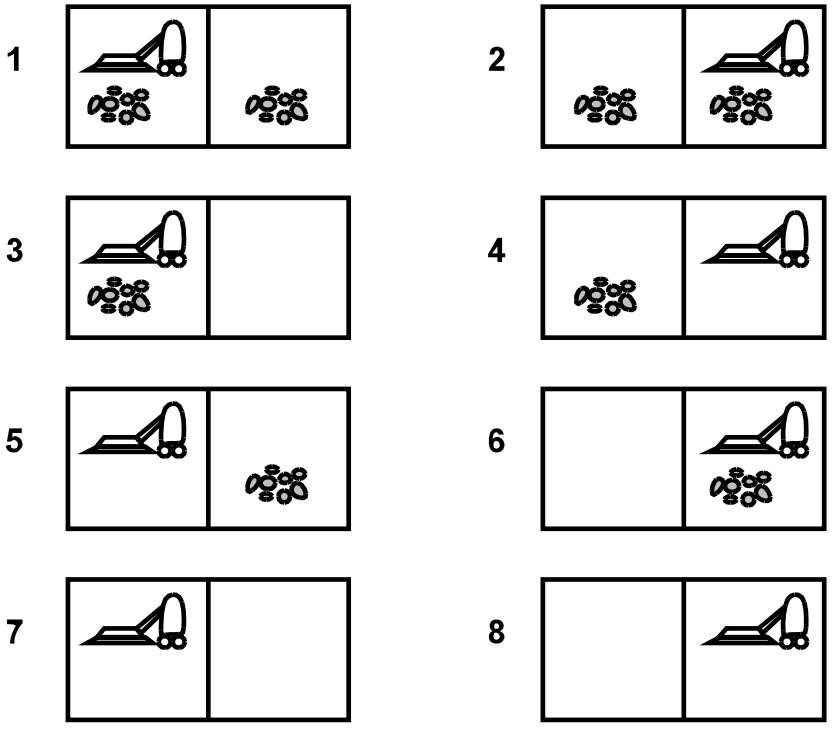
Existují čtyři základní různé typy problémů:

- problémy pro jeden stav,
- problémy pro více stavů,
- problémy nahodilé, a
- problémy k prozkoumání.

Typy problémů

Uvažme opět svět vysávání nečistoty: nechť pro jednoduchost obsahuje pouze dvě místa. Každé z míst může nebo nemusí obsahovat nečistotu. Agent se může nacházet v jednom z míst. Existuje tedy 8 možných stavů světa. Agent má k dispozici tři možné akce (v této verzi světa): *Doleva, Doprava, Vysávat*. Dále předpokládejme, že vysávání je účinné na 100%. Cílem je zlikvidovat veškerou nečistotu, tj. cíl je ekvivalentní s dosažením množiny stavů {7, 8}.

Osm možných stavů zjednodušeného světa vysávání nečistot:



Předpokládejme dále, že agentovy sensory mu poskytují dostatek informace k tomu, aby přesně věděl, v jakém stavu se nachází (tj. svět je přístupný) a dále, že agent ví, co každá jeho akce udělá. Pak je možné exaktně spočítat, v jakém stavu bude po jakékoliv sekvenci akcí.

Pokud je např. počáteční stav 5, pak lze určit, že po sekvenci akcí [*Doprava, Vysávat*] bude dosaženo cílového stavu. To je nejjednodušší případ, zvaný **problém jednoho stavu**.

Uvažujme dále agenta, který zná všechny důsledky svých akcí, ale má omezený přístup ke stavům světa (např. v extrémním případě nemusí mít žádné sensory). V takovém případě jediné, co agent ví, je, že jeho počátečním stavem je jeden z množiny stavů {1, 2, 3, 4, 5, 6, 7, 8}. Kupodivu agentova kritická situace není beznadějná—protože ví, k čemu po jeho akcích dochází, pak může určit, že např. po akci *Doprava* se ocitne v jednom ze stavů {2, 4, 6, 8}. Ve skutečnosti může agent odhalit, že po akční sekvenci [*Doprava, Vysávat, Doleva, Vysávat*] je zaručeno dosažení cíle bez ohledu na počáteční stav.

Shrnutí: Pokud není svět plně přístupný, pak agent musí uvažovat o množině stavů, do nichž se může dostat, nikoliv o jednotlivých stavech. Uvedená situace se nazývá **vícestavový problém**.

Případ neznalosti důsledku akcí lze řešit obdobným způsobem. Např. uvažujme situaci, kdy prostředí se jeví nedeterministické (a splňuje Murphyho pravidlo): akce zvaná *Vysávání* někdy umístí nečistotu na koberec, *ale pouze v tom případě, že tam ještě není žádná nečistota*.

Pokud agent tedy ví, že se nachází např. ve stavu 4, pak také ví, že bude-li vysávat, dosáhne jednoho ze stavů {2, 4}. Pro jakýkoliv *známý* počáteční stav existuje posloupnost akcí, která zaručuje dosažení cílového stavu.

Někdy ovšem *neznalost zabraňuje* agentovi nalézt sekvenci zaručující dosažení cíle. Dejme tomu, že agent je ve světě Murphyho pravidla, má poziční sensor a sensor pro lokaci nečistoty, ale nemá sensor schopný detekovat nečistotu v jiných místech (čtvercích). Řekněme, že sensory poskytují informaci, že agent je v jednom ze stavů {1, 3}. Agent může formulovat sekvenci akcí [*Vysát, Doprava, Vysát*]. Vysátí změní stav na jeden z {5, 7}, pohyb doprava na jeden z {6, 8}. Je-li skutečným stavem 6, pak sekvence akcí je úspěšná; pokud 8, pak plán selhal.

Pokud by agent vybral jednodušší sekvenci akcí [*Vysát*], pak to někdy může uspět, ale ne vždy. Ukazuje se, že neexistuje žádná fixní posloupnost akcí, která by zaručila řešení problému.

V uvedeném případě má agent cestu k řešení problému z jednoho stavu {1, 3}: vysát, zatočit doprava, a pak vysát *pouze pokud tam je nečistota*. Tento postup ovšem vyžaduje možnost vnímání *během exekuční fáze*. V tomto případě musí agent spočítat celý strom akcí (nikoliv jen sekvenci pro jednu akci).

Obecně se každá z větví stromu zabývá možnou nahodilostí, která se může objevit. Proto se tato situace nazývá **problém nahodilosti**.

V realitě k těmto problémům patří velké množství situací, protože přesná predikce je nemožná—z tohoto důvodu např. lidé obvykle důkladně sledují situaci při přecházení silnice nebo při řízení.

Problémy jednoho stavu a více stavů mohou být zpracovávány podobnými vyhledávacími technikami.

Problémy s nahodilostí však vyžadují mnohem složitější algoritmy a často vedou ke změně návrhu agenta, kdy agent může provést akci *před* nalezením zaručeného plánu. To je užitečné proto, že může být snazší nebo lepší něco začít dělat a pak zjišťovat, jaké nahodilosti se objevily, ve srovnání s postupem, kdy se předem uvažuje každá možná nahodilost, která by se mohla vyskytnout během exekuční fáze.

Agent pak může pokračovat v řešení problému za předpokladu získání přídavné informace. Tento postup se nazývá **prokládání** činnosti vyhledávání a exekuce.

Dobře definované problémy a řešení

Pod pojmem **problém** zde budeme rozumět souhrn informací, které agent použije k rozhodnutí o své činnosti.

Základními prvky definice problému jsou *stavy* a *akce*.

- **Počáteční stav** – stav, o němž agent ví, že v něm je.
- **Operátor** – soubor agentových možných akcí. Termín *operátor* se používá k popisu akce, a to pomocí pojmů, jakých stavů lze docílit provedením akce z konkrétního stavu. Někdy se používá formulace **následnická funkce *S***: je-li dán konkrétní stav *x*, pak *S(x)* vrací soubor stavů dosažitelných z *x* libovolnou jednou akcí.
- **Stavový prostor** problému – soubor všech stavů dosažitelných z počátečního stavu libovolnou sekvencí akcí.
- **Cesta** – ve stavovém prostoru jde prostě o jakoukoliv sekvenci akcí, která vede z jednoho stavu do druhého.
- **Test cíle** – test aplikovaný agentem na popis jednoho z cílů, zda jde o cílový stav. Pokud je cílových stavů více, test cíle rozezná, zda se o jeden z nich jedná či nikoliv. Pozn.: někdy je cíl specifikován nějakou abstraktní vlastností spíše než explicitně vyjmenovaným seznamem stavů, např. v šachu jde o dosažení cíle *mat* (protivníkův král je napaden figurou a nemá kam ustoupit ani to napadení nelze zlikvidovat, tj. dalším tahem lze krále vzít bez ohledu na to, co oponent vlastnící daného krále udělá).
- **Funkce ceny cesty *g*** – funkce *g* přiřazující dané cestě nějaké náklady (nebo cenu). Nadále budeme uvažovat o ceně cesty jako o sumě cen za jednotlivé akce podél dané cesty.

Počáteční stav, soubor operátorů, test cíle a cena cesty dohromady definují problém. Typ dat representující problém:

```
datatype Problem
     components:   Initial-State, Operators,
                   Goal-Test, Path-Cost-Function
```

Instance uvedeného typu dat slouží jako vstup vyhledávacího algoritmu; výstupem algoritmu je **řešení**, tj. cesta z počátečního stavu do stavu vyhovujícímu cílovému testu.

Pro zpracování vícestavových problémů se provede jednoduchá modifikace: problém je složen z počátečního *souboru* stavů a dále ze *souboru* operátorů. Test cíle a funkce ceny cesty zůstávají jako předtím.

Cesta nyní propojuje *soubor* stavů a **řešením** se nyní rozumí cesta vedoucí do souboru těch stavů, které vyhovují cílovým stavům. Stavový prostor je nahrazen prostorem souboru stavů.

Měření výkonnosti řešení problémů

Tři možnosti:

1. Bylo vůbec nalezeno řešení?
2. Je řešení dobré (tj. cesta s nízkou cenou)?
3. Jaká byla **cena** za nalezené **řešení** vzhledem k požadované paměti a času?

Celková cena vyhledávání je tvořena součtem ceny cesty a ceny vyhledávání (v robotice se cena vyhledávání, tj. části prováděné před interakcí s prostředím, nazývá **offline cena** a cena cesty **online cena**).

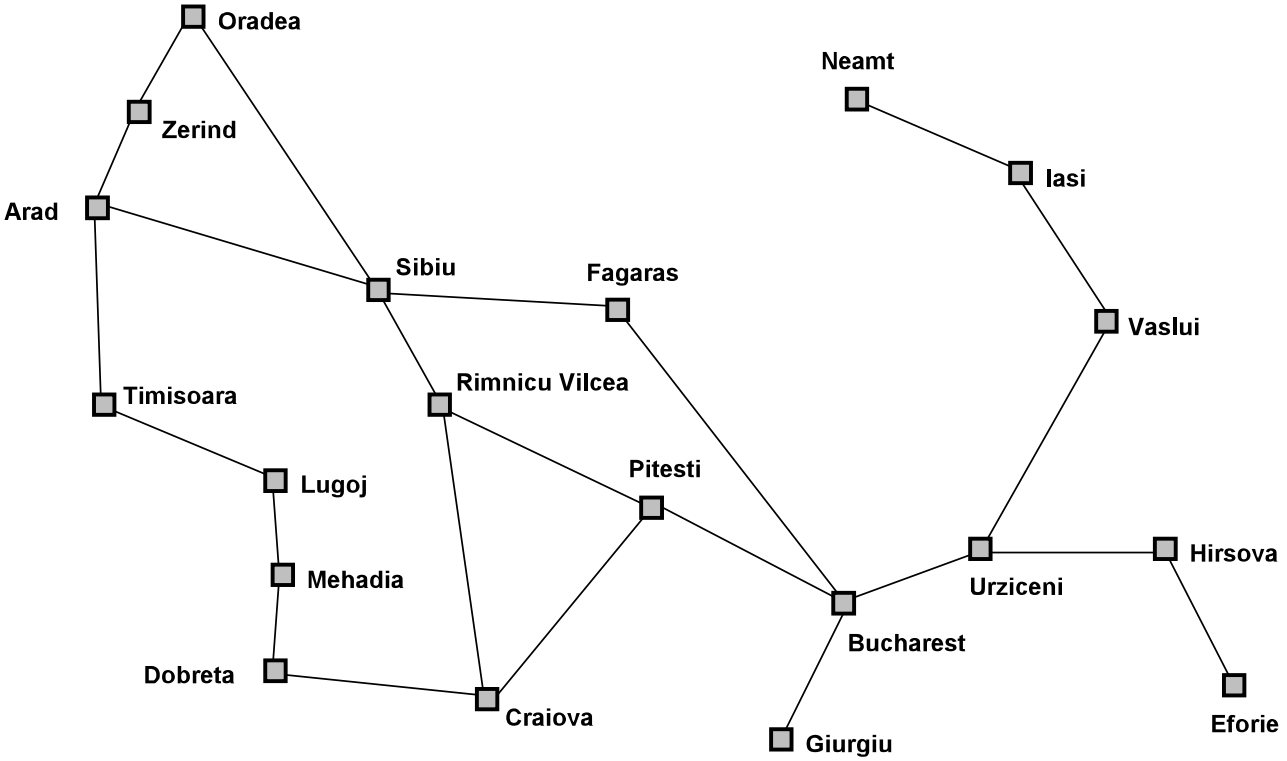
Např. cesta z města **A** do města **B**: cena cesty může být úměrná počtu kilometrů plus náklady za opotřebení na různých typech silnic.

Cena vyhledávání závisí na okolnostech:

- Ve statickém prostředí je nulová (nezávislost na čase).
- Je-li nějaká urgentní potřeba dojet do **B**, pak je prostředí semidynamické, protože delší uvažování o cestě cenu zvýší (můžeme uvažovat o přibližně lineární závislosti na době výběru, přinejmenším pro kratší časové úseky). Zde je tedy nutno dát dohromady kilometry a sekundy (cenu cesty a cenu vyhledávání, což není obecně snadný problém. Navíc pro velmi malé stavové prostory je snadné najít řešení s minimální cenou cesty. Pro rozsáhlé a komplikované problémy je nutno nalézt kompromis.

Výběr stavů a akcí

Příklad: problém dojetí z města **Arad** do města **Bucharest**, přičemž k dispozici je mapa cest:



Odpovídající stavový prostor má 20 stavů (měst), každý stav je definován pouze polohou specifikovanou jako *město*. **Počáteční stav** je tedy "v Aradu", **test cíle** je "je to Bucharest"? **Operátory** odpovídají jízdě po silnicích mezi městy.

Jedno možné řešení: **Arad→Sibiu→Rimnicu Vilcea→Pitesti→Bucharest**.

(Je mnoho dalších řešení.) K rozhodnutí o tom, které řešení je lepší, je nutno znát, co měří funkce ceny: *celkovou délku cesty* nebo *celkový čas cesty*?

Daná mapa neobsahuje ani jedno, takže se použije počet kroků (průjezdních silnic mezi městy): to je 3 přes **Sibiu** a **Fagaras**, tedy nejlepší řešení.

Určitý problém řešení tkví v popisu stavů a operátorů. Pro srovnání je dobré porovnat jednoduchý popis *v Aradu* se skutečnou jízdou přes zemi, kdy skutečný stav světa obsahuje velké množství věcí. Řada věcí může být jako irelevantní zcela vynechána—tento způsob popisu se nazývá **abstrakce**.

Abstrakce popisu stavů jsou doprovázeny abstrakcemi v popisu akcí (řízení auta má mnoho akcí, nejen např. změnu pozice, kterou v našem případě bereme do úvahy jako jedinou věc.

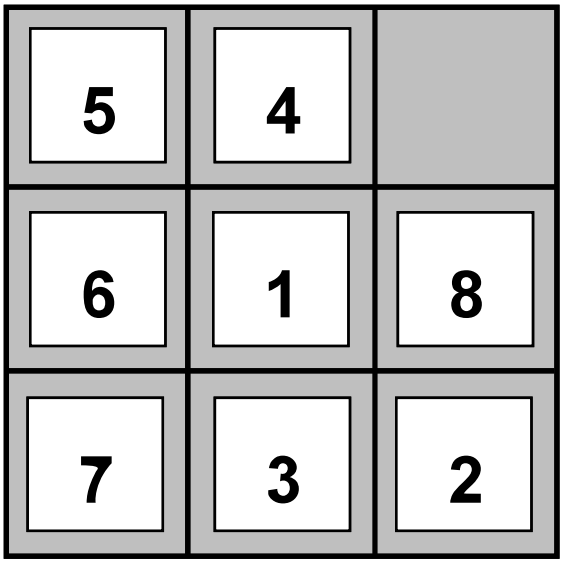
Příklady problémů

Obvykle je vhodné uvažovat o **problémech her** a o **problémech reálných** (které jsou opravdové, avšak obvykle mnohem obtížnější pro hledání řešení).

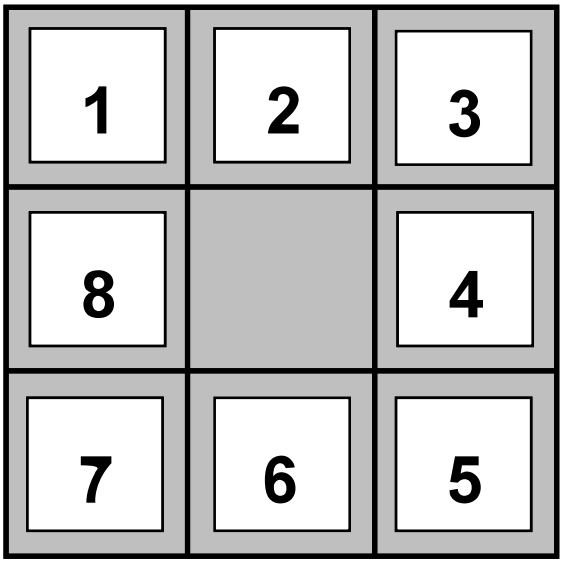
U problémů her je ale snadnější vytvořit jasný a exaktní popis, takže mohou být snadněji použity pro srovnání různých metod apod.

Problémy her

8-hlavolam



Start State



Goal State

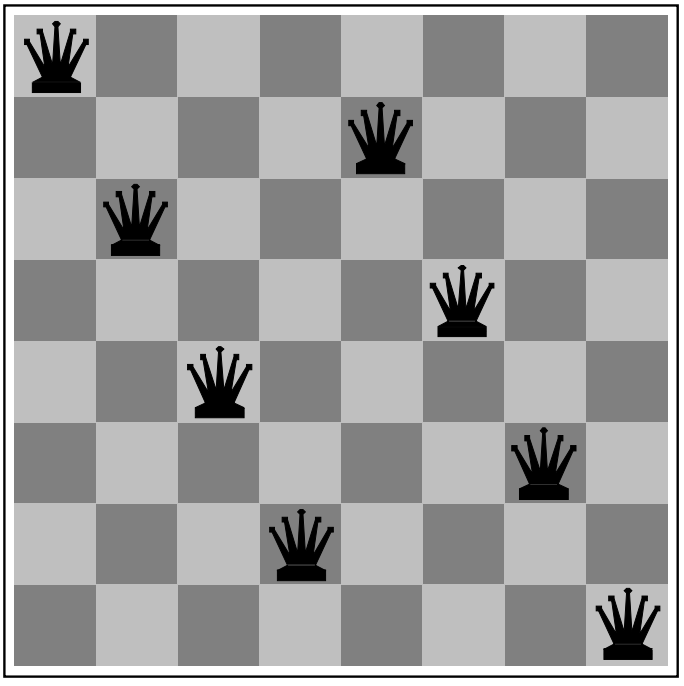
Při řešení problému seřazení posuvných kosteček je vhodné použít určité "triky", které řešení usnadní. Příklad: místo použití operátorů typu *přesun kostky s číslem 4 na volnou pozici* je mnohem rozumnější použít operátory typu *volná pozice mění místo s kostkou nalevo*, apod. (vede to k menšímu počtu operátorů...).

Lze vytvořit následující formulaci:

- **Stavy:** popis stavu specifikuje umístění každé z osmi kosteček na jednom z devíti polí (je užitečné zahrnout i pozici volného místa).
- **Operátory:** volné místo se pohybuje doleva, doprava, nahoru, dolů.
- **Test cíle:** stav se shoduje s cílovou konfigurací na obrázku.
- **Cena cesty:** každý krok stojí 1, takže cesta stojí totéž, co je její délka.

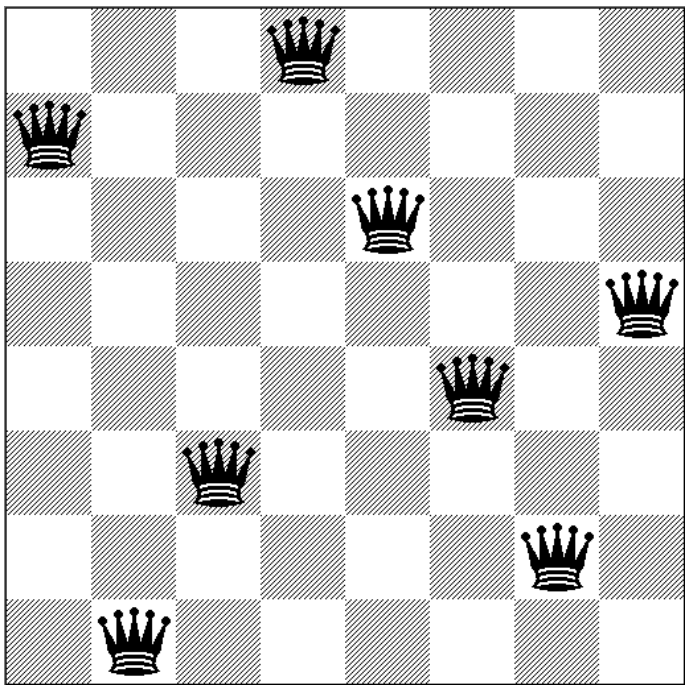
8-hlavolam patří ke skupině úloh přemísťování bloků, což obecně jsou NP-kompletní problémy, kde nelze očekávat, že bude nalezeno řešení lepší než pomocí vyhledávacího algoritmu. Stejná, ale složitější úloha 15-hlavolam, se spolu s 8-hlavolamem používá často k testování nových vyhledávacích algoritmů v umělé inteligenci.

Problém 8 dam



Cílem je, aby na šachovnici bylo umístěno 8 dam, aniž by žádná z nich napadala libovolné ostatní. Na předchozím obrázku cíl splněn není.

Příklad jednoho z mnoha splnitelných cílů:



Pro tento problém existují efektivní řešení, ale přesto se používá pro hledání a testování nových vyhledávacích algoritmů.

Existují dvě hlavní skupiny formulací problému:

- inkrementální (umísťování dam po jedné), a
- kompletní stav (začíná se s 8 dámami a mění se jejich pozice).

Zde nehraje roli cena cesty (počítají se pouze dosažená řešení)ů algoritmy jsou porovnávány na základě ceny vyhledávání (velikost paměti, čas procesoru).

- **Test cíle:** 8 dam na šachovnici, navzájem se neatakují.
- **Cena cesty:** 0.

Dále existují různé možnosti stavů a operátorů pro inkrementální postup:

- **Stavy:** libovolné rozmísťování 0-8 dam na šachovnici.
- **Operátory:** přidat dámu na libovolné pole.

Zde je 64^8 = 281,474,976,710,656 možných sekvencí k prozkoumávání. Rozumnější volba může např. využít skutečnost, že umístění dámy na atakované pole nevede k řešení, protože umístění dalších dam tento útok neodstraní:

- **Stavy:** rozmísťování 0-8 dam, přičemž se žádné neatakují.
- **Operátory:** umístění dámy do nejbližšího levého neobsazeného sloupce tak, aby nebyla atakována žádnou z dalších dam.

Tyto akce nevytvářejí žádné stavy s atakovanými dámami; existují však stavy, kdy není žádná akce možná (na prvním obrázku s dámami nelze po umístění prvních sedmi vložit osmou). Zde je 2057 možných sekvencí k prozkoumání.

Je evidentní, že ***správná formulace vysoce ovlivňuje rozměr prohledávaného prostoru***.

Při formulaci kompletního stavu lze problém stanovit takto:

- **Stavy:** rozmístění 8 dam, po jedné v každém sloupci.
- **Operátory:** přemístění každé atakované dámy na jiné pole v tomtéž sloupci.

Tato formulace umožní nakonec najít řešení, ale ještě lepší by bylo přemísťování atakované dámy na sloupci pouze na neatakovaná pole (pokud je to možné).

Kryptoaritmetika:

V kryptoaritmetických problémech jsou číslice nahrazeny písmeny a cílem je nalézt substituci číslic za písmena tak, aby výsledný součet byl aritmeticky správný. Obvykle každé písmeno představuje jinou číslici:

```
 FORTY      Řešení:    29786   F=2, O=9, R=7 atd.
+  TEN                   850
+  TEN                   850
------                 -----
 SIXTY                 31486
```

Asi nejjednodušší řešení:

- **Stavy:** kryptoaritmetická záhada s některými písmeny nahrazenými číslicemi.
- **Operátory:** náhrada všech výskytů písmene číslicí, která se dosud nevyskytuje v záhadě.
- **Test cíle:** záhada obsahuje pouze číslice a představuje správný součet.
- **Cena cesty:** 0 (všechna řešení jsou stejně platná).

Zde je např. vidět, že náhrada `E` pomocí `6` a `F` pomocí `7` je totéž jako `F` pomocí `7` a `E` pomocí `6`—pořadí nehraje roli ve správnosti součtu, takže snaha je vyhnout se permutacím těchže substitucí.

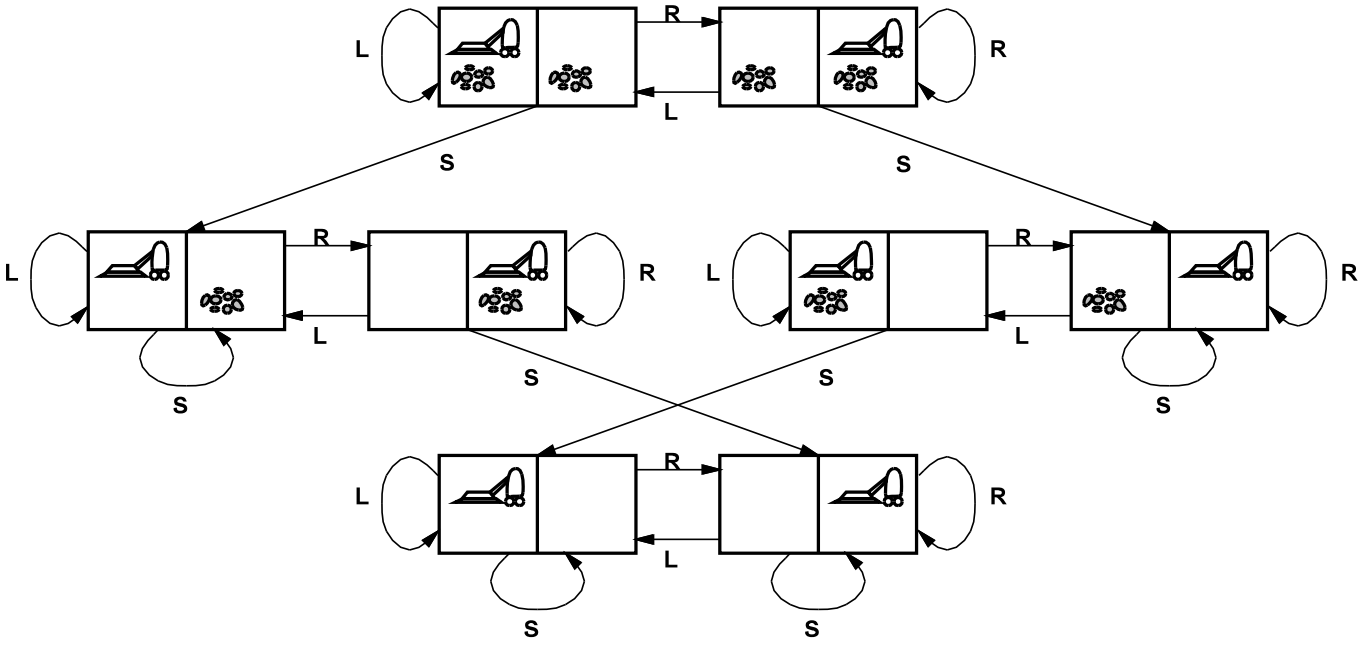
Jednou z možností je použít pevné pořadí, např. abecední. Lepší možností je použít co nejomezenější substituce, tj. písmene, které má nejméně správných možností za předpokladu dané záhady.

Svět vakua

Zde si definujeme zjednodušený vakuový svět z obrázku o vysávání nečistot:

Předpokládejme, že agent zná své umístění včetně umístění všech částic nečistoty a že vysávání probíhá ve správném pořadí:

- **Stavy:** jeden z 8 stavů zobrazených na obrázku (nebo na předchozím vysavačovém obrázku).
- **Operátory:** pohyb doleva, pohyb doprava, vysátí.
- **Test cíle:** žádná nečistota v žádném čtverci.
- **Cena cesty:** každá akce stojí 1.



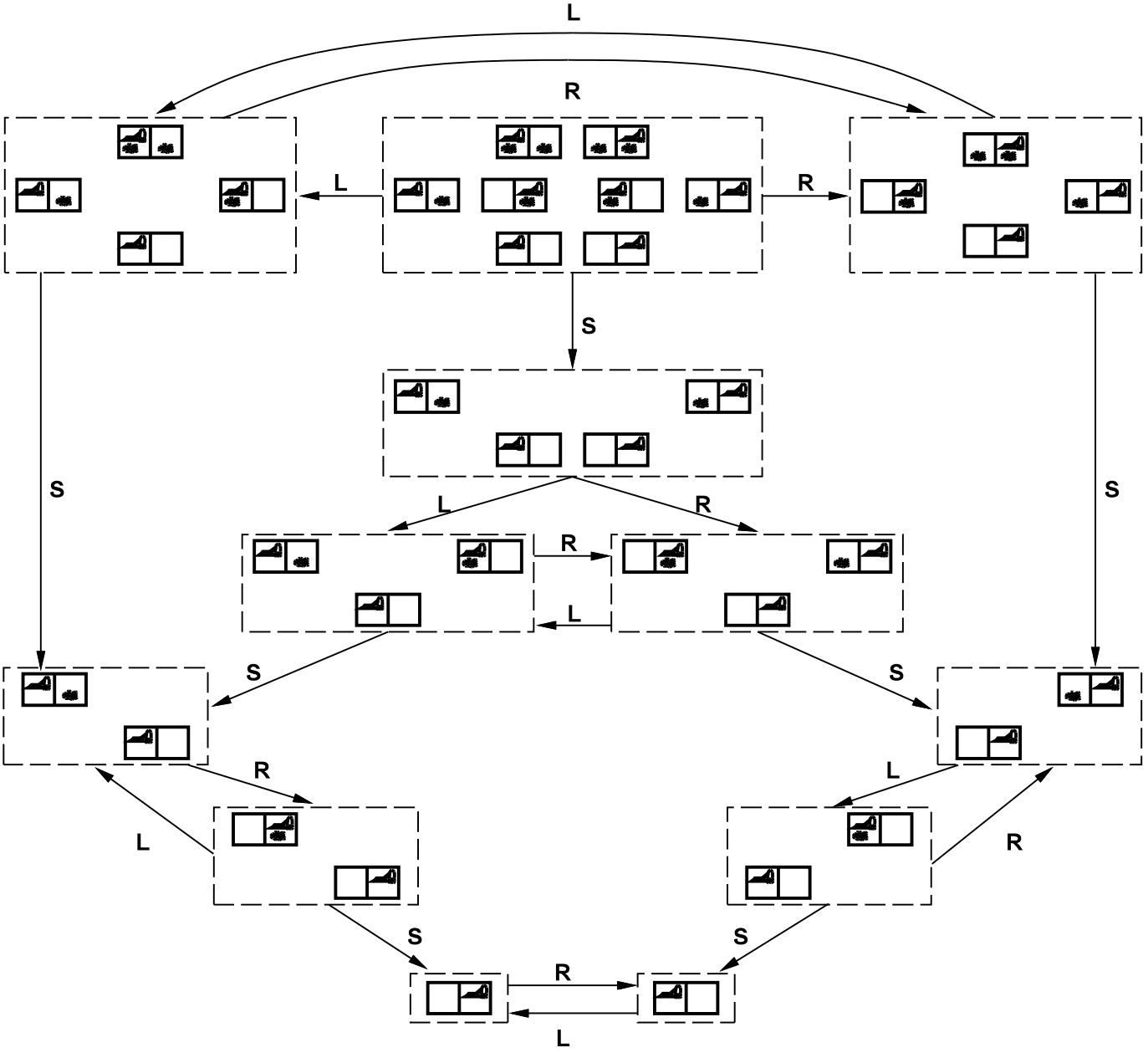
(S = vysátí, L = doleva, R = doprava, oblouky označují akce)

Na obrázku je kompletní stavový prostor se všemi možnými cestami. Řešení z libovolného počátečního stavu znamená jednoduše se pohybovat ve směru šipek k cílovému stavu. To je samozřejmě možný postup pro všechny problémy, ale většinou je stavový prostor nesrovnatelně složitější a propletenější.

Uvažme nyní případ, kdy agent nemá žádné sensory, ale jeho úkolem je vysátí veškeré špíny. Jedná se o mnohostavový problém:

- **Soubor stavů:** podmnožina stavů 1-8 (viz obrázek).
- **Operátory:** pohyb doleva, pohyb doprava, vysátí.
- **Test cíle:** všechny stavy v souboru stavů jsou bez špíny.
- **Cena cesty:** každá akce stojí 1.

Výchozí množina stavů je množina všech stavů, protože agent nemá sensory. Řešením je jakákoliv posloupnost vedoucí z výchozí množiny stavů do stavů bez špíny (v čárkovaných obdélnících jsou soubory stavů; v libovolném bodě agent neví, v kterém stavu z množiny stavů je):



Misionáři a kanibalové

Tři misionáři a tři kanibalové jsou na jedné straně řeky a mají k dispozici jednu loďku, která uveze jednoho nebo dva lidi. Úkolem je najít způsob přepravy všech šesti lidí na druhou stranu řeky tak, aby nikdy nezůstala skupina misionářů s číselnou převahou kanibalů na jednom místě (břehu).

Tento problém patří mezi proslulé problémy v oblasti umělé inteligence, protože byl jako první v literatuře formulován z analytického hlediska (v r. 1968). Zde je nutno napřed výrazně abstrahovat, zbavit úlohu nepotřebných detailů (které by se v reálné situaci přirozeně vyskytly).

Číselná převaha není v loďce možná, takže zbývají pouze oba břehy jako možnost. Čas dopravy není nutný. Pokud má do loďky nastoupit cestující, je jedno, který kanibal či misionář, na kterém břehu se začíná... apod.

- **Stavy:** stav se skládá z uspořádané sekvence tří čísel representujících počet misionářů, počet kanibalů, a počet loděk na každém břehu. Tedy počáteční stav je (3, 3, 1).
- **Operátory:** z každého stavu existují možné operátory: do loďky 1 nebo 2 misionáři, 1 nebo 2 kanibalové, 1 misionář a 1 kanibal, tj. max. 5 operátorů, i když ve většině stavů jich bude zřejmě méně kvůli vyhnutí se nepřípustným stavům. (Pozn.: kdybychom rozlišovali jednotlivce jako různé osoby, pak bychom měli 27 operátorů.)
- **Test cíle:** dosažení stavu (0, 0, 0).
- **Cena cesty:** počet přejezdů přes řeku.

Uvedený stavový prostor je dostatečně malý, takže pro počítač je nalezení řešení triviální, pro lidi je problém o dost obtížnější, zejména proto, že některé nezbytné kroky se zdají být retrográdní.

Problémy reálné

Hledání trasy

Byl demonstrován problém nalezení cesty z jednoho města do druhého. Tento typ problémů má široké aplikační možnosti, např. hledání trasy v počítačové síti nějakým počítačem, automatizovaný cestovní poradní systém, plánování cesty leteckým spojením, apod. Posledně uvedená aplikace patří ke značně komplikovaným z hlediska složitosti ceny za cestu v penězích, pohodlí, období dne, typu letadla, bonusů za časté sestovaní určitou společností atd. Akce tohoto problému navíc nemají zcela známé výsledky—dochází ke zpoždění, vystavení více letenek než je v letadle míst (overbooking), chybějící spojení, mlha...

Problémy typu “obchodní cestující”

Na mapce cestování z **A** do **B** silze představit rovněž nutnost vyřešení problému obchodního cestujícího (návštěva každého města jednou, start i cíl ve městě Bucharest). Zde je nutno uchovávat více informace než při pouhém přemístění se z jednoho místa do druhého (udržování záznamu o již navštívených místech...). Požadavkem je co nejkratší (časově a/nebo délkově) cesta.

Návrh obvodů VLSI

Tato úloha patří mezi nejsložitější návrhy—typický VLSI má miliony hradel, pozice a propojení každého hradla je rozhodující pro správnou činnost. Zde se používají nástroje typu CAD v každé fázi procesu návrhu. K nejobtížnějším částem patří rozložení jednotlivých buněk a propojení; tyto realizační problémy se řeší po stanovení návrhu komponent a jejich propojení, účelem je minimalizovat plochu a délku spojení pro dosažení co nejvyšší rychlosti. Při návrhu buněk jsou primitivní komponenty obvodu sdružovány do buněk, z nichž každá provádí nějakou stanovenou funkci. Každá buňka má pevný tvar a velikost a vyžaduje určitý počet propojení s ostatními buňkami. Buňky se nesmí na čipu překrývat, mezi nimi musí být dost místa pro propojovací cesty.

Navigace robotů

Navigace robotů je vlastně zobecněním problému nalezení trasy. Spíše než v diskrétní množině tras se robot pohybuje ve spojitém prostoru, kde je principiálně nekonečný počet možných stavů a akcí. Robot pohybující se do kruhu na ploše je pouze dvoudimensionální; jakmile má robot ruce a nohy, které je nutno řídit, pak problém přechází do multidimensionálního stavového prostoru. Pokročilé techniky jsou zapotřebí už jen pro limitaci stavů na konečný počet. Kromě toho se po reálných robotech vyžaduje, aby se uměli vypořádat s chybami způsobenými sensory a řízením servomotorů.

Montážní posloupnosti

Automatická montáž složitých objektů robotem byla poprvé demonstrována v r. 1972 robotem Freddy. Od té doby dochází k pomalému, avšak jistému pokroku, kdy takto lze např. ekonomicky smontovávat elektromotory. Jedná se o nalezení posloupnosti montáže součástek (při chybě nelze např. zamontovat další součástky, apod.). Jedná se o vysoce složitý geometrický vyhledávací problém spojený s navigací robotických zařízení.

Vyhledávání řešení

Dosud bylo ukázáno, jak definovat problém a jak stanovit typ jeho řešení. Zbývající částí je nalezení vhodného řešení, což se provádí určitým prohledáváním stavového prostoru.

Generování sekvencí akcí

Generování nových stavů—z počátečního stavu je nutno vybudovat rozšíření tak, aby vznikly nové stavy k prozkoumání, zda se tak dosáhne cíle nebo přiblížení k cíli (ke změně pozice z města **A** do města **B** nestačí mít jen stav “jsem v **A**”, kde je k dispozici jen akce “z **A** do **A**”, což je neužitečné. Je zapotřebí vytvořit nové možné stavy (další místa propojená s **A**, tedy něco jako jednokrokové spojení s následujícími možnými pozicemi). Kdykoliv je k dispozici více možností, je nutno mít k dispozici možnost výběru, co dál.

Expanse stavu

Generování nových stavů se také nazývá *expansí stavu*—z hlediska nalezení cesty z města Arad je stav "v Aradu" expandován na nové možné stavy "v Sibiu", "v Timisoara" a "v Zerind", protože existuje jeden krok (trasa) z Aradu do uvedených tří měst. Pokud by byl jen jeden další stav, tak je prostě zvolen a agent pokračuje. Při více možnostech dochází k volbě.

Výše uvedený popis tvoří základ vyhledávání. Řekněme, že agent se přesunul do města Zerind, tj. dosáhl nového stavu:

- **Test cíle:** cíl není dosažen (tím je Bucharest).

Okamžitý stav je expandován na možnosti jet do města Arad a do Oradea. Volba zde může vybrat kterýkoliv z těchto dvou dalších stavů, dále jet zpět a pak vybrat Sibiu nebo Timisoara. Pokračuje tedy výběr, testování a expanse dokud není dosaženo cíle nebo dokud již nejsou k dispozici další stavy k expansi.

Výběr stavu, který má být expandován jako první, je **strategie výběru**.

O procesu vyhledávání je vhodné uvažovat jako o **vytváření vyhledávacího stromu**, jímž dochází k superpozici přes stavový prostor.

Kořen stromu je **vyhledávací uzel** odpovídající počátečnímu stavu.

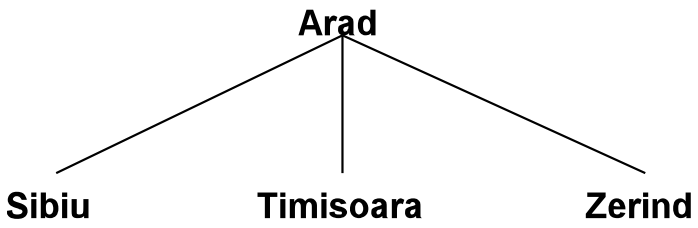
Listy stromu odpovídají stavům, které nemají ve stromu následné stavy—buď nebyly expandovány nebo po expansi generují prázdnou množinu.

Následující obrázek ukazuje některé expanse ve vyhledávacím stromu pro nalezení trasy z města Arad do města Bucharest.

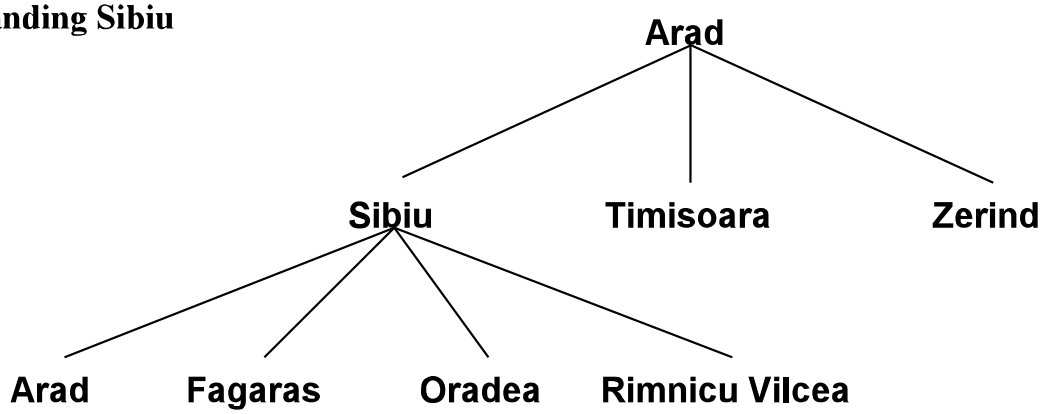
Dále následuje obecný vyhledávací algoritmus.

(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu




```
function General-Search(problem, strategy) returns řešení nebo
     neúspěch

     inicialisace vyhledávacího stromu pomocí počátečního stavu,
     v němž se nachází problem

     loop do
       if nejsou žádní kandidáti na expansi then return neúspěch
       vyber listový uzel pro expansi v souladu se strategy
       if uzel obsahuje cílový stav then return řešení
       else expanduj uzel a přidej výsledné uzly ke stromu
     end
```

Je důležité rozlišovat mezi stavovým prostorem a vyhledávacím stromem:

Pro problem nalezení trasy existuje pouze 20 stavů ve stavovém prostoru, jeden pro každé město (20 měst).

Obecně může stavový prostor obsahovat nekonečný počet cest, tj. strom má nekonečný počet uzlů. Např. větev Arad–Sibiu–Arad pokračuje Arad–Sibiu–Arad–Sibiu–Arad, atd. do nekonečna.

Je zjevné, že dobrý vyhledávací algoritmus se vyhne takovým cestám.

Datové struktury pro vyhledávací strom

Pro representaci uzlů je mnoho způsobů, zde bude použita representace uzlu jako datová struktura s pěti složkami:

- stav, jemuž uzel ve stavovém prostoru odpovídá;
- uzel ve vyhledávacím stromu, který vygeneroval daný uzel (**rodičovský uzel**);
- operátor aplikovaný na generování uzlu;
- počet uzlů na cestě z kořene do daného uzlu (**hloubka** uzlu);
- cena cesty (přesunu z počátečního stavu do uzlu).

```
datatype node
     components: State, Parent-Node, Operator, Depth, Path-Cost
```

Mezi *stavy* a *uzly* existuje rozdíl:

Uzel je záznamová datová struktura použitá pro representaci vyhledávacího stromu pro určitou instanci konkrétního problému generovaná konkrétním algoritmem.

Stav representuje konfiguraci (nebo soubor konfigurací) světa—uzly mají hloubky a rodiče, stavy nikoliv. Dva různé uzly mohou obsahovat tentýž stav, pokud je tento stav generován dvěmi různými sekvencemi akcí.

Pro representaci souboru uzlů, které čekají na na expansi, se používá pojem **periferie** (fringe). Nejjednodušší representací je množina uzlů. Vyhledávací strategie pak je funkce, která vybírá další uzel k expansi z této množiny. To ovšem může být výpočetně náročné, např. pokud by se vybíral k prozkoumání každý periferní uzel, aby byl nakonec vybrán ten nejlepší.

Zde budeme předpokládat implementaci těchto uzlů jako **frontu**. Operace nad frontou:

- `Make-Queue(`*`Elements`*`)` vytváří frontu z daných prvků.
- `Empty?(`*`Queue`*`)` vrací TRUE pokud již ve frontě nejsou další prvky (prázdná fronta).
- `Remove-Front(`*`Queue`*`)` odstraní prvek na počátku fronty a předá jej volající funkci či proceduře.
- `Queuing-Fn(`*`Elements, Queue`*`)` vloží soubor prvků do fronty. Různé typy funkcí pro zařazování do fronty produkují různé typy vyhkledávacích algoritmů.

Formálnější verze obecného vyhledávacího algoritmu:

```
function General-Search(problem, Queuing-Fn) returns řešení nebo
     neúspěch

     nodes ← Make-Queue(Make-Node(Initial-State[problem]))

     loop do
      if nodes is empty then return neúspěch
      node ← Remove-Front(nodes)
      if Goal-Test[problem] pro State(node) je úspěšný
      then return node
      nodes ← Queuing-Fn(nodes,Expand(node,Operators[problem]))
     end
```

Pozn.: `Queuing-Fn` je proměnná, jejíž hodnotou je funkce.

Strategie vyhledávání

Klíčovým problémem v umělé inteligenci vždy bylo nalezení správné vyhledávací strategie pro daný problem. Zde se budeme zabývat vyhodnocením strategie ze čtyř hledisek:

- **Úplnost:** zaručuje strategie nalézt řešení, pokud řešení existuje?
- **Časová složitost:** kolik času zabere najít řešení?
- **Prostorová složitost:** kolik paměti zabere vyhledávání?
- **Optimalita:** najde strategie nejlepší řešení, pokud existuje více různých řešení?

Vyhledávání existuje (z určitého pohledu) ve dvou formách: **neinformované** a **informované**. Neinformované vyhledávání má šest základních strategií.

Neinformované hledání se vyznačuje tím, že proces nemá žádné informace o počtu kroků či ceně cesty z aktuálního stavu do stavu cílového—vše, co je toto hledání schopno rozlišit, je zda se jedná o cílový nebo necílový stav.

Neinformované vyhledávání se někdy nazývá jako **slepé hledání**.

V příkladu o nalezení cesty z města Arad (počáteční stav) jsou tři akce vedoucí do tří nových stavů: Sibiu, Timisoara a Zerind (viz mapku). Neinformované hledání nemá pro žádnou z těchto akcí preferenci, avšak chytřejší agent by si mohl povšimnout, že cíl (Bucharest) je jihovýchodně od Aradu, a že pouze Sibiu leží tímto směrem, takže je pravděpodobné, že Sibiu je nejlepší volba.

Strategie, které využívají podobné uvažování, se nazývají **informované hledání** nebo také **heuristické vyhledávací strategie** (bude probráno později).

Neinformované hledání je méně efektivní, avšak stále velmi důležité, protože existuje mnoho problémů, kde informované vyhledávání nelze použít—k dispozici prostě není žádná přídavná informace.

Šest základních vyhledávacích neinformovaných strategií se od sebe liší *pořadím* prozkoumávaných uzlů, na čemž může hodně záležet.

Hledání prvně do šířky

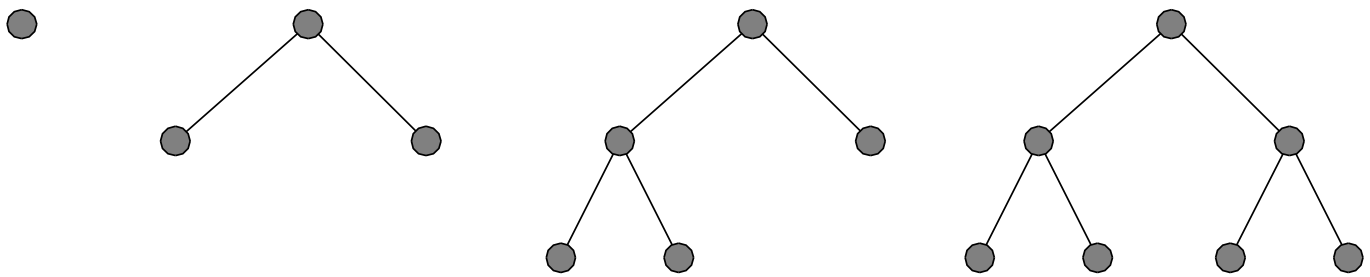
Je to jednoduchá strategie—prvně se expanduje kořen vyhledávacího stromu, dále všechny uzly generované kořenem, dále všichni potomci generovaní těmito uzly, atd. Obecně jsou veškeré uzly v hloubce *d* expandovány před uzly v hloubce *d+1*.

Hledání prvně do šířky lze implementovat voláním algoritmu `General-Search` s funkcí `Queuing-Fn` (umísťování do fronty), která umísťuje nově generované stavy na konec fronty, tj. za všechny dříve generované stavy, `Enqueue-At-End`:

```
function Breadth-First-Search(problem) returns řešení nebo
     neúspěch

     return General-Search(problem, Enqueue-At-End)
```

Tato strategie je velmi systematická, protože uvažuje napřed všechny cesty délky 1, pak všechny délky 2, atd. Postup vyhledávání v jednoduchém binárním stromě ukazuje obrázek:



Na obrázku je znázorněna expanse uzlů v pořadí 0, 1, 2 a 3.

Existuje-li řešení, hledání prvně do šířky zaručuje jeho nalezení.
Existuje-li více řešení, vždy se najde to první, "nejmělčejší" z hlediska hloubky.

Z hlediska čtyř uvedených kritérií: hledání je kompletní; optimální je tehdy, pokud *cena cesty je neklesající funkcí hloubky uzlu*. (Tato podmínka je obvykle splněna jen tehdy, když všechny operátory mají stejnou cenu.)

Tento typ vyhledávání není vhodný tehdy, když nastanou potíže s časem a prostorem hledání. Uvažme hypotetický stavový prostor, kde každý stav může být expandován do b nových stavů (**faktor větvení** je b)—kořen geberuje b uzlů, každý tento uzel generuje rovněž b uzlů, takže na druhé úrovni je b^2 uzlů (kořen je na úrovni č. 0).

Má-li řešení délku cesty d, pak max. počet expandovaných uzlů bude

$$1 + b + b^2 + b^3 + \ldots + b^d,$$

což je maximální počet; řešení ovšem může být nalezeno v libovolném bodě na úrovni d, tj. počet může být menší. Složitost je tedy exponenciální: $O(b^d)$. Pro zajímavost, nechť při hledání prvně do šířky je b=10, rychlost zpracování 1000 uzlů/s a každý uzel vyžaduje 100 B. To zhruba odpovídá mnoha problémům typu řešení hlavolamu (faktor může být 100 krát větší či menší) na moderních počítačích. Hodnoty růstu ukazuje tabulka:

hloubka	**počet uzlů**	**čas**	**paměť**
0	1	1 ms	100 B
2	111	0.1 s	11 kB
4	11 111	11 s	1 MB
6	10^6	18 min.	111 MB
8	10^8	31 hod.	11 GB
10	10^{10}	128 dnů	1 TB
12	10^{12}	35 let	111 TB
14	10^{14}	3500 let	11 111 TB

Z uvedených výsledků lze vyvodit dvě věci:

1. Paměťové požadavky jsou větším problémem u hledání prvně do šířky, než požadavky časové. Řada lidí asi počká 18 minut na prohledání v hloubce 6, avšak 111 MB paměti může znamenat příliš mnoho. A počkat 31 hodin také lze, ovšem 11 GB paměti pro prohledání hloubky 8 je těžko řešitelný problém. Existují ovšem jiné strategie, které vyžadují méně paměti.

2. Časové požadavky jsou stále velmi důležitým faktorem. Pokud by problém vyžadoval hloubku 12, pak v uvedeném příkladu to znamená 35 let. I když by po 10 letech byl k dispozici počítač 100 krát rychlejší (dle současných trendů vývoje) za stejnou cenu, pak by bylo třeba 128 dní, a pro hledání na úrovni 14 opět 35 let. Obecně platí, že problémy s exponenciální složitostí vyhledávání nemohou být řešeny hledáním prvně do šířky pro jiné než malé úlohy.

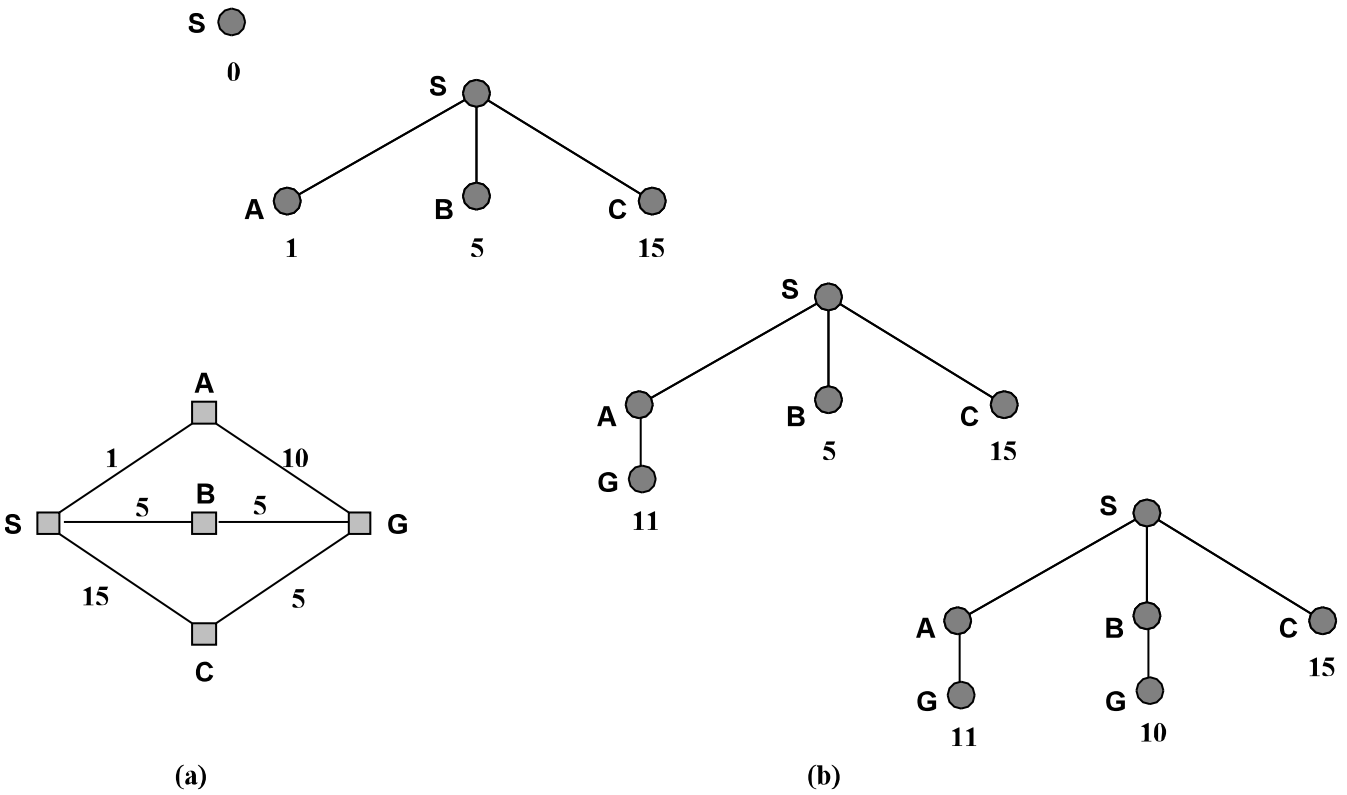
Uniformní cenové hledání

Nalezení cíle v nejmenší hloubce nemusí vždy být nejlevnějším řešením z hlediska obecné funkce ceny za cestu. Hledání uniformní-cenové modifikuje strategii hledání prvně do šířky tak, že vždy prvně expanduje nejlevnější uzel na dosaženém okraji vyhledávání (vzhledem k měření ceny cesty $g(n)$) místo uzlu nejméně hlubokého.

Hledání prvně do šířky je uniformní hledání, pokud platí $g(n) = \text{Depth}(n)$, tj. cena cesty odpovídá pouze hloubce.

Při splnění určitých podmínek platí, že prvně nalezené řešení je zaručeně to nejlevnější, protože pokud by k řešení vedla levnější cesta, byla by expanzemi nalezena dříve, takže by musela být nalezena dříve.

Následující obrázek ilustruje popisovanou strategii (problém nalezení trasy).



Část a) ukazuje stavový prostor. Problémem je dostat se z počátku *S* do cíle *G*, přičemž je vyznačena cena pro každý operátor. Strategie napřed expanduje počáteční stav a získá tím cesty do *A, B* a *C*. Protože do *A* je to nejlevnější, tak *A* je expandováno jako další uzel, a cesta *S*-*A*-*G* je nalezena, i když není optimální.

Algoritmus nepovažuje výsledek za konečný, protože cena je 11, takže tato cesta je do fronty umístěna *zaS*-*B*, kde je zatím cena pouze 5 a možná, že další kroky do *G* mohou vyjít levněji (důvodem je hledání optima). Důležité je, že zatím z *B* (ani *C*) není vidět cíl *G*, takže pokud se tam lze dostat i odtud, je také možné, že levněji.

B je levnější než *C*, takže je expandováno *B* a vytvořena cesta *S*-*B*-*G*, která je nyní 10—levnější než *S*-*A*-*G*, zároveň je tato cesta rozeznána jako vedoucí do cíle a optimální z nalezených řešení. Protože *S*-*C* má cenu 15, tak i kdyby *C*-*G* stálo 0, bude to dražší než *S*-*B*-*G*, takže hledání končí. Uniformní cenové vyhledávání najde nejlevnější řešení za předpokladu, že platí jednoduchý požadavek: cena cesty nesmí nikdy klesat při pohybu po ní:

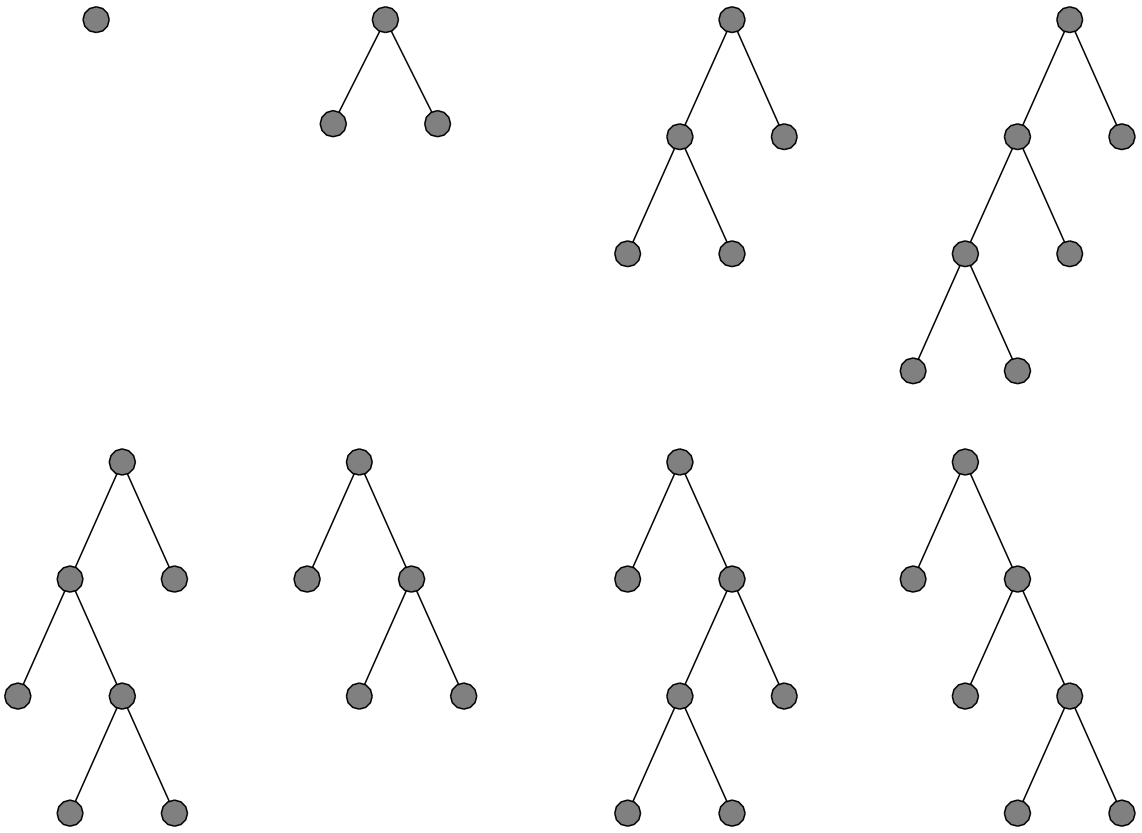
$g($`Successor`$(n)) \geq g(n)$ pro každý uzel n.

Uvedené omezení dává smysl, pokud je cena cesty pro uzel počítána jako součet cen operátorů, které tu cestu generují. Má-li každý operátor nezápornou cenu, pak cena cesty nemůže klesat a metoda najde nejlevnější cestu bez nutnosti projít celým stromem.

Pokud by mohla existovat záporná cena, pak pro nalezení nejlevnější cesty by se musel projít celý vyhledávací strom (např. cena *C*-*G* by mohla být -6, tedy cesta *S*-*C*-*G* by pak byla nejlevnější ($g = 15 - 6 = 9 < 10 < 11$) a nešla by najít jinak, než vyčerpávajícím prohledáním stromu).

Prohledávání prvně do hloubky

Tento typ vyhledávání vždy expanduje jeden z uzlů na nejnižší úrovni stromu. Pouze když hledání dojde do uzlu, který není cílový a nemá expansi, vrací se hledání ve stromu zpátky a expanduje uzly na nižších úrovních. Uvedená strategie může být implementována funkcí `General-Search` s funkcí pro umísťování do fronty, která vždy dává nově generované stavy na počátek fronty. Protože expandovaný uzel je nejhlubší, jeho následníci budou ještě hlubší a tudíž nyní nejhlubší. Zobrazení postupu (uzly na úrovni 3 zde nemají žádné následníky):



Vyhledávání prvně do hloubky se vyznačuje malými paměťovými nároky. Na obrázku je vidět, že se musí ukládat pouze jedna cesta z kořene do listu plus zbývající sourozenecké uzly pro každý uzel na cestě.

Pro stavový prostor s faktorem větvení b a maximální hloubkou m ukládá tato metoda pouze bm uzlů oproti b^d uzlům při hledání prvně do šířky pro případ, že nejméně hluboký cíl je na úrovni d.

Při srovnání s tabulkou požadavků na čas a paměť, uvedenou výše, by hledání prvně do hloubky potřebovalo (za stejných podmínek) 12 kB místo 111 TB (tera = 10^{12}) v hloubce d= 12, tj. velmi významný faktor zmenšení.

Časová složitost je $O(b^m)$. Prohledávání prvně do hloubky může být ve skutečnosti rychlejší než do šířky pro problémy, které mají *velmi mnoho* řešení, protože obvykle je dobrá možnost najít řešení po prohledání pouze malé části prostoru.

Hledání do šířky by muselo i v tomto případě hledat ve všech cestách délky d-1 před přechodem na libovolnou cestu délky d.

Hledání do hloubky by bylo $O(b^m)$ v nejhorším případě.

Nevýhodou hledání do hloubky je však skutečnost, že může uvíznout při pohybu dolů po špatné cestě. Mnoho problémů má velmi hluboké (případně nekonečně hluboké) vyhledávací stromy, takže hledání prvně do hloubky by se nemuselo vůbec vzpamatovat z chybně zvoleného uzlu blízko kořene stromu. Pak může dojít k uvíznutí v nekonečném cyklu, v lepším případě (z těch špatných) najde cestu k řešení, která vůbec není optimální. Z toho plyne poznatek, že hledání prvně do hloubky *není ani optimální, ani úplné*. Proto je vhodné se tomuto hledání vyhnout pro hluboké stromy. Implementace (často bývá rekursivní, kdy se sama volá na úrovni každého svého potomka):

```
function Depth-First-Search(problem) returns řešení nebo
     neúspěch

     return General-Search(problem, Enqueue-At-Front)
```

Hledání s omezenou hloubkou

Tento přístup se vyhýbá možné pasti uvíznutí při hledání do hloubky tím, že použije přerušení postupu do další hloubky při dosažení předem stanoveného limitu. Operátory musí uchovávat informaci o dozažené hloubce.

Např. na uvedené mapě Rumunska je 20 měst, takže je známo, že pokud existuje řešení, tak cesta musí být nanejvýš 19 jednotek dlouhá: Pokud jsem ve městě **A** a cesta zabrala méně než 19 kroků, generuje se stav ve městě **B** s délkou cesty zvětšenou o 1.

Zde je zaručeno nalézt řešení, pokud existuje, ale není zaručeno optimální řešení. Hledání s omezenou hloubkou je kompletní, ale není optimální. Pokud zvolíme hloubku příliš malou, pak toto hledání není ani kompletní.

Čas a prostor mají podobnou složitost jako hledání prvně do hloubky:

čas: $O(b^l)$ prostor: $O(bl)$

kde l je limit hloubky.

Hledání iterativně do hloubky

Pokouší se o vyřešení obtížného problému stanovit vhodný limit. U mapy byl vybrán limit 19 jako zřejmý. Na základě podrobnějšího studia mapy lze odhalit, že libovolné město může být dosaženo z libovolného jiného města po cestě dlouhé maximálně 9 kroků.

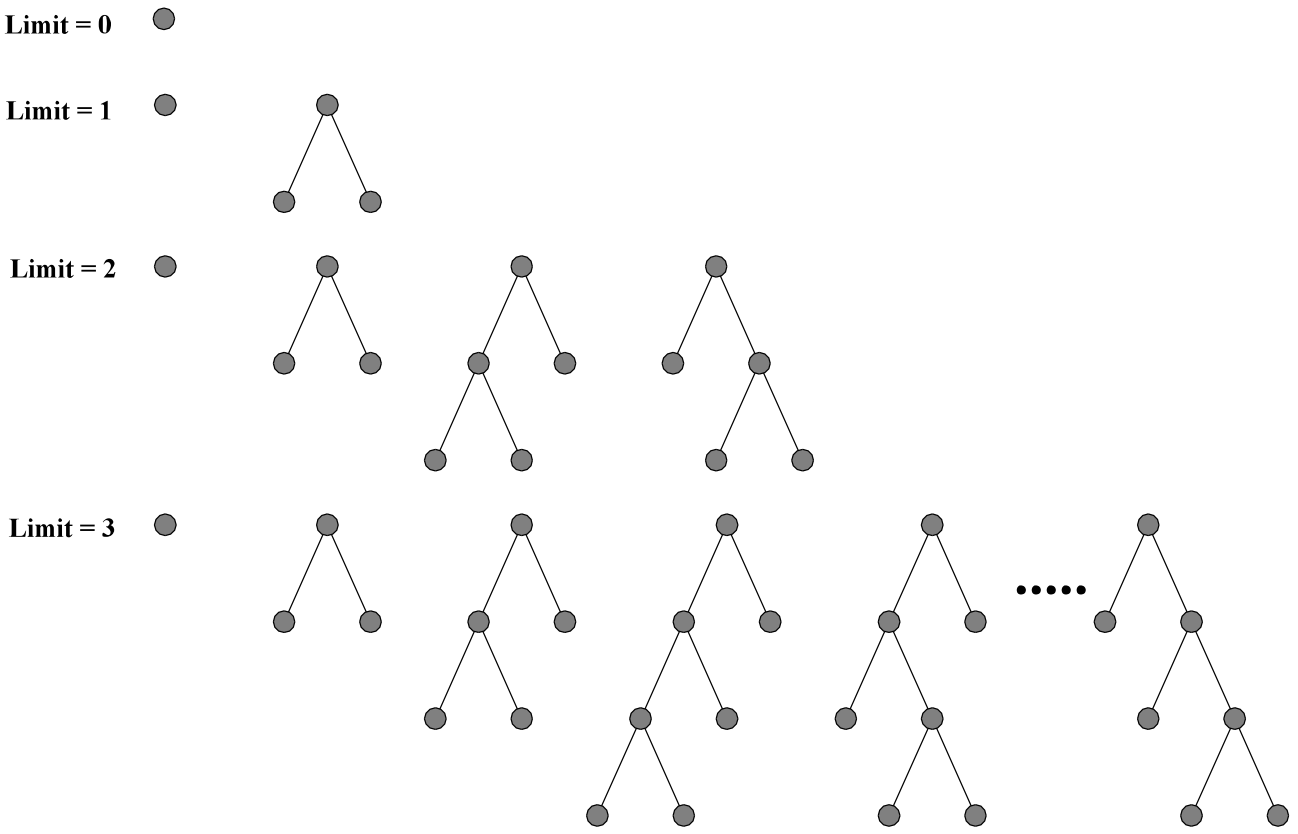
Toto číslo (zde 9) je tzv. **průměr (diametr)** stavového prostoru a poskytuje lepší limit hloubky a efektivnější prohledávání s omezenou hloubkou. Ale pro většinu problémů, jak se v praxi ukazuje, neznáme dobrý limit do té doby, než nalezneme řešení.

Hledání iterativně do hloubky je strategie, která se vyhýbá problému výběru nejlepšího limitu hloubky tak, že zkouší všechny možné hloubkové limity: prvně hloubku 0, pak 1, 2, atd. Algoritmus je zapsán následovně:

```
function Iterative-Deepening-Search(problem) returns sekvenci
     řešení
     inputs: problem, problém

     for depth ← 0 to ∞ do
          if Depth-Limited-Search(problem, depth) je úspěšný
          then return výsledek hledání
     end
     return neúspěch
```

Algoritmus kombinuje výhody hledání prvně do šířky a prvně do hloubky. Je optimální a kompletní, ale má pouze skromné paměťové požadavky. Pořadí expanse stavů je obdobné hledání prvně do šířky, s tou výjimkou, že některé stavy jsou expandovány vícekrát. Prohledávání ilustruje obrázek, kde jsou zobrazeny první čtyři iterace provedené na binárním vyhledávacím stromu pomocí funkce `Iterative-Deepening-Search`:



Zdánlivě jde o marnotratný algoritmus, expandující mnoho stavů vícekrát. Avšak pro mnoho problémů je režie spojená s expanzí ve skutečnosti malá.

Intuitivně lze problém chápat tak, že ve stromu s exponenciální vyhledávací složitostí jsou téměř všechny uzly ve spodní části, takže nezáleží téměř vůbec na tom, že vrchnější uzly jsou vícekrát expandovány. Počet expansí při hledání omezeném hloubkou d s faktorem větvení b je

$$1 + b + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

což pro konkrétní hodnoty $b = 100$ a $d = 5$ dává

$$1 + 10 + 100 + 1000 + 10\,000 + 100\,000 = 111\,111.$$

V iterativním hledání do hloubky jsou uzly na spodní úrovni expandovány jednou, uzly v úrovni nad nimi dvakrát, atd., směrem ke kořeni stromu, který je expandován d+1 krát. Celkový počet expansí iterativního hledání do hloubky je tedy

$$(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + \ldots + b^d$$

což pro stejné konkrétní hodnoty $b = 100$ a $d = 5$ dává

$$6 + 50 + 400 + 3000 + 20\,000 + 100\,000 = 123\,456.$$

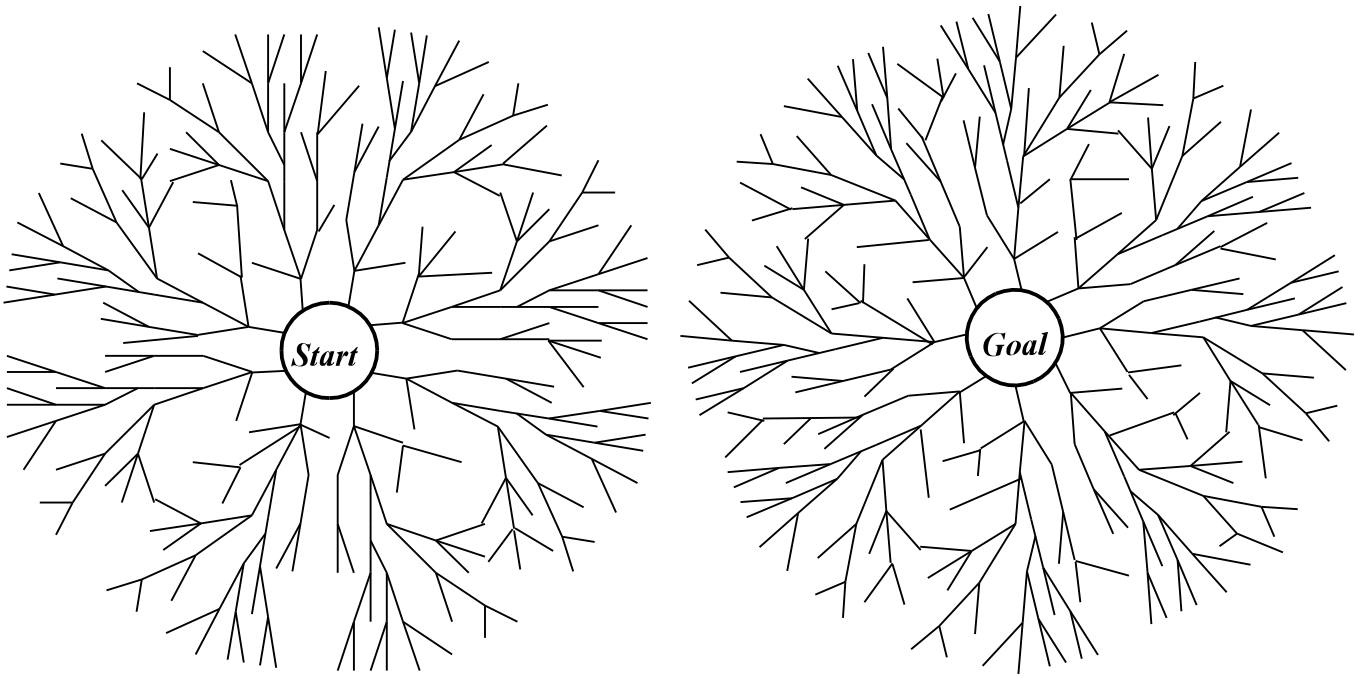
Rozdíl je pouze 11% uzlů navíc. Čím vyšší je faktor větvení, tím nižší je režie opakovaně expandovaných stavů. I když je faktor větvení = 2, iterativní hloubkové hledání zabere pouze dvojnásobek délky kompletního hledání prvně do šířky.

Časová složitost iterativního hledání do hloubky je tedy $O(b^d)$, a paměťová složitost je $O(bd)$.

Obecně platí, že iterativní hledání do hloubky je preferovaná metoda pro rozsáhlé vyhledávací prostory, přičemž hloubka řešení není známá.

Dvousměrné vyhledávání

Základní myšlenkou je současně vyjít ze dvou směrů—pohyb vpřed z počátečního stavu a pohyb zpět z koncového stavu (cíle), přičemž vyhledávání se zastaví, když se někde uprostřed cesty setkají:



U problémů, kde je faktor větvení b z obou směrů, má dvousměrné vyhledávání veliký vliv. Za předpokladu, že řešení leží v hloubce d, tak je lze najít v počtu kroků úměrném $O(2b^{d/2}) = O(b^{d/2})$, protože hledání vpřed i zpět potřebuje pouze poloviční cestu.

Pro hodnoty např. $b = 10$ a $d = 6$ generuje hledání prvně do šířky 1 111 111 uzlů, zatímco dvousměrné hledání pouze 2 222.

Algoritmus má však některé potíže, které je nutno při implementaci vzít do úvahy:

- Otázkou je, co znamená *hledání zpět směrem od cíle*? Jako **předchůdci** uzlu *n* jsou definovány uzly mající *n* jako následníka. Hledání zpět tedy znamená generování předchůdců postupně z cílového uzlu.
- Pokud jsou všechny operátory oboustranné, předchůdce i následník jsou identičtí; pro některé problémy je výpočet předchůdců velmi obtížný.
- Jak postupovat, existuje-li mnoho možných cílových stavů? Pokud je k dispozici explicitní seznam cílových stavů (viz např. výše obrázek jednoduchého vysavačového světa s 8 stavy), pak lze aplikovat funkci pro generování předchůdců souboru stavů tak, jak byla aplikována funkce pro generování následníků při vícestavovém hledání. Pokud máme k dispozici pouze *popis* souboru, pak se může podařit odvodit popis "souborů stavů, které by generovaly cílový soubor", avšak to může být velmi komplikovaná záležitost. Například je zde otázka: které stavy to jsou, jež předcházejí obecný cíl hry *mat* ve hře šachy? Kolik je těch stavů?
- Musí být k dispozici účinný způsob pro testování každého nového uzlu, aby bylo zcela zřejmé, zda se již vyskytuje nebo nevyskytuje ve vyhledávacím stromu v druhé části hledání (test uzlu generovaného ve zpětné části, zda se vyskytuje v dopředné části).
- Musíme rozhodnout, jaký typ hledání použijeme v každé části hledání. Předchozí obrázek ukazuje dvě vyhledávání typu prvně do šířky—je to nejlepší výběr?

Složitost $O(b^{d/2})$ předpokládá, že proces testování, zda a kde se protnou hranice vyhledávání vpřed a zpět, má konstatní časové nároky (tj. nezávisí na počtu stavů). Požadavek lze často splnit použitím hašovací tabulky.

Aby se dvě vyhledávání skutečně setkala, musí se v paměti udržovat uzly přinejmenším jednoho z nich (jako u hledání prvně do šířky). Z toho plyne paměťová složitost uniformního dvoucestného vyhledávání $O(b^{d/2})$.

Porovnání hledacích strategií

Následující tabulka přináší srovnání uvedených šesti vyhledávacích strategií z hlediska čtyř výše definovaných kritérií. Význam symbolů:

b . . faktor větvení
d . . hloubka řešení
m . . maximální hlobka vyhledávacího stromu
l . . limit hloubky

Kritérium	Prvně do šířky	Uniformní cena	Prvně do hloubky	Omezená hloubka	Iterativní hloubka	Obou-směrné
čas	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
prostor	b^d	b^d	bm	bl	bd	$b^{d/2}$
optimální	ano	ano	ne	ne	ano	ano
kompletní	ano	ano	ne	ano pro $l \geq d$	ano	ano

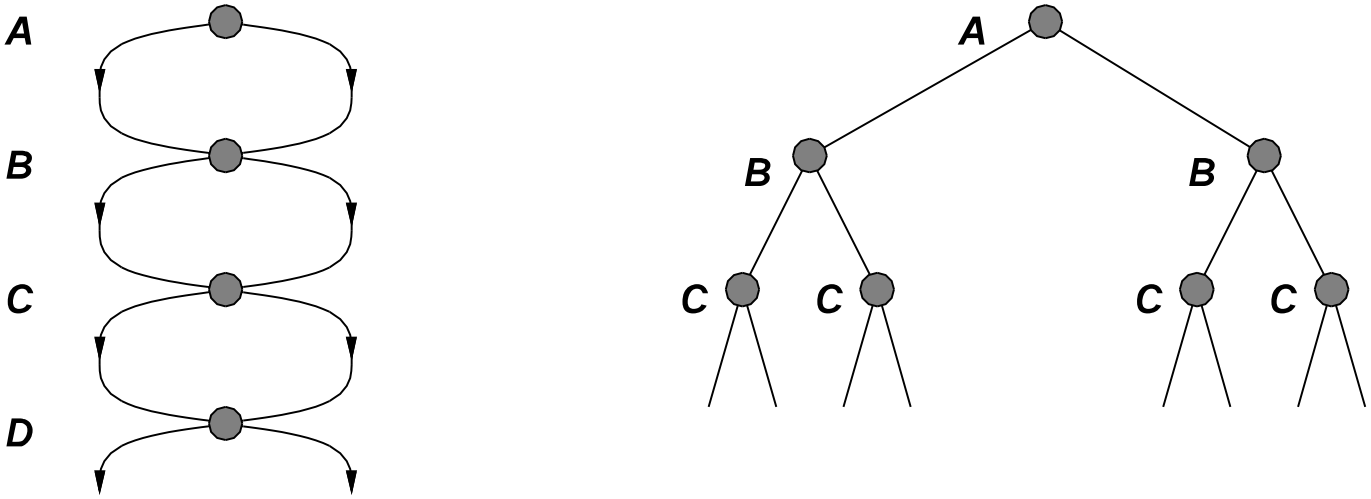
Odstranění opakovaných stavů

Dosud byla ignorována jedna z nejdůležitějších komplikací vyhledávacího procesu: možnost plýtvání časem při expansi stavů, které již se vyskytly a byly expandovány v jiných větvích.

U některých problémů k tomu vůbec nemusí dojít—ke každému stavu se lze dostat pouze jedinou cestou, např. problém 8 dam lze účinně formulovat tak, aby ke každé možné korektní pozici vedla jediná cesta.

Existují problémy, kde se opakování stavů nelze vyhnout, např. problémy s oboustrannými operátory (hledání trasy) nebo problém misionářů a kanibalů. Vyhledávací stromy jsou u těchto problémů nekonečné, ale při prořezání některých opakovaných stavů lze strom redukovat na konečný rozměr—tedy generovat pouze částečný strom pokrývající graf stavového prostoru. I pro konečně veliké stromy lze odstraněním opakovaných stavů získat exponenciální redukci ceny hledání.

Klasickým příkladem je následující obrázek, kde prostor obsahuje pouze m+1 stavů, kde m je maximální hloubka. Protože strom obsahuje každou možnou cestu stavovým prostorem, má 2^m větví:



Vlevo je stavový prostor se dvěma možnými akcemi vedoucími z **A** do **B**, dvěma z **B** do **C**, atd. Vpravo je odpovídající vyhledávací strom.

Existují tři možnosti, jak se zbavit opakovaných stavů, seřazené ve vzrůstající efektivnosti a výpočtové režii:

- Nikdy se nevracet do stavu, z něhož se právě vyšlo. Znamená to vytvořit expansní funkci (nebo soubor operátorů), která odmítne generovat následníka jenž představuje *stejný stav jako jeho uzel-rodič*.
- Nikdy nevytvářet cesty s cykly v nich. Znamená to vytvořit expansní funkci (nebo soubor operátorů), která odmítne generovat následníka jenž představuje *stejný stav jako libovolný z jeho předků*.
- Nikdy nevytvářet žádný stav, který již byl někdy vytvořen. Znamená to, že každý stav, který byl vygenerován, musí být udržován v paměti—to má za následek potenciální prostorovou složitost $O(b^d)$. Zde je výhodnější o složitosti uvažovat jako o $O(s)$, kde s je počet stavů v celém stavovém prostoru. Tato možnost využívá pro implementaci často hašovací tabulky, kde jsou uloženy všechny generované uzly, tedy kontrola opakování uzlu je pak rozumně efektivní.

Hledání vyhovující omezením

Problém, který má určitá omezení (CSP—a constraint satisfaction problem) je speciální druh problému, který vyhovuje některým přídavným strukturálním vlastnostem, které jsou obecně mimo základní požadavky na problém.

U CSP jsou stavy definovány hodnotami souboru **proměnných** a test na cíl specifikuje soubor **omezení**, která musí tyto hodnoty splňovat. Např. problém 8 dam je CSP problém v němž jako *proměnné* jsou umístění každé z osmi dam; možné *hodnoty* jsou políčka šachovnice; *omezení* požaduje, aby žádné dvě dámy nebyly v téže řadě, sloupci nebo diagonále.

Řešení CSP specifikuje hodnoty pro všechny proměnné tak, aby byla splněna omezení. Kryptoaritmetika a VLSI také mohou být popsány jako CSP, podobně jako mnoho druhů problémů návrhu a plánování, takže jde o velmi důležitou podtřídu.

CSP lze samozřejmě řešit *obecnými* vyhledávacími algoritmy, ale speciální struktura umožňuje použití *zvláštních* algoritmů navržených konkrétně pro CSP, které pracují mnohem lépe.

Omezení mají několik druhů:

- *unární* omezení se zabývají hodnotou jediné proměnné (např. proměnné, týkající se nejlevější číslice v libovolném kryptoaritmetickém problému mají omezení, že nejsou = 0);
- *binární* omezení se vztahuje ke dvojicím proměnných (např. problém 8 dam obsahuje samá binární omezení—vzhledem ke každé dvojici dam);
- omezení *vyšších řádů* zahrnují tři a více proměnných (např. sloupce v kryptoaritmetických problémech musí splňovat přídavná omezení a mohou zahrnovat několik proměnných);
- *absolutní* omezení, která vylučují některá potenciální řešení;
- *preferenční* omezení stanovují, která řešení mají přednost.

Každá proměnná V_i v CSP má **doménu** D_i, kterou tvoří množina možných hodnot, kterých může proměnná nabývat. Doména může být *diskrétní* nebo *spojitá*.

Např. při návrhu automobilu mohou součástky zahrnovat složku váhy (spojitou) a složku výrobce (diskrétní).

Unární omezení specifikuje povolenou podmnožinu domény.

Binární omezení mezi dvěma proměnnými určuje podmnožinu vektorového součinu dvou domén.

U diskrétních CSP s konečnými doménami mohou být domény jednoduše representovány výčtem povolených kombinací hodnot.

Např. pro 8 dam: je-li V_1 řada, na kterou působí dáma z prvního sloupce a V_2 řada, na kterou působí dáma z druhého sloupce, pak domény V_1 a V_2 jsou {1,2,3,4,5,6,7,8}. Omezení pro "neatakování" spojuje domény V_1 a V_2 a lze je representovat množinou dvojic povolených hodnot pro V_1 a V_2 :

$$\{\langle 1,3\rangle,\langle 1,4\rangle,\langle 1,5\rangle, \ldots,\langle 2,4\rangle,\langle 2,5\rangle, \ldots\}.$$

Ze 64 možných kombinací je "neatakovacím" omezením vyloučeno 22. Tato myšlenka výčtu vede k možné redukci jakéhokoliv diskrétního CSP na binární CSP. Řešení spojitých CSP vyžaduje složitější algebru.

Několik stručných doplňkových poznámek: Problém 8 dam byl prvně publikován anonymně v německém šachovém časopise *Schach* v r. 1848. V roce 1850 byl publikován znovu a zaujal pozornost matematika Carla Friedricha Gausse, který se pokusil vytvořit výčet všech možných řešení—podařilo se mu nalézt 72 z celkového počtu 92 řešení (publikovaných téhož roku později Nauckem), což svědčí o obtížnosti problému. V r. 1901 byl problém zobecněn na "šachovnici" o rozměru $n \times n$ polí. V r. 1986 bylo dokázáno, že tento problém patří mezi problémy s NP-úplnou složitostí. Uniformně-cenové hledání vynalezl v r. 1959 Dijkstra. Iterativní hledání do hloubky bylo poprvé použito v r. 1977 šachovým programem CHESS 4.5 (Slate a Atkin).