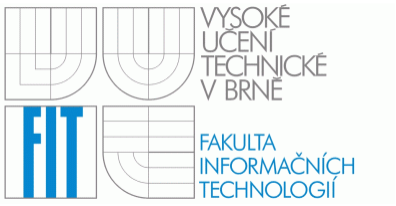
Titulní list



Základy umělé inteligence
IZU

Studijní opora

Tento učební text vznikl za podpory projektu „Zvýšení konkurenceschopnosti IT odborníků – absolventů pro Evropský trh práce“, reg. č. CZ.04.1.03/3.2.15.1/0003. Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

© doc. Ing. František Zbořil, CSc.
© Ing. František Zbořil, Ph.D.

Verze: 5. 2012

OBSAH

Obsah

1 PŘEDMLUVA

1.1 Organizační informace

Tato studijní opora je pomocným učebním textem pro stejnojmenný předmět, zahrnuje přibližně 90% přednášené látky a je zdrojem minimálního rozsahu znalostí potřebných pro úspěšné absolvování tohoto předmětu. Její text by měl být skutečnou „oporou“, to znamená, že pouze doplňuje ostatní studijní materiály předmětu a nenahrazuje knižní učebnici.

Předmět se skládá z přednášek a z vedených počítačových cvičení. Účast na přednáškách není povinná. Zkušenosti však přesvědčivě ukazují, že neúčast na přednáškách patří mezi nejčastější příčiny neúspěchu při půlsemestrální i závěrečné zkoušce.

Podrobné informace o předmětu, s aktualizacemi pro každý akademický rok, lze nalézt na adrese http://www.fit.vutbr.cz/study/courses/IZU/private/, ke které mají přístup všichni zapsaní studenti.

Zkouška a hodnocení předmětu

V počítačových cvičeních lze za předepsané aktivity získat až 20 bodů. Dalších 20 bodů lze získat při půlsemestrální písemné zkoušce. Podmínkou zápočtu je pak získání minimálně 15 bodů (z výše uvedených 40 možných bodů). Neudělení zápočtu znamená neúspěšné absolvování předmětu.

Závěrečnou písemnou zkoušku (max. 60 bodů) smí psát pouze ti studenti, kteří získali zápočet.

Přednášející a učitelé vedoucí počítačová cvičení mají právo oceňovat mimořádné aktivity studentů prémiovými body, které však nejsou nárokové.

Klasifikace úspěšných studentů se řídí studijními předpisy FIT VUT a zásadami ECTS (Evropského kreditového systému). Podmínkou úspěšného absolvování předmětu je pak získání nejméně 50 bodů ze 100 možných.

Odhad časové náročnosti

Přibližný odhad časové náročnosti předmětu (4 kredity) lze stanovit takto:

1 kredit = 25 až 30 hodin práce ⇒ 4 kredity odpovídají 100 až 120 hodinám práce studenta, z toho:

- přednášky 26 hod
- počítačová cvičení 13 hod
- průběžné studium, včetně přípravy příkladů na cvičení cca 50 hod
- příprava na půlsemestrální a závěrečnou zkoušku cca 30 hod

1.2 Metodické informace

Předmět Základy umělé inteligence je povinným předmětem ve čtvrtém semestru bakalářského studijního programu Informační technologie. Je úvodním předmětem do široké a zajímavé problematiky a získané znalosti mohou studenti využít v řadě jiných předmětů.

Znalost základních metod řešení problémů je nezbytným předpokladem dobré práce každého budoucího profesionálního programátora a odborníka v oblasti informačních technologií. Stejně důležitá je znalost principů logických a funkcionálních jazyků, základů strojového učení, obecné teorie rozpoznávání, počítačového vidění i zpracování řeči a přirozeného jazyka.

Předmět Základy umělé inteligence (pod názvem Umělá inteligence) prošel na FIT (dříve FEI) více než patnáctiletým vývojem. K tvorbě náplně předmětu sloužily především články z časopisů, protože odborné knihy s touto tématikou nebyly na konci osmdesátých let minulého století prakticky dostupné. Hlavní

studijní literaturu proto představovalo po dlouhou dobu několik různých vydání skript "Umělá inteligence", která jsou již nedostupná a jejich obsah je navíc již zastaralý. Proto uvažujeme o novém a výrazně přepracovaném vydání, které by respektovalo poslední úpravu obsahu předmětu vyplývající ze zavedení tříletého bakalářského studijního programu.

Čtenářům této studijní opory doporučujeme, aby věnovali potřebný čas k řádnému prostudování uvedených příkladů i k vyřešení všech úkolů zadaných u důležitých partií textu.

Odborná terminologie

Přesné myšlení je samozřejmým předpokladem každé inženýrské práce a musí se proto opírat o přesné pojmy a správnou terminologii. Odborný žargon, který je často živým komunikačním nástrojem každodenního života, není vhodný ani pro publikační ani pro vyspělé prezentační aktivity. Proto se i text této opory snaží o přesnou a terminologicky správnou prezentaci nových pojmů.

Anglická terminologie

V textu se budou často vyskytovat anglické termíny. Jejich první výskyty budou uváděny za příslušnými českými pojmy v závorce a budou psány *kurzívou.*

Grafická úprava

Text je psán s použitím standardního typu Times New Roman, 12 pt. Významné pojmy nebo části textu jsou zvýrazněny **tučně** a/nebo podtrženě. Doplňující poznámky jsou psány menším typem Times New Roman, 10 pt.

Učební cíle a kompetence - specifické

Studenti se seznámí s metodami řešení úloh, a to nejen v klasických jazycích, ale i v jazycích logických (PROLOG) a funkcionálních (LISP). Dále získají představu o základních schématech reprezentace znalostí, o strojovém učení a o teorii rozpoznávání, o principech počítačového vidění a o principech práce s přirozeným jazykem.

Učební cíle a kompetence - generické

Studenti získají základní vědomosti o obecných metodách řešení úloh, které patří k základním znalostem všech odborníků oboru IT.

Anotace

Řešení úloh: řešení úloh prohledáváním stavového prostoru, řešení úloh rozkladem na podúlohy, hraní her. Reprezentace znalostí. Základy jazyků PROLOG a LISP, implementace prohledávacích algoritmů v těchto jazycích. Principy strojového učení. Příznakové a strukturální rozpoznávání obrazů. Základy počítačového vidění. Základní principy práce s přirozeným jazykem. Aplikační oblasti umělé inteligence.

Přibližný rozpis přednášek

1. Úvod, typy UI úloh.
2. Metody řešení úloh
3. Metody řešení úloh, pokračování
4. Metody řešení úloh, pokračování
5. Metody hraní her
6. Logika a umělá inteligence
7. Jazyk PROLOG
8. Jazyk LISP
9. Reprezentace znalostí
10. Strojové učení
11. Rozpoznávání
12. Principy počítačového vidění
13. Principy zpracování přirozeného jazyka

Studijní literatura

1. Russel,S., Norvig,P.: Artificial Intelligence, Prentice-Hall, Inc., 1995, ISBN 0-13-360124-2, second edition 2003, ISBN 0-13-080302-2
2. Mařík,V., Štěpánková,O., Lažanský,J. a kol.: Umělá inteligence (1)+(2), ACADEMIA Praha, 1993 (1), 1997 (2), ISBN 80-200-0502-1

Poznámka

Text této studijní opory vznikal ve velmi krátkém časovém období a ve velké časové tísni. Tím mohlo dojít k různým nedokonalostem, chybám či překlepům. Autoři budou vděčni za upozornění na takové nedostatky a žádají čtenáře o jejich zaslání na adresu: zboril@fit.vutbr.cz.

2 ÚVOD

Cílem této úvodní kapitoly je diskutovat pojem *umělá inteligence*, uvést, jak bude tento pojem v dalším textu chápán, stručně seznámit čtenáře s dosavadním vývojem této disciplíny a nastínit současný stav.


0.5 hod

2.1 Definice pojmu *umělá inteligence*

Definice pojmu *umělá inteligence*

Inteligence je složitou kognitivní vlastností osobnosti, která umožňuje jedinci poznávat svět a využívat získané poznatky pro přizpůsobení se změnám prostředí. Pojem kognitivní se přitom používá pro souhrnné označení vnímání a racionálního myšlení, které zahrnuje chápavost, představivost, hodnocení, paměť a usuzování.

Existuje celá řada definic pojmu *inteligence*, obecně však nebyla přijata žádná z nich. Například ve Všeobecné encyklopedii DIDEROT jsou uvedeny dvě definice inteligence.

1. Schopnost účinně a rychle řešit na základě vlastní rozvahy obtížné nebo nové situace a problémy, nacházet podstatné prvky a jejich souvislosti a vztahy v těchto situacích, náležitě užívat výsledky vlastního myšlení, učit se ze zkušenosti.
2. Společenská vrstva intelektuálů.

Pro účely předmětu IZU lze použít pouze první z těchto definic.

Pozn.: Inteligence je zřejmě jedinou vlastností, která není dostatečně definována a přesto se přesně vyhodnocuje. Její jednotkou je tzv. inteligenční kvocient (IQ), který se zjišťuje pomocí inteligenčních testů. Jedna z definic inteligence pak dokonce zní: "Inteligence je to, co měří inteligenční test".

Umělá inteligence (*Artificial Intelligence*) samozřejmě napodobuje inteligenci přirozenou. Proto i pro pojem umělá inteligence existuje celá řada různých definic, například:

- Umělá inteligence je vlastnost uměle vytvořeného systému, který má schopnost rozpoznávat předměty a jevy, analyzovat vztahy mezi nimi a tak si vytvářet modely světa, dělat účelná rozhodnutí a předvídat jejich důsledky, řešit problémy včetně objevování nových zákonitostí a zdokonalování své činnosti [Z. Kotek a kol., 1986].
- Umělá inteligence je modelování intelektuální činnosti člověka počítačem při řešení složitých úloh, kde postup vyžaduje schopnost výběru z mnoha nebo z nezřetelně popsaných variant; též samočinné rozpoznávání tvarů nebo předmětů, usuzování z jednoho výroku na jiný, vytváření analogií mezi jednotlivými úsudky, generování a ověřování hypotéz, tvorba a uplatnění znalostí na základě přijatých vstupních dat a informací, schopnost eliminovat nepříznivé reakce na podněty z okolí a usměrňovat činnost systému v probíhajících procesech s ohledem na měnící se a často nezřetelné vnější podmínky [Všeobecná encyklopedie DIDEROT, 1999].

Pojem umělá inteligence se současně používá i pro označení vědní disciplíny, která se zabývá studiem a realizací umělé inteligence jako výše uvedené vlastnosti, a pouze takto bude tento pojem chápán v dalším textu.

Umělá inteligence (dále jen UI) jako vědní disciplína pak zastřešuje několik relativně samostatných oblastí, jejichž popisu jsou věnovány následující

kapitoly. Velmi stručná, avšak výstižná charakteristika takto chápaného pojmu UI je tato:

- Umělá inteligence je věda o tom, jak konstruovat stroje, jejichž činnost, kdyby ji vykonávali lidé (a kdybychom nevěděli, že ji vykonávají stroje!), bychom považovali za projev jejich inteligence [M. Minsky, 1967].

2.2 Stručná historie umělé inteligence

Stručná historie umělé inteligence

Za první práce spadající do problematiky UI jsou považovány práce, na jejichž základě Warren McCulloch a Walter Pitts definovali v roce 1943 umělý neuron a naznačili možnosti umělých neuronových sítí, včetně jejich učení.

V roce 1949 formuloval Donald Hebb základní pravidlo pro učení neuronových sítí, které se používá pro učení některých neuronových sítí dodnes.

V roce 1950 publikoval Alan Turing článek *Computing Machinery and Intelligence*, ve kterém formuloval principy strojového učení (*Machine learning*), genetických algoritmů (*Genetic algorithms*), posilovaného učení (*Reinforcement learning*) a především testu kvality umělé inteligence (*Turing test*).

V roce 1951 postavili Marvin Minsky a Dean Edmond v rámci svých disertačních prací (Princeton University) první neuronový počítač.

Vlastní pojem *Artificial intelligence* se poprvé objevil v názvu dvouměsíční pracovní konference *The Dartmouth Summer Research Project on Artificial Intelligence*, která se uskutečnila v roce 1956 v Dartmouth College v Hanoveru (New Hampshire, USA). Zúčastnilo se jí celkem deset vědeckých pracovníků: John McCarthy, Marvin Minsky, Claude Shannon, Nathaniel Rochester (Dartmouth College), Trenchard More (Princeton), Arthur Samuel (IBM), Ray Solomonoff, Olivier Selfridge (MIT), Allen Newell, Herbert Simon (CMU).

Konference byla věnována úvahám o možnostech počítačové simulace procesů probíhajících v mozku a je považována za počátek umělé inteligence jako samostatné vědní disciplíny.

Výše uvedení účastníci konference se na mnoho dalších let stali vůdčími pracovníky v tomto novém oboru, a to spolu se svými kolegy a studenty na MIT (Massachusetts Institute of Technology), CMU (Carnegie Mellon University), SRI (Stanford Research Institute) a IBM (International Business Machines).

Na konferenci *The Dartmouth Summer Research Project on Artificial Intelligence* byl předveden program *Logic Theorist* (Newell, Simon), který byl určen pro dokazování matematických teorémů. Tento program byl prvním programem, který místo s čísly pracoval se symboly, a je proto považován za první funkční "inteligentní" program.

Závěry konference byly velmi optimistické. Převládl zde názor, že inteligentní činnost je založena především na dedukci, kterou brzy zvládnou výkonnější počítače pomocí ad hoc technik. Účastníci konference dokonce předpovídali, že počítače nahradí veškerou lidskou intelektuální činnost nejpozději do dvaceti pěti let.

Skutečnost však byla zcela jiná. UI prošla od zmíněné konference poměrně složitým vývojem, který lze rozdělit do několika etap.

První etapa (zbytek padesátých a šedesátá léta) byla charakterizována nadšenou

prací a velkým očekáváním. Z významných výsledků této etapy stojí za zmínku zejména:

- LISP (List Processor; J. McCarthy, 1958) - nejpoužívanější programovací jazyk pro umělou inteligenci v USA.
- GPS (General Problem Solver, A. Newell, 1960), první program určený k řešení obecných úloh.
- Adaline (ADAptivní Lineární Neuron, B. Widrow, M. Hoff) – algoritmus pro učení neuronů založený na Hebbově pravidle.
- Perceptron (F. Rosenblatt, 1962) – neuronová síť pro rozpoznávání obrazu, důkaz konvergence algoritmu pro učení této sítě.
- První programy určené pro hraní her: dáma (A. Samuel, 1960), šachy (A. Newell, J. C. Shaw, H.A. Simon, 1962).
- První program pro symbolickou integraci (J. Slage, 1961).
- ELIZA (J. Weizenbaum, 1966) - první program pro zpracování přirozeného jazyka (simuloval nepřímou psychoterapii).
- A* Algoritmus (P. E. Hart, N. J. Nilsson, B. Raphael, 1968) - základní algoritmus pro řešení úloh.
- QA3 (Question-Answering System, C. Green, 1969) - jeden z prvních programů postavených na rezoluční metodě. Byl určen k řešení jednodušších úloh (pohyby robotů, hříčky).

Koncem šedesátých let se začalo zřetelně ukazovat, že problematika umělé inteligence je mnohem obtížnější, než se původně předpokládalo. Dosažené výsledky zdaleka nesplnily všechna očekávání - zcela například selhaly pokusy o strojový překlad založený pouze na slovnících a základních gramatických pravidlech. Poukazováno bývá zejména na překlad věty *The spirit is willing but the flesh is weak* do ruštiny a zpět, který údajně vedl ke větě *The vodka is good but the meat is rotten*. Stále zřetelněji se ukazovalo, že základem inteligence jsou znalosti, včetně znalostí kontextových a že při hledání řešení obtížnějších úloh jsou nezbytné heuristiky ke zvládnutí prohledávání kombinatorické exploze možných stavů těchto úloh.

Druhá etapa vývoje umělé inteligence (sedmdesátá léta) se proto označuje jako období návratu k realitě, někdy také jako období stagnace (podpora výzkumu UI byla v tomto období výrazně omezena). Přesto i v této etapě bylo dosaženo významných výsledků, mezi které především patří:

- SRI Vision Module (G. J. Agin, G. J. Gleason, 1970) - prototyp "vidícího" sytému pro průmyslové použití (rozpoznávání součástek podle obrysů).
- STRIPS (R. Fikes, N. J. Nilsson, 1971) - program/jazyk určený k řešení úloh. Byl určen pro plánování činnosti robota Shakey.
- DENDRAL (E.A. Feigenbaum, 1971) - první expertní systém. Byl určen pro identifikaci organických sloučenin.
- SHRDLU (T. Winograd, 1972) - program pro komunikaci v přirozeném jazyku o jednoduchém světě kostek (barvy kostek, vzájemné pozice ap.). Pracoval na základě syntaktické a sémantické analýzy.
- PROLOG (PROgramming in LOGic; A. Colmerauer, 1972, P. Roussel, 1975, D. H. D. Warren, L. M. Pereira, F. Pereira, 1977) - nejpoužívanější jazyk pro umělou inteligenci v Evropě a v Japonsku.
- MYCIN (E. H. Shortlife, 1976) - expertní systém pro diagnostiku infekčních onemocnění a návrh terapie.

- PROSPECTOR (R. Duda, P. Hart, 1978) - expertní systém pro hledání ložisek rud.
- HEARSAY II (L.D. Erman a kol., 1980) - systém pro rozpoznávání řeči se slovníkem asi 1000 slov.

Třetí etapa (léta osmdesátá) se vyznačovala obnovením zájmu o problematiku umělé inteligence a bývá charakterizována těmito rysy:

- Výsledky UI se staly komerčními (expertní systémy pro různé oblasti lidské činnosti, analyzátory a syntetizátory řeči, systémy pro zpracování přirozeného jazyka, systémy pro zpracování obrazů, ap.).
- UI se stala uznávanou vědou - byla vytvořena odborná terminologie a byly publikovány příručky a monografie k jednotlivým oblastem UI (expertní systémy, počítačové vidění, zpracování přirozeného jazyka ap.).
- Byly navrženy počítače s architekturami vhodnými pro umělou inteligenci (LISPovské a PROLOGovské počítače).
- Byly realizovány rozsáhlé projekty s využitím výsledků umělé inteligence, především japonský projekt páté generace počítačových systémů.
- Po značném útlumu v sedmdesátých letech se naplno obnovil výzkum problematiky neuronových sítí.

Za zmínku stojí i skutečnost, že v roce 1987 umělý systém, resp. program (HITECH - H. Berliner, C. Ebeling), poprvé porazil v šachové hře mezinárodního šachového velmistra.

Poslední etapa vývoje umělé inteligence trvá dodnes. Došlo v ní k překonání izolace UI od ostatních počítačových věd a pozornost výzkumných pracovníků se zaměřila na práci s nejistými a neúplnými informacemi, na tzv. *Softcomputig* (zahrnuje neuronové sítě, genetické algoritmy, fuzzy množiny a fuzzy logiku, hrubé množiny, chaos). V současné době je také intenzívně zkoumána problematika distribuované UI, tj. problematika agentů a multiagentních systémů. Poslední etapu vývoje UI lze však charakterizovat především tím, že UI masivně přenáší z výzkumných laboratoří do reálného světa.

Současné aplikační oblasti UI jsou například tyto:

- Autonomní plánování (roboti NASA na mimozemských objektech).
- Hraní her (šachový program Deep Blue firmy IBM).
- Autonomní řízení (roboti NASA, automobily).
- Medicínská diagnostika (expertní systémy zaměřené na tuto problematiku pracují již na úrovni specialistů z několika různých oblastí medicíny).
- Logistické plánování (bylo prakticky použito například při plánování logistických operací během války v Perském zálivu v roce 1991).
- Robotika (komplexní využití poznatků UI).
- Zpracování přirozeného jazyka (spotřebiče ovládané hlasem).
- Řešení složitých úloh (například luštění křížovek).

2.3 Shrnutí

Shrnutí

V úvodní kapitole byl diskutován pojem umělá inteligence a bylo zde uvedeno, že v dalším textu bude tento pojem chápán jako označení vědní disciplíny, která se zabývá studiem a realizací umělé inteligence jako vlastnosti umělých systémů. Dále byl v této kapitole stručně popsán dosavadní vývoj této disciplíny.

3 METODY ŘEŠENÍ ÚLOH

30 hod

Text kapitoly je psán s cílem seznámit studenty s nejznámějšími metodami řešení úloh UI. Jsou zde popsány metody založené na prohledávání stavového prostoru (neinformované, informované, metody lokálního prohledávání), metody řešení úloh, které musí respektovat zadané omezující podmínky, metody založené na rozkladech úloh na podproblémy a metody hraní her dvou antagonistických protihráčů. Jde o významnou problematiku, která představuje jednu z klíčových oblastí klasické UI.

3.1 Úvodní informace

Úvodní informace

Klasická UI úloha je definovaná počátečním stavem, množinou cílových stavů a množinou operátorů, které umožňují stavy této úlohy měnit. Řešením úlohy je nalezení posloupnosti operátorů, jejichž postupnou aplikací se dostaneme z počátečního stavu do některého z množiny cílových stavů. Často je nutné výchozí úlohu rozdělit na jednodušší podúlohy a ty potom řešit samostatně.

Z dnešního pohledu jde o úlohu v prostředí plně pozorovatelném (v každém časovém okamžiku můžeme zjistit úplný stav tohoto prostředí), deterministickém (následující stav prostředí je určen pouze aktuálním stavem tohoto prostředí a použitým operátorem), statickém (stavy lze měnit výhradně definovanými operátory) a diskrétním (stavy se mění v diskrétních časových okamžicích).

Prvním úkolem při řešení každé úlohy je jednoznačná formulace jejích cílů a jednoznačná definice operátorů, která zahrnuje i případné podmínky omezující jejich použití. Metody řešení úloh nám pak nabízejí postupy, kterými lze cílové stavy, resp. posloupnosti operátorů vedoucí k cílovým stavům, nalézat a představují tak prostředky nepostradatelné ve všech aplikačních oblastech UI.

V této studijní opoře se budeme zabývat pouze jednoduchými úlohami, jejichž formulace je snadná a většinou i známá. U takových úloh můžeme zaměřit svou pozornost pouze na principy jednotlivých metod, aniž bychom museli ztrácet čas potřebný k pochopení podstaty složitějších problémů.

Z důvodů víceméně formálních si pro další výklad rozdělíme metody řešení úloh do čtyř skupin, a to na:

- Metody založené na prohledávání stavového prostoru (*State Space Search Methods*)
- Metody řešení úloh s omezujícími podmínkami (*Constraint Satisfaction Methods*)
- Metody založené na rozkladu úloh/problémů na podproblémy (*Problem Reduction Methods*)
- Metody hraní her (*Game Playing Methods*)

3.2 Metody řešení úloh založené na prohledávání stavového prostoru

Metody řešení úloh založené na prohledávání stavového prostoru

Stavový prostor definován dvojicí (S,O), kde symbol S (*States*) označuje množinu všech možných stavů úlohy $S = \{s_1, s_2, s_3, \ldots\}$ a symbol O (*Operators*) množinu všech operátorů $O = \{o_1, \ldots o_j\}$, kterými lze stavy úlohy měnit.

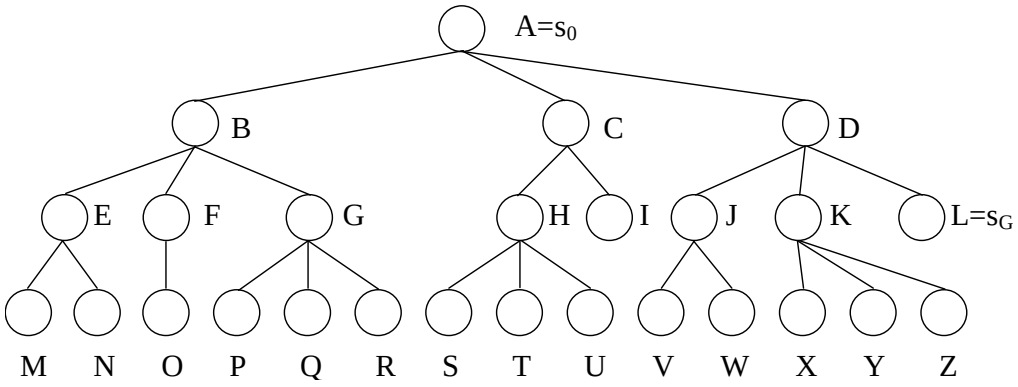
Úloha je definována dvojicí (s_0, G), kde symbol $s_0 \in S$ značí počáteční stav a symbol $G \subset S$ množinu cílových stavů této úlohy (*Goals*).

Řešením úlohy je posloupnost operátorů $s_1 = o_1(s_0)$, $s_2 = o_2(s_1)$, …, $s_n = o_n(s_n\text{-}1)$, $s_n \in G$.

Stavový prostor si můžeme představit jako orientovaný graf/strom, jehož uzly představují jednotlivé stavy úlohy a jehož hrany reprezentují přechody mezi těmito stavy způsobené definovanými operátory. Cesta z počátečního stavu do některého cílového stavu je zřejmě řešením úlohy. V řadě úloh je nutné minimalizovat cenu této cesty, která se získá součtem cen jednotlivých přechodů. U některých úloh naopak cesta není vůbec důležitá a důležitý je pouze cílový stav (optimalizační úlohy). V dalším textu budeme předpokládat, že ceny přechodů jsou dány kladnými čísly.

Pro jednotlivé uzly stromu (Obr. 3.1) budeme používat následující terminologii:

- uzel **A** je uzel kořenový,
- uzly **I**, **L**, **M, …, Z** jsou uzly listové,
- uzel **C** je bezprostředním předchůdcem uzlu **H**, atp.,
- uzly **A**, **D**, **J** jsou předchůdci uzlu **V**, atp.,
- uzel **K** je bezprostředním následníkem uzlu **D**, atp.,
- uzly **H**, **I**, **S**, **T, U** jsou následníci uzlu **C**, atp.,
- uzel **A** má hloubku 0, uzly **B**, **C**, **D** mají hloubku 1, atp.,
- expanzí uzlu se rozumí určení všech jeho bezprostředních následníků,
- generací uzlu se nazývá proces jeho vytvoření,
- ohodnocením uzlu je dána součtem cen přechodů z kořenového do tohoto uzlu,
- uzel **A** označuje počáteční stav (s_0),
- uzel **L** označuje cílový stav (s_G).



Obr. 3.1 Demonstrační stavový prostor

3.2.1 Typy úloh

Typy úloh - úlohy, které lze řešit prohledáváním jejich stavového prostoru

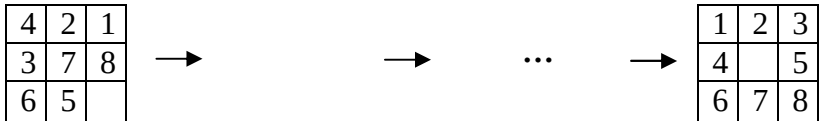
K demonstraci principů metod založených na prohledávání stavového prostoru bude v dalším textu použito několik klasických úloh:

- Úloha dvou džbánů (*Two water jugs problem*). Úloha byla formulována ve 13. století a jejím původním cílem bylo naplnit pětilitrový džbán pomocí tříllitrového džbánu čtyřmi litry vody, když přípustné operace byly následující: 1) úplné naplnění libovolného džbánu z neomezeného zdroje vody, 2) úplné vyprázdnění libovolného džbánu, 3) vzájemné přelévání, a to buď do úplného naplnění cílového džbánu, nebo do úplného vyprázdnění

zdrojového džbánu. Úloha byla později rozšířena na naplnění dvou džbánů různých velikostí předepsaným množstvím vody.

Pozn.: Důkaz řešitelnosti rozšířené úlohy lze nalézt v: *Ptaff, T. J., Tran, M. M.: The Generalized Jug Problem, Journal of Recreational Mathematics Vol 31, Num 2, 2002-2003.* Obecně existují stavy, kterých není možné dosáhnout, a proto konkrétní úloha nemusí být řešitelná (představují-li nedosažitelné stavy jediné cíle této úlohy).

- Hlavolam "8", resp. hlavolam "15" (*8-puzzle, 15-puzzle*). Cílem je dosažení zvoleného cílového stavu z daného počátečního stavu, a to postupným posouváním kamenů přes volná políčka, například:



Pozn.: Zvolený cílový stav není z libovolného počátečního stavu dosažitelný – všechny možné stavy hlavolamu tvoří dva disjunktní stavové prostory.

- Úloha N dam (*N Queens problem*). Úlohu zformuloval v roce 1848 šachista M. Bazzel. Cílem úlohy je postavit N dam na šachovnici o rozměrech N x N tak, aby žádná dáma neohrožovala jinou dámu.

  Pozn.: Úloha není řešitelná pro N < 4, zajímavou se stává pro N >= 8.

- Úloha nalezení nejkratší cesty (*Shortest path problem*). Hledá se nejkratší cesta mezi dvěma různými místy.
- Úloha obchodního cestujícího (*TSP - Traveling salesman problem*). Obchodní cestující musí navštívit N měst, každé pouze jedenkrát, a vrátit se do výchozího města tak, aby celková délka cesty byla minimální.

Pozn.: K dalším praktickým úlohám, které jsou řešitelné dále uvedenými metodami, patří například úlohy barvení map, rozmísťování prvků VLSI obvodů, navigace robotů, atd.

3.2.2 Formulace úloh

Formulace úloh

Řádná formulace řešené úlohy, především operátorů pro změnu stavů, je velmi důležitá a mnohdy na ni závisí i úspěch či neúspěch vlastního řešení. Jako názorné příklady mohou sloužit dva různé přístupy k řešení úlohy osmi dam a dva různé přístupy k řešení hlavolamu "8".

Úloha osmi dam:

1. s_0 = prázdná šachovnice
   o_{ij} = postavení volné dámy Q_i ($i \in <1,8>$) na libovolnou volnou pozici p_{ij} šachovnice ($j = 65 - i$)
   s_G = 8 dam na šachovnici, žádná neohrožuje jinou
2. s_0 = prázdná šachovnice
   o_{ij} = postavení volné dámy Q_i ($i \in <1,8>$) na libovolný volný řádek p_{ij} ve sloupci i šachovnice ($j \in <1,8>$)
   s_G = 8 dam na šachovnici, žádná neohrožuje jinou

V prvním případě má stavový prostor úlohy $64 \cdot 63 \cdot 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57 \approx 2 \cdot 10^{14}$ různých stavů, zatímco ve druhém případě se tento prostor zredukuje na $8^8 \approx 2 \cdot 10^7$ různých stavů.

Hlavolam “8”:

V případě hlavolamu “8” má stavový prostor 9!/2 = 181440 různých stavů (zmínili jsme se již dříve, že prostor všech možných stavů je rozdělen na dva disjunktní stavové prostory) a zvolené operátory nemají na velikost tohoto prostoru žádný vliv. Budeme-li uvažovat čtyři různé tahy pro každou kostku (nahoru, dolů, doleva, doprava) pak budeme mít celkem 32 operátorů, jejichž použitelnost musíme testovat v každém tahu. Stejného výsledku prohledávání dosáhneme, budeme-li místo kostek „pohybovat“ prázdným políčkem, avšak počet operátorů se nám v tomto případě zredukuje na pouhé čtyři operátory.

Pozn.: 181 tisíc stavů lze samozřejmě snadno prohledat. Pro hlavolam “15” již však má stavový prostor $16!/2 \approx 10^{13}$ různých stavů, a hlavolam ”24“ dokonce $25!/2 \approx 7.8 \cdot 10^{24}$ různých stavů. Zatímco hlavolam “15” je pomocí dnešních počítačů již zvládnutelný, hlavolam “24” ještě obecně řešitelný není.

3.2.3 Hodnotící kritéria

Hodnotící kritéria

Metody řešení úloh se hodnotí podle čtyř kritérií, kterými jsou:

1. Úplnost (nalezne metoda řešení, pokud toto existuje?).
2. Časová náročnost.
3. Paměťová náročnost.
4. Optimálnost (nalezne metoda nejlepší řešení?).

3.2.4 Přehled metod

Přehled metod řešení úloh založených na prohledávání stavového prostoru

Metody prohledávání stavového prostoru se dělí na dvě základní skupiny:

- Neinformované/slepé (*Uninformed/Blind Search*)
- Informované (*Informed Search*)

Neinformované (slepé) metody:

- Metoda prohledávání do šířky (*BFS - Breadth First Search*)
- Metoda stejných cen (*UCS - Uniform Cost Search*)
- Metoda prohledávání do hloubky (*DFS - Depth First Search*)
- Metoda omezeného prohledávání do hloubky (*DLS - Depth Limited Search*)
- Metoda postupného zanořování do hloubky (*IDS - Iterative deepening DFS*)
- Metoda zpětného navracení (*Backtracking*)
- Metoda obousměrného prohledávání (*BS - Bidirectional BFS search*)

Informované metody (do češtiny se jejich názvy obvykle nepřekládají):

- Metody *Best First Search*:
  - Metoda *Greedy search,*
  - Metoda A^* *search.*
- Metody lokálního prohledávání (*Local search*)
  - Metoda *Hill-climbing,*
  - Metoda simulovaného žíhání (*Simulated annealing*),
  - Metody založené na genetických algoritmech (*Genetic algorithms*).

3.2.5 Neinformované metody

Neinformované metody

Neinformované metody nemají k dispozici žádnou informaci o cílovém stavu a nemají ani žádné prostředky, jak aktuální stavy hodnotit. Podobné metody musí někdy používat i člověk - například při prohledávání mapy, hledá-li cestu z nějakého (výchozího) místa do jiného (cílového) místa a nemá-li vůbec žádnou představu, ve kterém směru od výchozího místa cílové místo leží.

Metoda BFS **Metoda slepého prohledávání do šířky (BFS)**

Základní algoritmus metody BFS je následující :

1. Sestroj frontu OPEN (bude obsahovat všechny uzly určené k expanzi) a umísti do ní počáteční uzel.
2. Je-li fronta OPEN prázdná, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Vyber z čela fronty OPEN první uzel.
4. Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů, nebo operátorů). Jinak pokračuj.
5. Vybraný uzel expanduj, všechny jeho bezprostřední následníky umísti do fronty OPEN a vrať se na bod 2.

Je-li počet bezprostředních následníků každého uzlu konečný, pak algoritmus BFS je úplný a optimální. Časová i paměťová náročnost algoritmu BFS je však exponenciální - je dána výrazem $O(b^{d+1})$, kde *b* je tzv. faktor větvení (*branching factor*), tj. průměrný počet bezprostředních následníků každého uzlu, a *d* (*depth*) je hloubka nejlepšího řešení, tj. řešení, které se nachází v nejmělčí hloubce.

Pozn.: Metodu BFS lze modifikovat testováním cílového stavu již při expanzi uzlu. Pak se body 4 a 5 sloučí v jediný:

4. Vybraný uzel expanduj. Pokud je některý z jeho bezprostředních následníků uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů, nebo operátorů). Jinak umísti všechny bezprostřední následníky do fronty OPEN a vrať se na bod 2.

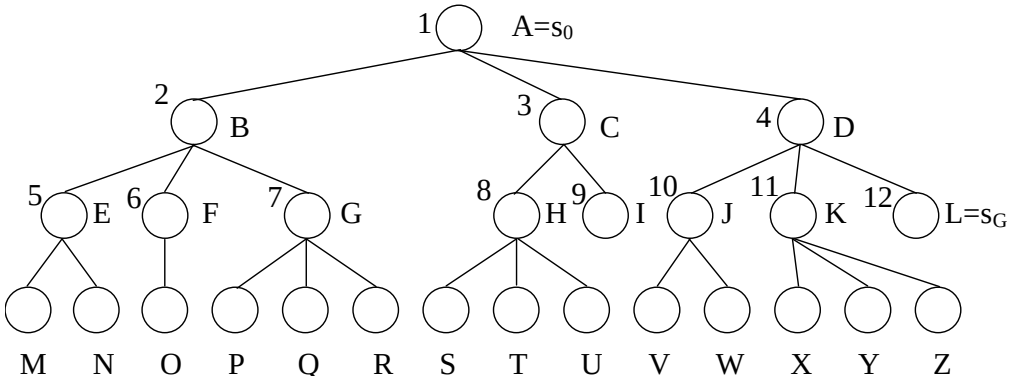
Časová i paměťová náročnost modifikované metody se sice redukuje na $O(b^d)$, nicméně metoda přestává být „kompatibilní“ s následujícími metodami. Z tohoto důvodu a z důvodů uvedených níže nebudeme uvedenou modifikaci metody BFS dále uvažovat.

Na Obr. 3.2 je ukázáno pořadí výběru a expanze uzlů z fronty OPEN pro demonstrační prostor z Obr. 3.1 (čísla u uzlů ukazují pořadí jejich expanze).

I když je algoritmus BFS velmi jednoduchý a průzračný, je pro svou časovou a paměťovou náročnost pro řešení složitějších úloh prakticky nepoužitelný. Přesto je algoritmus BFS považován za základní algoritmus. Navíc jeho praktické použití na jednoduchých úlohách ukazuje na jeden závažný a obecný problém, kterým je opakované generování již expandovaných uzlů.

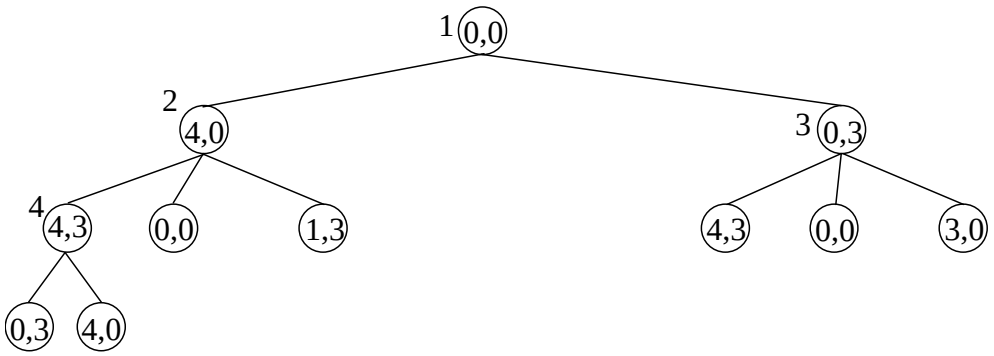
Zmíněný problém budeme demonstrovat na řešení úlohy dvou džbánů, kdy pro jednoduchost budeme uvažovat menší džbány: nechť větší má obsah 4 litry a menší 3 litry (stavový prostor obsahuje celkem 20 stavů (0,0), (0,1), …, (4,3)). Nechť počáteční stav úlohy je definován prázdnými džbány a nechť cílovým stavem je naplnění některého ze džbánů dvěma litry vody $(0,0)\rightarrow\{(0,2),(2,0)\}$; poznamenejme, že pro daný počáteční stav je 14 stavů dosažitelných a 6 stavů nedosažitelných. Operátory se budeme snažit aplikovat v pořadí: $\rightarrow$V, $\rightarrow$M, V$\rightarrow$, M$\rightarrow$, M$\rightarrow$V, V$\rightarrow$M (naplň velký, naplň malý, vyprázdni velký, vyprázdni malý, plň velký z malého, plň malý z velkého).

Situace pro čtyři první kroky je znázorněna na Obr. 3.3 (čísla v uzlech označují stavy, čísla mimo udávají pořadí expanze uzlu; pro jednoduchost nejsou u uzlů uváděni jejich předchůdci). Uvedené čtyři kroky stačí k tomu, aby ukázaly, že problém opakovaného generování již expandovaných uzlů je velmi významný - opakovaně generované uzly mají navíc stále více předchůdců.



Pořadí expanze	Fronta OPEN
0	[[A,-]]
1	[[B,A,-],[C,A,-],[D,A,-]]
2	[[C,A,-],[D,A,-],[E,B,A,-],[F,B,A,-],[G,B,A,-]]
3	[[D,A,-],[E,B,A,-],[F,B,A,-],[G,B,A,-],[H,C,A,-], [I,C,A,-]]
	…
	…
11	[[L,D,A,-],[M,E,B,A,-], …,[T,H,C,A,-],…,[Z,K,D,A,-]]
12	[[M,E,B,A,-], …,[Z,K,D,A,-]]
Řešení	[L,D,A,-]: A ⇒ D ⇒ L

Obr. 3.2 Prohledávání demonstračního stavového prostoru metodou BFS



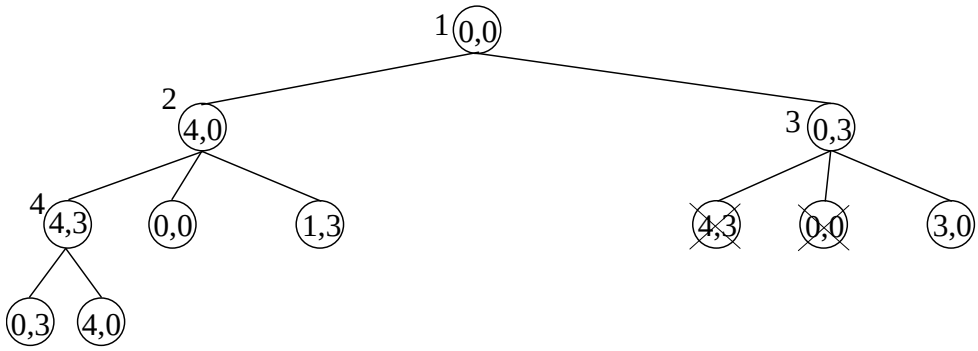
Pořadí expanze	Fronta OPEN
0	[(0,0)]
1	[(4,0),(0,3)]
2	[(0,3),(4,3),(0,0),(1,3)]
3	[(4,3),(0,0),(1,3),(4,3),(0,0),(3,0)]
4	[(0,0),(1,3),(4,3),(0,0),(3,0),(0,3),(4,0)]
5	….

Obr. 3.3 Prohledávání stavového prostoru úlohy dvou džbánů metodou BFS

Částečné řešení zmíněného problému opakovaného generování již expandovaných uzlů spočívá v úpravě bodu 5 algoritmu BFS – do fronty OPEN se neukládají generované uzly, které se již v této frontě nacházejí:

5. Vybraný uzel expanduj, do fronty OPEN umísti všechny jeho bezprostřední následníky, kteří v ní ještě nejsou, a vrať se na bod 2.

Situace pro první čtyři kroky algoritmu BFS s eliminací generovaných uzlů, které se již ve frontě OPEN nacházejí, je ukázána na Obr. 3.4. Je zřejmé, že algoritmus pracuje výrazně efektivněji, přesto se však do fronty OPEN opět ukládají (a později pak expandují) uzly, které již dříve expandovány byly.



Pořadí expanze	Fronta OPEN
0	[(0,0)]
1	[(4,0),(0,3)]
2	[(0,3),(4,3),(0,0),(1,3)]
3	[(4,3),(0,0),(1,3),(3,0)]
4	[(0,0),(1,3),(3,0),(0,3),(4,0)]
5	…

Obr. 3.4 Prohledávání stavového prostoru úlohy dvou džbánů metodou BFS s eliminací stejných stavů v OPEN

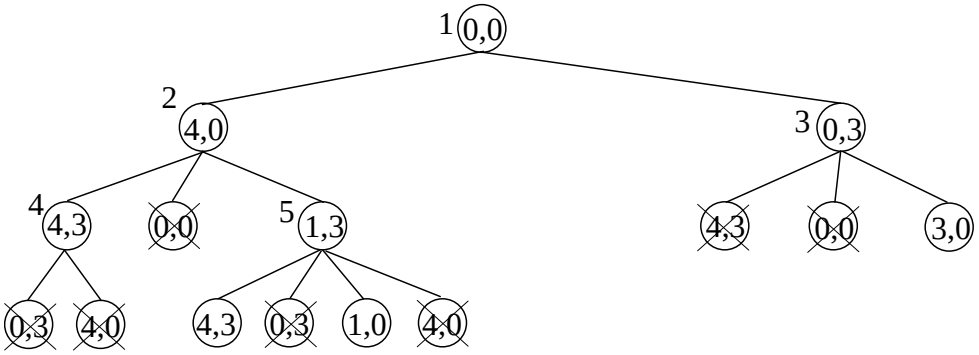
Další „vylepšení“ algoritmu BFS spočívá v další úpravě bodu 5 algoritmu – do fronty OPEN se neukládají generované uzly, které se již v této frontě nacházejí a uzly, které jsou předchůdci generovaného uzlu:

5. Vybraný uzel expanduj, do fronty OPEN umísti všechny jeho bezprostřední následníky, kteří v této frontě ještě nejsou a kteří nejsou předky generovaného uzlu, a vrať se na bod 2.

Situace pro prvních pět kroků algoritmu BFS s eliminací generovaných uzlů, které se již ve frontě OPEN nacházejí, a s eliminací předků generovaných uzlů je ukázána na Obr. 3.5. Poznamenejme, že v tomto případě již nemůžeme opomenout uvádění předchůdců uzlů. Je zřejmé, že algoritmus pracuje opět efektivněji, přesto však i nyní dochází k ukládání a poté k expanzi uzlů, které již dříve expandovány byly (viz uzel (4,3)).

K úplnému zabránění opakování expanzí dříve expandovaných uzlů je nutné použít další seznam, do kterého se budou expandované uzly postupně ukládat. Takový seznam se nazývá seznamem CLOSED a kromě původního účelu umožňuje, aby spolu u generovaných uzlů byly zaznamenány pouze jejich

bezprostřední předchůdci místo všech předchůdců. K rekonstrukci řešení (cesty stromem) totiž stačí postupný průchod uzly v seznamu CLOSED a jejich bezprostředními následníky tak, jak je naznačeno na Obr. 3.6.



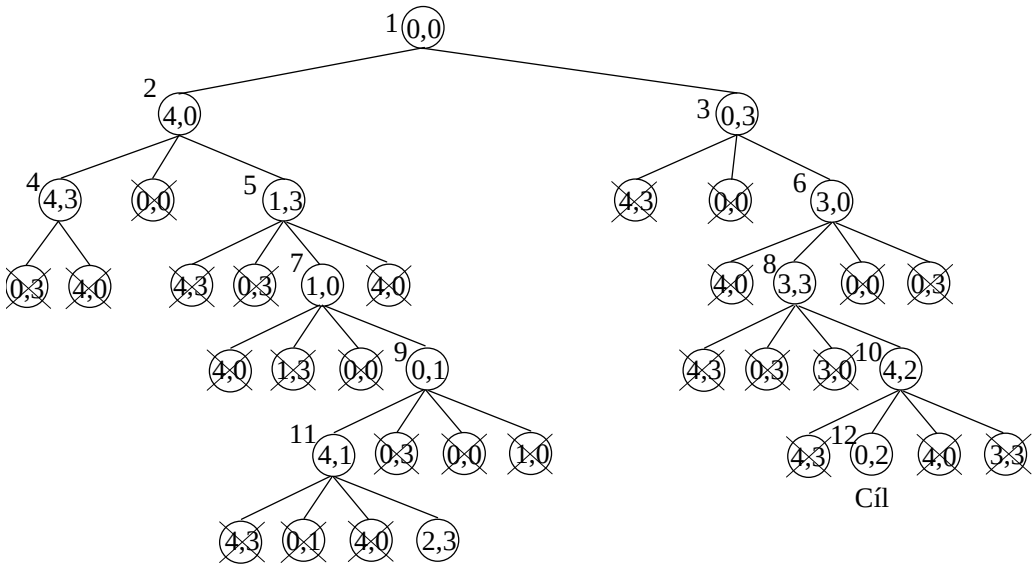
Pořadí expanze	Fronta OPEN
0	[(0,0)]
1	[((4,0),(0,0),-),((0,3),(0,0),-)]
2	[((0,3),(0,0),-),((4,3),(4,0),(0,0),-),((1,3),(4,0),(0,0),-)]
3	[((4,3),(4,0),(0,0),-),((1,3),(4,0),(0,0),-),((3,0),(0,3),(0,0),-)]
4	[((1,3),(4,0),(0,0),-),((3,0),(0,3),(0,0),-)]
5	[((3,0),(0,3),(0,0),-),((4,3),(1,3),(4,0),(0,0),-), ((1,0),(1,3),(4,0),(0,0),-)]
6	…

Obr. 3.5 Prohledávání stavového prostoru úlohy dvou džbánů metodou BFS s eliminací stejných stavů v OPEN a s eliminací předků generovaných uzlů

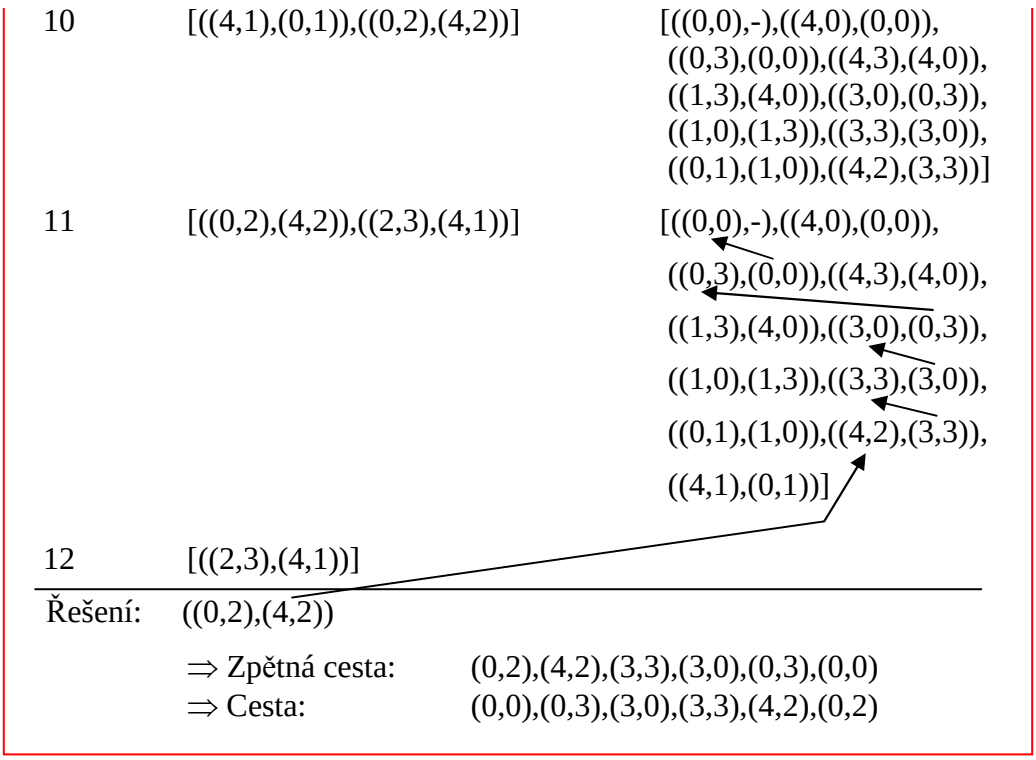
Algoritmus BFS s uvažováním seznamu CLOSED je následující:

1. Sestroj dva prázdné seznamy, frontu OPEN (bude obsahovat všechny uzly určené k expanzi) a CLOSED (bude obsahovat seznam expandovaných uzlů). Do fronty OPEN umísti počáteční uzel.
2. Je-li fronta OPEN prázdná, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Vyber z čela fronty OPEN první uzel a umísti tento uzel do seznamu CLOSED.
4. Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů). Jinak pokračuj.
5. Vybraný uzel expanduj a jeho bezprostřední následníky, kteří nejsou ani ve frontě OPEN, ani v seznamu CLOSED, umísti do fronty OPEN a vrať se na bod 2.

Poznamenejme, že přestože na první pohled se jeví úpravy algoritmu BFS jako přínosné, porovnávání generovaných uzlů s uzly ve frontě OPEN, s uzly (předchůdci) uloženými přímo v seznamu generovaného uzlu i s uzly v seznamu CLOSED jsou výpočetně poměrně náročné operace, které značně prodlužují doby výpočtů.



Pořadí Expanze	Fronta OPEN	Seznam CLOSED
0	[((0,0),-)]	
1	[((4,0),(0,0)), ((0,3),(0,0))]	[((0,0),-)]
2	[((0,3),(0,0)),((4,3),(4,0)), ((1,3),(4,0))]	[((0,0),-),((4,0),(0,0))]
3	[((4,3),(4,0)),((1,3),(4,0)) ((3,0),(0,3))]	[((0,0),-),((4,0),(0,0)), ((0,3),(0,0))]
4	[((1,3),(4,0)),((3,0),(0,3))]	[((0,0),-),((4,0),(0,0)), ((0,3),(0,0)),((4,3),(4,0))]
5	[((3,0),(0,3)),((1,0),(1,3))]	[((0,0),-),((4,0),(0,0)), ((0,3),(0,0)),((4,3),(4,0)), ((1,3),(4,0))]
6	[((1,0),(1,3)),((3,3),(3,0))]	[((0,0),-),((4,0),(0,0)), ((0,3),(0,0)),((4,3),(4,0)), ((1,3),(4,0)),((3,0),(0,3))]
7	[((3,3),(3,0)),((0,1),(1,0))]	[((0,0),-),((4,0),(0,0)), ((0,3),(0,0)),((4,3),(4,0)), ((1,3),(4,0)),((3,0),(0,3)), ((1,0),(1,3))]
8	[((0,1),(1,0)),((4,2),(3,3))]	[((0,0),-),((4,0),(0,0)), ((0,3),(0,0)),((4,3),(4,0)), ((1,3),(4,0)),((3,0),(0,3)), ((1,0),(1,3)),((3,3),(3,0))]
9	[((4,2),(3,3)),((4,1),(0,1))]	[((0,0),-),((4,0),(0,0)), ((0,3),(0,0)),((4,3),(4,0)), ((1,3),(4,0)),((3,0),(0,3)), ((1,0),(1,3)),((3,3),(3,0)), ((0,1),(1,0))]



Obr. 3.6 Prohledávání stavového prostoru úlohy dvou džbánů metodou BFS s použitím seznamu CLOSED a rekonstrukce řešení (cesty stromem)

Úkol: Vyřešte pomocí všech uvedených modifikací algoritmu BFS následující úlohu hlavolamu "8", jestliže za operátory zvolíme operátory posouvající volné políčko v pořadí nahoru, doprava, dolů a doleva:

1	4	2
8		3
7	6	5

→

1	2	3
8		4
7	6	5

Metoda UCS **Metoda stejných cen (UCS)**

Metoda UCS pracuje s podobným algoritmem jako metoda BFS, uvažuje však skutečné ceny přechodů (kladná čísla) a skutečné ceny cest (jsou dány součtem cen příslušných přechodů). Pro expanzi pak vybírá ze seznamu OPEN uzel s nejmenším ohodnocením, tj. uzel s nejnižší cenou cesty.

Základní algoritmus metody UCS je následující :

1. Sestroj seznam OPEN (bude obsahovat všechny uzly určené k expanzi) a umísti do něj počáteční uzel včetně jeho (nulového) ohodnocení.
2. Je-li seznam OPEN prázdný, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Vyber ze seznamu OPEN uzel s nejnižším ohodnocením.
4. Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů, nebo operátorů). Jinak pokračuj.
5. Vybraný uzel expanduj, všechny jeho bezprostřední následníky včetně jejich ohodnocení umísti do seznamu OPEN a vrať se na bod 2.

Pro hodnocení algoritmu UCS platí stejné závěry, jako pro algoritmus BFS. Je-li počet bezprostředních následníků každého uzlu konečný, pak algoritmus UCS je úplný a optimální, jeho časová i paměťová náročnost je exponenciální - je dána výrazem $O(b^k)$, kde b je faktor větvení a k je koeficient získaný podílem ceny optimálního řešení C^* a nejmenšího přírůstku ceny mezi dvěma uzly Δc_{min}:

$$k = C^* / \Delta c_{min}$$

Varianty algoritmu UCS jsou podobné, jako varianty algoritmu BFS. Opět spočívají buď v modifikacích bodu 5 algoritmu UCS:

5. Vybraný uzel expanduj, všechny jeho bezprostřední následníky umísti do seznamu OPEN, a to včetně jejich ohodnocení. Z uzlů, které se v seznamu OPEN vyskytují vícekrát, ponech pouze uzel s nejlepším ohodnocením, ostatní ze seznamu OPEN vyškrtni a vrať se na bod 2.

resp.

5. Vybraný uzel expanduj, všechny jeho bezprostřední následníky, kteří nejsou jeho předky, umísti do seznamu OPEN, a to včetně jejich ohodnocení. Z uzlů, které se v seznamu OPEN vyskytují vícekrát, ponech pouze uzel s nejlepším ohodnocením, ostatní ze seznamu OPEN vyškrtni, a vrať se na bod 2.

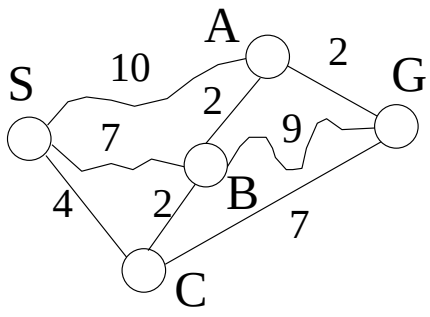
nebo v použití seznamu CLOSED:

1. Sestroj dva prázdné seznamy, OPEN (bude obsahovat uzly určené k expanzi) a CLOSED (bude obsahovat seznam expandovaných uzlů). Do seznamu OPEN umísti počáteční uzel včetně jeho ohodnocení.
2. Je-li seznam OPEN prázdný, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Vyber ze seznamu OPEN uzel s nejlepším (tj. nejmenším) ohodnocením a umísti tento uzel do seznamu CLOSED.
4. Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů). Jinak pokračuj.
5. Vybraný uzel expanduj a jeho bezprostřední následníky, kteří nejsou ani v seznamu OPEN ani v seznamu CLOSED, umísti do seznamu OPEN. Z uzlů, které se v seznamu OPEN vyskytují vícekrát, ponech pouze uzel s nejlepším ohodnocením, ostatní ze seznamu OPEN vyškrtni a vrať se na bod 2.

Použití metody UCS je demonstrováno na úloze hledání nejkratší cesty (Obr. 3.7), kde je použit algoritmus eliminující uzly, které jsou v seznamu OPEN s vyšším ohodnocením a s eliminací uzlů, které jsou předchůdci generovaných uzlů. Uzly s nejlepším ohodnocením, které se v jednotlivých krocích vybírají k expanzi, jsou na obrázku vyznačeny tučně.

Poznamenejme, že metoda UCS není vhodná pro úlohy, kdy cesta z výchozího do cílového uzlu prochází pouze několika málo jinými uzly s vysokými cenami přechodů, protože pak prohledává zbytečně mnoho cest s nízkými cenami (například při hledání cesty z Brna do Plzně přes jediný uzel (Prahu) dojde k prohledávání cest Brno-Česká, Brno-Jinačovice, Brno-Ostopovice, …., Česká-Lelekovice, Česká-Kuřim, atd.).

Čísla u přechodů značí jejich ceny. Úlohou je nalézt nejkratší cestu z místa startu S do cílového místa G.



Pořadí expanze	Seznam OPEN
0	[((S),0)]
1	[((A,S),10),((B,S),7),**((C,S),4)**]
2	[((A,S),10),**((B,C,S),6)**,((G,C,S),11)]
3	[((G,C,S),11),**((A,B,C,S),8)**]
4	[**((G,A,B,C,S),10)**] = cíl ⇒ cesta = (S,C,B,A,G)

Obr. 3.7 Prohledávání stavového prostoru úlohy nejkratší cesty metodou UCS s eliminací uzlů, které jsou v OPEN s vyšším ohodnocením a s eliminací uzlů, které jsou předchůdci generovaných uzlů.

Úkol: Vyřešte pomocí všech uvedených modifikací algoritmu UCS úlohu dvou džbánů definovanou výše, když cena každého přechodu bude dána počtem litrů vody, se kterou se v tomto přechodu manipuluje.

Metoda DFS

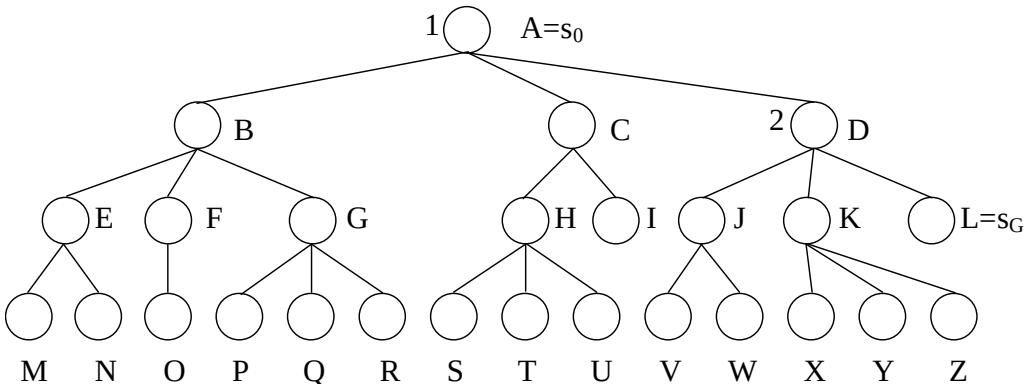
Metoda slepého prohledávání do hloubky (DFS)

Základní algoritmus metody DFS je následující :

1. Sestroj zásobník OPEN (bude obsahovat všechny uzly určené k expanzi) a umísti do něj počáteční uzel.
2. Je-li zásobník OPEN prázdný, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Vyber z vrcholu zásobníku OPEN první uzel.
4. Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů, nebo operátorů). Jinak pokračuj.
5. Vybraný uzel expanduj, všechny jeho bezprostřední následníky umísti do zásobníku OPEN a vrať se na bod 2.

Algoritmus DFS není, na rozdíl od výše popsaných algoritmů BFS a UCS, ani úplný, ani optimální. Časová náročnost algoritmu DFS zůstává exponenciální – teoreticky je dána výrazem $O(b^m)$, kde *b* je faktor větvení a *m* je maximální prohledávaná hloubka stromu, paměťová náročnost je lineární a je dána výrazem O(bm); v případě opakování generování již expandovaných stavů může však hodnota *m* být nekonečná – viz dále.

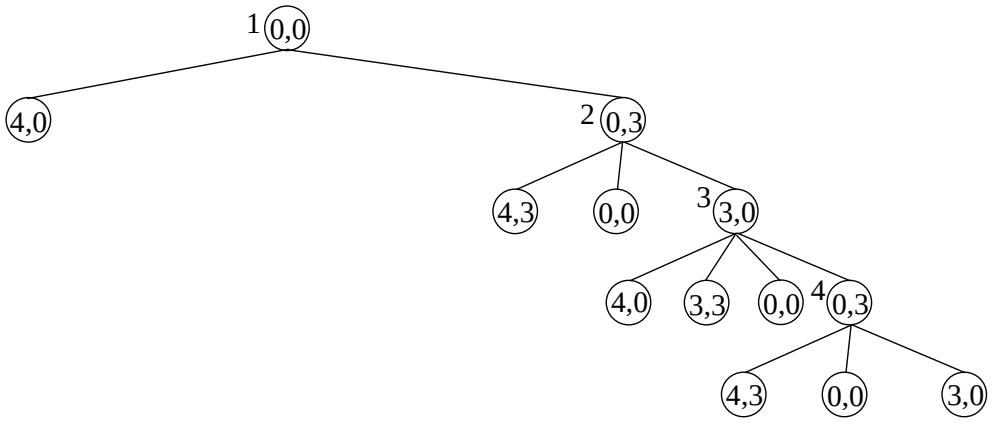
Na Obr. 3.8 je ukázáno pořadí výběru a expanze uzlů ze zásobníku OPEN pro demonstrační prostor z Obr. 3.1.



Pořadí expanze	Zásobník OPEN
0	[[A,-]]
1	[[D,A,-],[C,A,-],[B,A,-]]
2	[[L,D,A,-],[K,D,A,-],[J,D,A,-],[C,A,-],[B,A,-]]
Řešení:	[L,D,A,-]: A ⇒ D ⇒ L

Obr. 3.8 Prohledávání demonstračního stavového prostoru metodou DFS

I když je algoritmus DFS opět velmi jednoduchý, může selhat dokonce i při řešení tak jednoduché úlohy, jakou je výše uvedená úloha dvou džbánů (Obr. 3.9). Na obrázku Obr. 3.9 nejsou pro názornost uvedeny předchůdci uzlů uložených v zásobníku OPEN - je zcela zřejmé, že algoritmus opakovaně generuje uzly (3,0) a (0,3), a řešení proto nikdy nenalezne.



Pořadí expanze	Zásobník OPEN
0	[(0,0)]
1	[(0,3),(4,0)]
2	[(3,0),(0,0),(4,3),(4,0)]
3	[(0,3),(0,0),(3,3),(4,0),(0,0),(4,3),(4,0)]
4	[(3,0),(0,0),(4,3),(0,0),(3,3),(4,0),(0,0),(4,3),(4,0)]
5	…

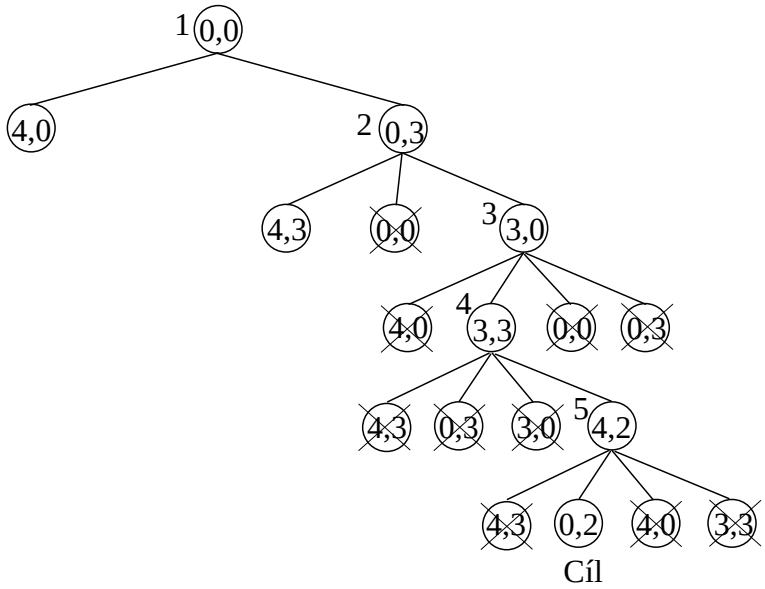
Obr. 3.9 Prohledávání stavového prostoru úlohy dvou džbánů metodou DFS

Metoda DFS v základním provedení je z uvedeného důvodu prakticky nepoužitelná, a je proto nutné prakticky vždy použít její modifikaci – úpravu bodu 5 algoritmu DFS tak, aby byly eliminovány uzly – předchůdci generovaného uzlu a uzly, které jsou již v zásobníku OPEN uložené:

5. Vybraný uzel expanduj a umísti do zásobníku OPEN všechny jeho bezprostřední následníky, kteří v tomto zásobníku ještě nejsou a kteří nejsou ani předky generovaného uzlu.

Na Obr. 3.10 je ukázáno řešení úlohy dvou džbánů metodou DFS s výše uvedenou úpravou bodu 5 algoritmu.

Poznamenejme, že použití seznamu CLOSED pro metodu DFS je sice také možné, ale ztratí se tím výhoda lineární paměťové náročnosti (která pak bude exponenciální), a proto se tato modifikace metody DFS prakticky nepoužívá.

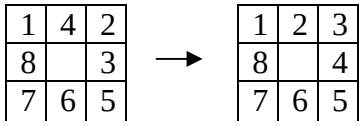


Pořadí expanze	Zásobník OPEN
0	[((0,0),-)]
1	[((0,3),(0,0),-),((4,0),(0,0),-)]
2	[((3,0),(0,3),(0,0),-),((4,3),(0,3),(0,0),-),((4,0),(0,0),-)]
3	[((3,3),(3,0),(0,3),(0,0),-),((4,3),(0,3),(0,0),-),((4,0),(0,0),-)]
4	[((4,2),(3,3),(3,0),(0,3),(0,0),-),((4,3),(0,3),(0,0),-), ((4,0),(0,0),-)]
5	[((0,2),(4,2),(3,3),(3,0),(0,3),(0,0),-),((4,3),(0,3),(0,0),-), ((4,0),(0,0),-)]
6	[((4,3),(0,3),(0,0),-),((4,0),(0,0),-)]
Řešení:	((0,2),(4,2),(3,3),(3,0),(0,3),(0,0),-)) ⇒ cesta = (0,0),(0,3),(3,0),(3,3),(4,2),(0,20

Obr. 3.10 Prohledávání stavového prostoru úlohy dvou džbánů metodou DFS s eliminací předchůdců a uzlů, které jsou již v zásobníku OPEN uložené.

Pozn.: Ověřte si, že záměna prvních dvou operátorů {→M, →V, V→, M→, M→V, V→M} vede k řešení, které není optimální (je v hloubce 7)!

Úkol: Vyřešte pomocí modifikovaného algoritmu DFS úlohu hlavolamu "8" (za operátory zvolte operátory posouvající volné políčko v pořadí nahoru, doprava, dolů a doleva):



Metoda DLS **Metoda omezeného prohledávání do hloubky (DLS)**

Pokud řešíme úlohu, u které dokážeme odhadnout hloubku řešení, můžeme problém s opakujícím se generováním stejných stavů vyřešit omezením hloubky prohledávání (například výše uváděná úloha dvou džbánů má celkem 20 různých stavů a proto řešení musí ležet v nejhorším případě v hloubce 19).

Algoritmus DLS má pak tvar:

1. Sestroj zásobník OPEN a umísti do něj počáteční uzel s označením hloubky (0).
2. Je-li zásobník OPEN prázdný, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Vyber z vrcholu zásobníku OPEN první uzel.
4. Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů, nebo operátorů). Jinak pokračuj.
5. Pokud je hloubka vybraného uzlu menší než zadaná maximální hloubka, tak tento uzel expanduj, všem jeho bezprostředním následníkům přiřaď hloubku o jedničku větší než je hloubka expandovaného uzlu a umísti je do zásobníku OPEN. Jinak pokračuj.
6. Vrať se na bod 2.

Algoritmus DLS není, stejně jako algoritmus DFS, ani úplný, ani optimální. Časová náročnost algoritmu DLS zůstává exponenciální - je dána výrazem $O(b^l)$, kde *b* je faktor větvení a *l* je maximální povolená hloubka prohledávání, paměťová náročnost je lineární a je dána výrazem O(bl).

Úkol: Vyřešte metodou DLS výše definovanou úlohu dvou džbánů, je-li prohledávání stavového prostoru omezeno maximální hloubkou = 5.

Metoda IDS **Metoda postupného zanořování do hloubky (IDS)**

Nelze-li stanovit hloubku řešení a nemáme-li k dispozici dostatek paměti pro použití metody BFS, můžeme se pokusit vyřešit tento problém použitím metody postupného zanořování do hloubky. Princip této metody spočívá v opakovaném použití metody DLS s postupným zvyšováním hloubky prohledávání. Algoritmus je následující:

1. Nastav hloubku prohledávání na hodnotu 1.
2. Použij metodu DLS. Skončí-li tato metoda úspěchem (nalezením cesty), skonči také úspěchem a vrať nalezenou cestu. Jinak pokračuj.
3. Dosáhla-li hloubka prohledávání maximální hodnotu, skonči neúspěchem. Jinak inkrementuj hloubku prohledávání a vrať se na bod 2.

Metoda postupného zanořování do hloubky se zdá být na první pohled velmi neefektivní, protože při každém dalším zanoření opakuje plně prohledávání, které již provedla při předcházející hloubce. Počet expandovaných uzlů pro řešení, které je v hloubce *d*, je zřejmě dán vztahem

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

(kořenový uzel se expanduje *d*-krát, uzly v hloubce 1 se expandují (*d*-1)-krát, až konečně uzly v hloubce *d* se expandují pouze jednou), z něhož je však zřejmé, že složitost metody je stále $O(b^d)$, tzn., že metoda postupného zanořování do hloubky má stejnou složitost, jako metoda BFS. Paměťová složitost metody postupného zanořování do hloubky je dána vztahem O(bd).

Úkol: Vyřešte pomocí metody IDS výše definovanou úlohu dvou džbánů a srovnejte výsledek s výsledkem získaným metodou DLS.

Metoda Backtracking

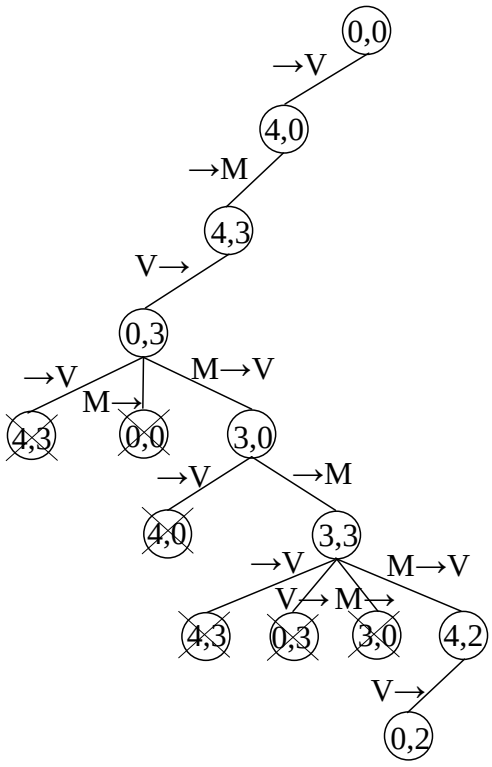
Metoda zpětného navracení (Backtracking)

Metoda zpětného navracení je speciální metodou prohledávání do hloubky, kdy místo expanze vybraného uzlu, tj. generování všech jeho bezprostředních následníků, se generuje pouze následník jeden a teprve v případě neúspěchu se generuje následník další, atd. Pokud uzel není na cestě k řešení (tj. všichni jeho bezprostřední následníci „ohlásili“ neúspěch, vrací uzel neúspěch svému bezprostřednímu předchůdci a popsaná situace s postupnou generací následníků se opakuje v tomto uzlu. Algoritmus je následující:

1. Sestroj zásobník OPEN (bude obsahovat uzly určené k expanzi) a umísti do něj počáteční uzel.
2. Je-li zásobník OPEN prázdný, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Jde-li na uzel na vršku zásobníku aplikovat první/další operátor, tak tento operátor aplikuj a pokračuj bodem 4, v opačném případě odstraň testovaný uzel z vrcholu zásobníku a vrať se na bod 2.
4. Je-li nový vygenerovaný uzel, tj. uzel vzniklý aplikací operátoru na uzel na vršku zásobníku, uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů, nebo operátorů). Jinak ulož nový uzel na vršek zásobníku a vrať se na bod 2.

Metoda má extrémně nízkou paměťovou náročnost, protože v paměti (tj. v zásobníku OPEN) jsou uloženy pouze uzly aktuální cesty spolu označením právě použitých operátorů. Pro časovou náročnost, úplnost a optimálnost platí stejné závěry jako pro metodu DFS: časová náročnost je exponenciální, metoda není ani optimální, ani úplná.

Modifikace metody spočívá v úpravě bodu 4. Nový uzel se do zásobníku OPEN neukládá, pokud se již v tomto zásobníku nachází, tj. pokud nový uzel je již předchůdcem uzlu na vršku zásobníku. Na Obr. 3.11 je ukázáno použití metody zpětného navracení při řešení úlohy dvou džbánů.



Krok	Zásobník OPEN
0	[((0,0),-)]
1	[((4,0),-),((0,0),→V)]
2	[((4,3),-),((4,0),→M),((0,0),→V)]
3	[((0,3),-),((4,3),V→),((4,0),→M),((0,0),→V)]
4	[((3,0),-),((0,3),M→V),((4,3),V→),((4,0),→M),((0,0),→V)]
5	[((3,3),-),((3,0),→M),((0,3),M→V),((4,3),V→),((4,0),→M), ((0,0),→V)]
6	[((4,2)-),((3,3),M→V),((3,0),→M),((0,3),M→V),((4,3),V→), ((4,0),→M),((0,0),→V)]
7	[((0,2),-),((4,2),V→),((3,3),M→V),((3,0),→M),((0,3),M→V), ((4,3),V→),((4,0),→M),((0,0),→V)] = řešení

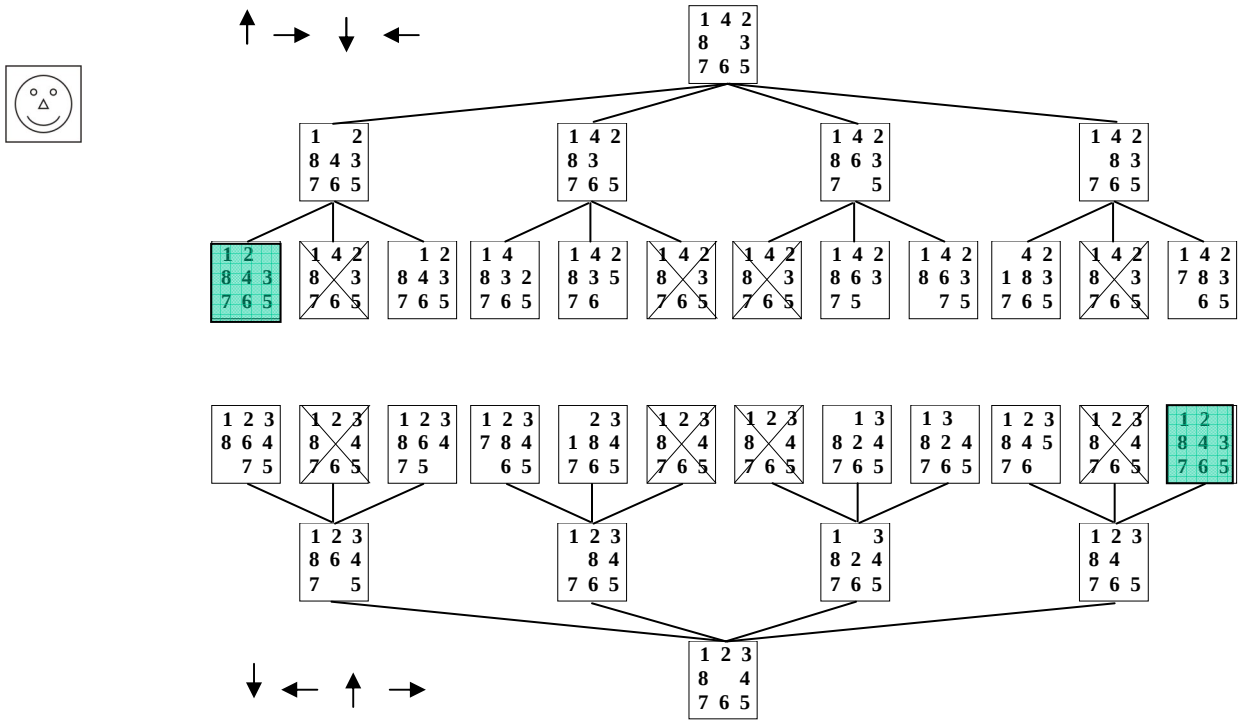
Obr. 3.11 Prohledávání stavového prostoru úlohy dvou džbánů metodou zpětného navracení s eliminací uzlů, které jsou již v zásobníku OPEN uložené.

Metoda BS

Metoda obousměrného prohledávání (BS)

U úloh, které používají inverzní operátory je možné prohledávat stavový prostor oběma směry metodou BFS – od počátečního k cílovému stavu a současně od cílového stavu ke stavu počátečnímu. Pokud se pak v každém směru prohledá poloviční hloubka, klesne časová i paměťová složitost generování uzlů z $O(b^d)$ na $O(2b^{d/2}) \sim O(b^{d/2})$. Pro časovou složitost porovnávání však toto neplatí, protože v každém kroku je nutné porovnávat všechny aktuální koncové stavy jednoho prohledávání se všemi aktuálními koncovými stavy druhého prohledávání – časová složitost proto opět zůstává $O(b^{d/2}\, b^{d/2}) = O(b^d)$.

Příklad použití metody obousměrného prohledávání je ukázán na Obr. 3.12. Zeleně jsou označeny stejné stavy v koncových uzlech obou prohledávání – ty tvoří hledaný „most“ přes který je následně možné zrekonstruovat výsledné řešení.



Obr. 3.12 Prohledávání stavového prostoru hlavolamu “8” metodou obousměrného prohledávání

Z Obr. 3.12 je zřejmé, že výsledné řešení je dáno postupnou aplikací operátorů ↑ (prázdné políčko nahoru), → (prázdné políčko doprava), ↓ (prázdné políčko dolů), ← (prázdné políčko doleva).

3.2.6 Informované metody

Informované metody

Informované metody mají k dispozici a používají nějakou informaci o cílovém stavu a mají prostředky, jak aktuální stavy prohledávání hodnotit. Právě podobné metody převážně používá i člověk – vrátíme-li se k příkladu prohledávání mapy, pak hledáme-li cestu z nějakého (výchozího) místa do jiného (cílového) místa máme obvykle alespoň přibližnou představu, ve kterém směru od výchozího místa cílové místo leží. Je zřejmé, že čím je naše představa o poloze cílového místa přesnější, tím menší oblast mapy musíme prohledávat.

Základními informovanými metodami jsou metody patřící do skupiny metod Best First Search.

Metody BestFS

Metody Best First Search

Metody založené na výběru nejlépe ohodnoceného stavu (Best First Search) jsou velmi podobné neinformované metodě UCS. Algoritmus je prakticky stejný:

1. Sestroj seznam OPEN (bude obsahovat všechny uzly určené k expanzi) a umísti do něj počáteční uzel včetně jeho ohodnocení.
2. Je-li seznam OPEN prázdný, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.
3. Vyber ze seznamu OPEN uzel s nejlepším (nejnižším) ohodnocením.
4. Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů, nebo operátorů). Jinak pokračuj.
5. Vybraný uzel expanduj, všechny jeho bezprostřední následníky, kteří nejsou jeho předky, umísti do seznamu OPEN, a to včetně jejich ohodnocení. Z uzlů, které se v seznamu OPEN vyskytují vícekrát, ponech pouze uzel s nejlepším ohodnocením, ostatní ze seznamu OPEN vyškrtni, a vrať se na bod 2.

Zásadní rozdíl spočívá pouze v ohodnocující funkci. V metodě UCS byla hodnotou této funkce v uzlu *n* cena cesty (součet cen přechodů) z počátečního uzlu do uzlu *n*. V informovaných metodách musí cena cesty obsahovat i odhad ceny z uzlu *n* do uzlu cílového. Obecně pak hodnota ohodnocující funkce *f(n)* uzlu *n* je dána součtem dvou složek (Obr. 3.13): ceny cesty *g(n)* z počátečního stavu do uzlu *n* a odhadu ceny cesty z uzlu *n* do uzlu cílového *h(n)*:

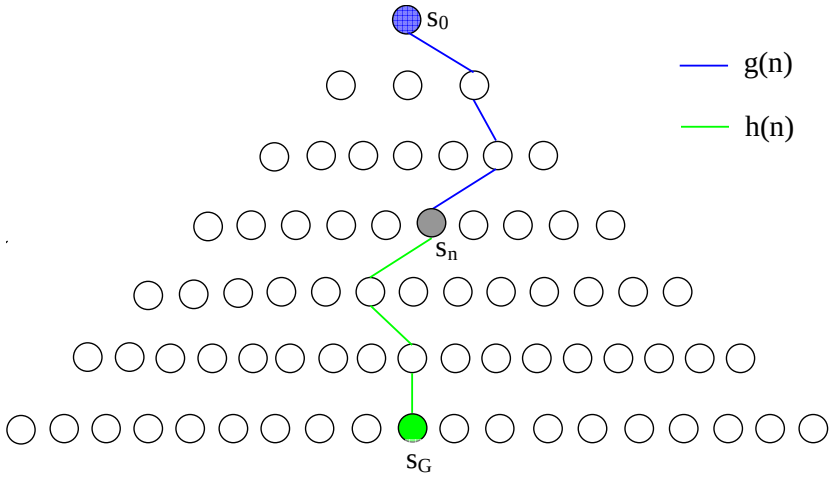
$$\mathbf{f(n) = g(n) + h(n)}$$

Je zřejmé, že cena cesty v počátečním uzlu je nulová ($g(0) = 0$), a že nulová musí být i hodnota heuristické funkce v cílovém uzlu ($h(G) = 0$).

Zdůrazněme, že ceny všech přechodů musí být kladné - musí platit vztah $g(s_{n+1}) > g(s_n)$.

Funkce *h(n)* se nazývá heuristickou funkcí, nebo krátce heuristikou. Čím přesnější je tato funkce, tím méně stavového prostoru je nutné prohledávat. V extrémním případě, pokud je heuristika přesným odhadem, tj. pokud vlastně známe řešení, tak samozřejmě nemusíme prohledávat žádný stavový prostor.

Metodu UCS lze považovat za extrémní metodu, pro kterou je heuristická složka pro všechny uzly nulová ($h(n) = 0$). Druhou extrémní metodou je metoda Greedy search, pro kterou je naopak pro všechny uzly nulová cena cesty ($g(n) = 0$).



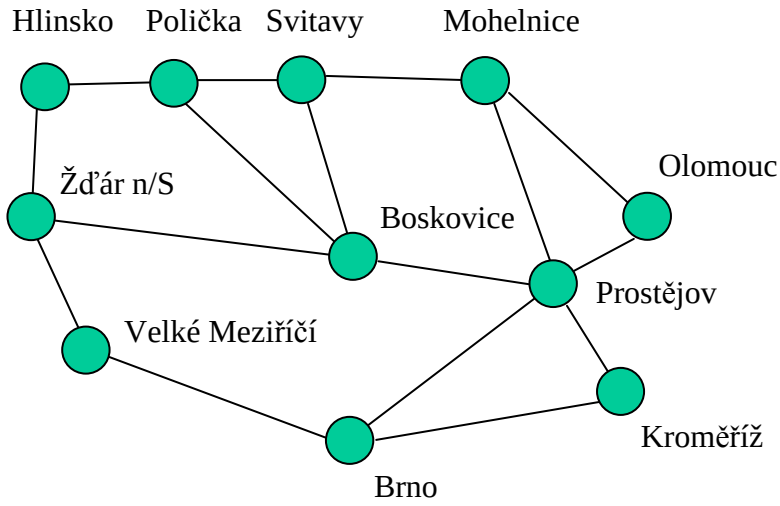
Obr. 3.13 Příklad ohodnocení cesty

Metoda Greedy search

Metoda Greedy search

Metoda Greedy search (volně lze přeložit jako metoda lačného prohledávání) ohodnocuje uzly pouze heuristickou funkcí, tj. odhadovanou cenou z daného uzlu do uzlu cílového - k expanzi vybírá uzel, který má toto hodnocení nejnižší. Z hlediska úplnosti, optimálnosti a časových a prostorových náročností platí pro metodu Greedy search stejné závěry jako pro metodu DFS. Dobrá heuristika nicméně může časovou náročnost výrazně redukovat!

Princip metody Greedy search ukážeme na úlohách hledání cesty (Obr. 3.14).



Obr. 3.14 Příklad úlohy hledání cesty

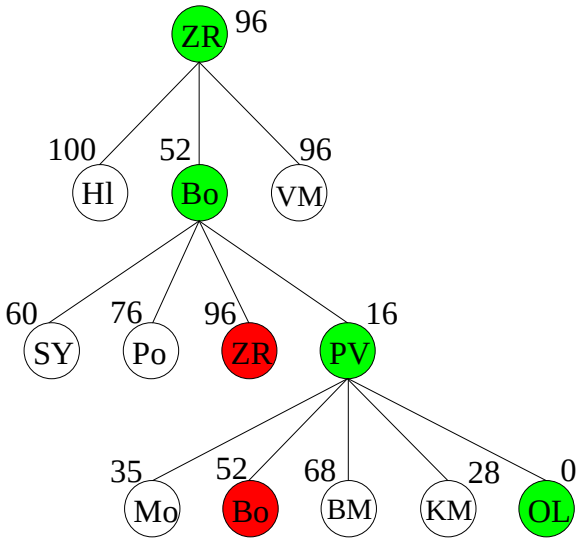
Uvažujme dvě úlohy:

1. ze Žďáru nad Sázavou do Olomouce
2. ze Žďáru nad Sázavou do Brna

Nejpoužívanější heuristikou pro podobné úlohy je přímá vzdálenost mezi místy, a to i v případech, kdy mezi nimi není přímé spojení.

Přímá vzdálenost [km] výchozí místo:	do OL	do BM
Žďár nad Sázavou	96	64
Hlinsko	100	80
Velké Meziříčí	96	48
Polička	76	62
Boskovice	52	32
Brno	68	0
Svitavy	60	65
Mohelnice	35	73
Prostějov	16	47
Kroměříž	28	58
Olomouc	0	68

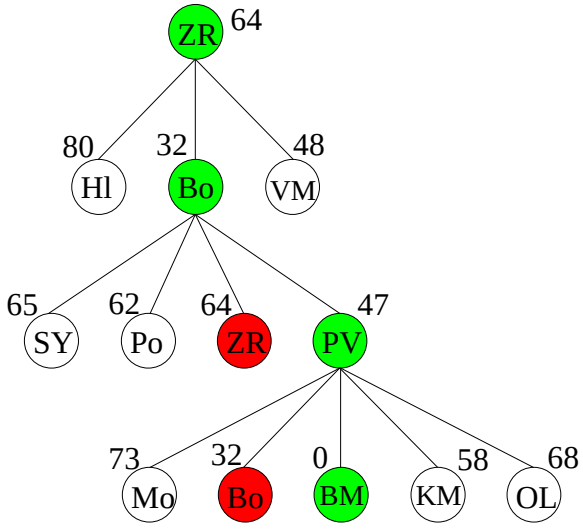
Úloha 1. (Žďár n/S – Olomouc):



Krok	Seznam OPEN
0	[((ZR,-),96)]
1	[((Hl,ZR,-l),100),((Bo,ZR,-),52),((VM,ZR,-),96)]
2	[((Hl,ZR,-),100),((VM,ZR,-),96),((SY,Bo,ZR,-),60), ((Po,Bo,ZR,-),76),((ZR,Bo,ZR,-),96),((PV,Bo,ZR,-),16)]
3	[((Hl,ZR,-),100),((VM,ZR,-),96),((SY,Bo,ZR,-),60), ((Po,Bo,ZR,-),76), ((Mo,PV,Bo,ZR,-),35),((Bo,PV,Bo,ZR,-),52), ((BM,PV,Bo,ZR,-),68),((KM,PV,Bo,ZR,-),28), ((OL,PV,Bo,ZR,-),0)]
4	((OL,PV,Bo,ZR,-),0) ⇒ Cíl

Obr. 3.15 Prohledávání stavového prostoru metodou Greedy search – úloha 1

Úloha 2. (Žďár n/S – Brno):



Krok	Seznam OPEN
0	[((ZR,-),64)]
1	[((Hl,ZR,-),80),((Bo,ZR,-),32),((VM,ZR,-),48)]
2	[((Hl,ZR,-),80),((VM,ZR,-),48),((SY,Bo,ZR,-),65), ((Po,Bo,ZR,-),62),((ZR,Bo,ZR,-),64),((PV,Bo,ZR,-),47)]
3	[((Hl,ZR,-),80),((VM,ZR,-),48),((SY,Bo,ZR,-),65), ((Po,Bo,ZR,-),62),((Mo,PV,Bo,ZR,-),73),((Bo,PV,Bo,ZR,-),32), ((BM,PV,Bo,ZR,-),0),((KM,PV,Bo,ZR,-),58), ((OL,PV,Bo,ZR,-),68)]
4	((BM,PV,Bo,ZR,-),0) ⇒ Cíl

Obr. 3.16 Prohledávání stavového prostoru metodou Greedy search – úloha 2

Metoda Greedy search nalezla optimální řešení první úlohy, zatímco nalezené řešení druhé úlohy optimální není (nalezená cesta není nejkratší).

Metoda A*

Metoda A*

Metoda A* (čti A s hvězdičkou) je nejznámější a nejpoužívanější metodou pro řešení úloh prohledáváním stavového prostoru. Jde opět o metodu Best First Search, ve které heuristická funkce $h(n)$ musí být tzv. spodním odhadem skutečné ceny cesty od ohodnocovaného uzlu k cíli (t.j. odhad musí být menší, než skutečná cena) – taková heuristika se pak nazývá přípustnou heuristikou, resp. přípustnou heuristickou funkcí.

Metoda A* je úplná a optimální. Důkaz optimálnosti je poměrně jednoduchý:

1. Označme cenu optimální cesty do optimálního cílového uzlu jako $f_o(s_{Gopt})$ a cenu jiné (neoptimální) cesty do stejného nebo jiného cílového uzlu jako $f(s_G)$. Zřejmě musí platit $f_o(s_{Gopt}) < f(s_G)$.
2. Nechť x je uzel na optimální cestě k optimálnímu cíli. Pro přípustnou heuristiku musí platit $f_o(s_{Gopt}) \geq f(x)$.
3. Pokud by uzel x nebyl vybrán k expanzi, zatímco cílový uzel s_G s ohodnocením $f(s_G)$ ano, muselo by platit $f(x) \geq f(s_G)$.
4. Z bodů 2 a 3 vyplývá, že $f_o(s_{Gopt}) \geq f(x) \geq f(s_G)$, tedy že $f_o(s_{Gopt}) \geq f(s_G)$, což je ve sporu se závěrem bodu 1.
5. Uzel x proto musí být vybrán k expanzi, stejně jako každý jiný uzel na optimální cestě k optimálnímu cíli a metoda proto musí nalézt optimální cestu.

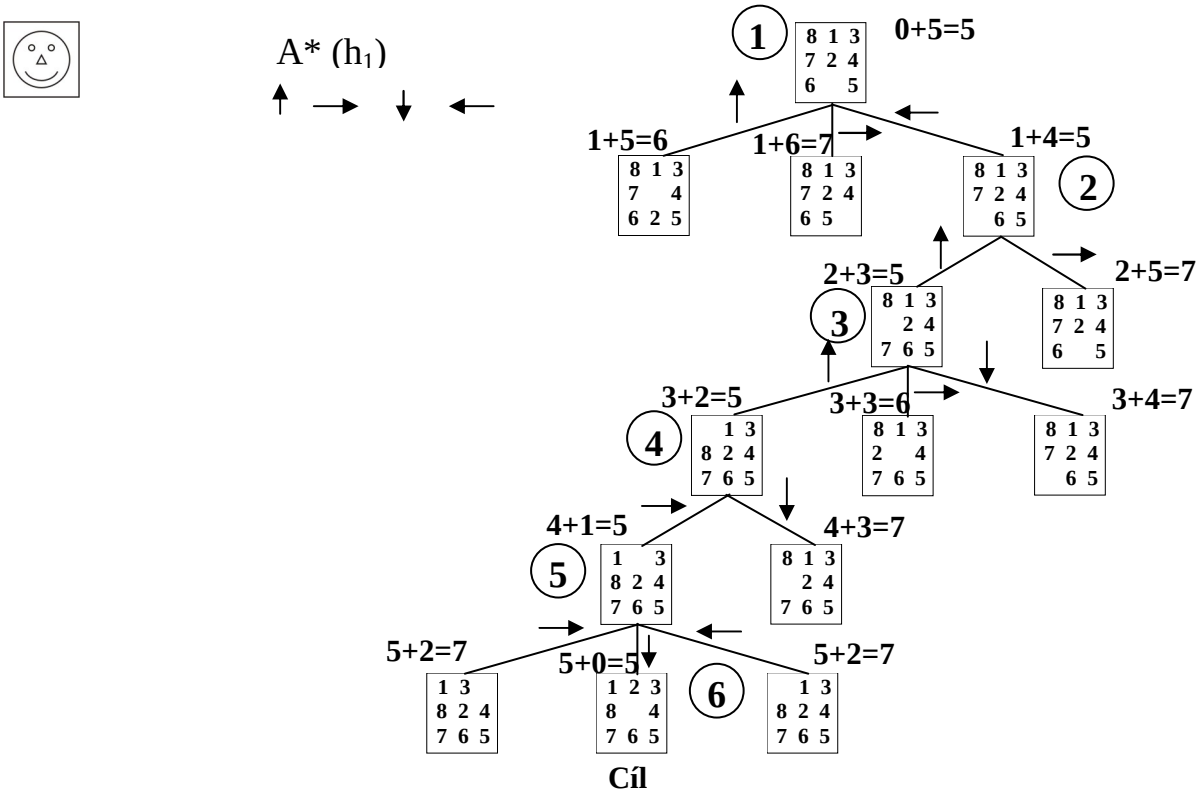
Algoritmus A* expanduje pouze uzly x pro které platí $f(s_x) \leq f_o$, kde f_o je cena optimální cesty. Časovou a prostorovou náročnost proto výrazně ovlivňuje použitá heuristika – pokud se blíží 0 (to je jistě spodní odhad skutečné ceny), pak složitost se blíží exponenciální složitosti, pokud je použitá heuristika dobrým spodním odhadem skutečné ceny, pak jsou expandovány pouze uzly kolem optimální cesty (je-li přesným odhadem, pak jsou expandovány pouze uzly na optimální cestě).

Pozn.: Doba výpočtu hodnot heuristické funkce se významně promítá do celkové doby výpočtu (závisí přirozeně na její složitosti). Často je proto výhodnější používat jednodušší funkce i za cenu většího počtu expandovaných uzlů.

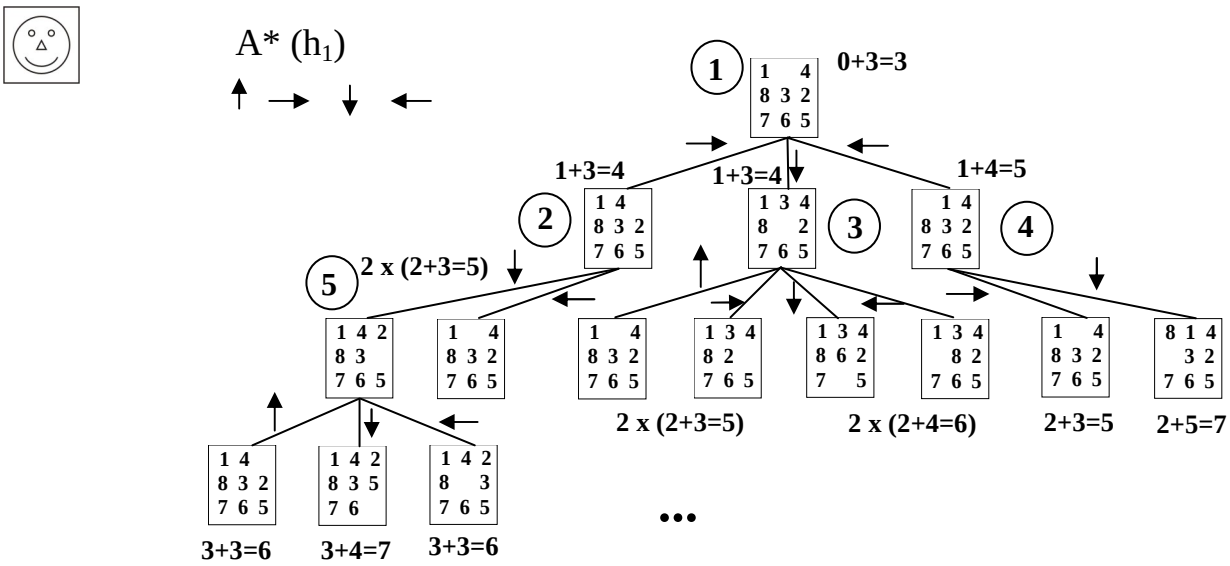
Pro úlohu hlavolamu “8” lze například použít následující dvě heuristiky, které

jsou jistě spodním odhadem:

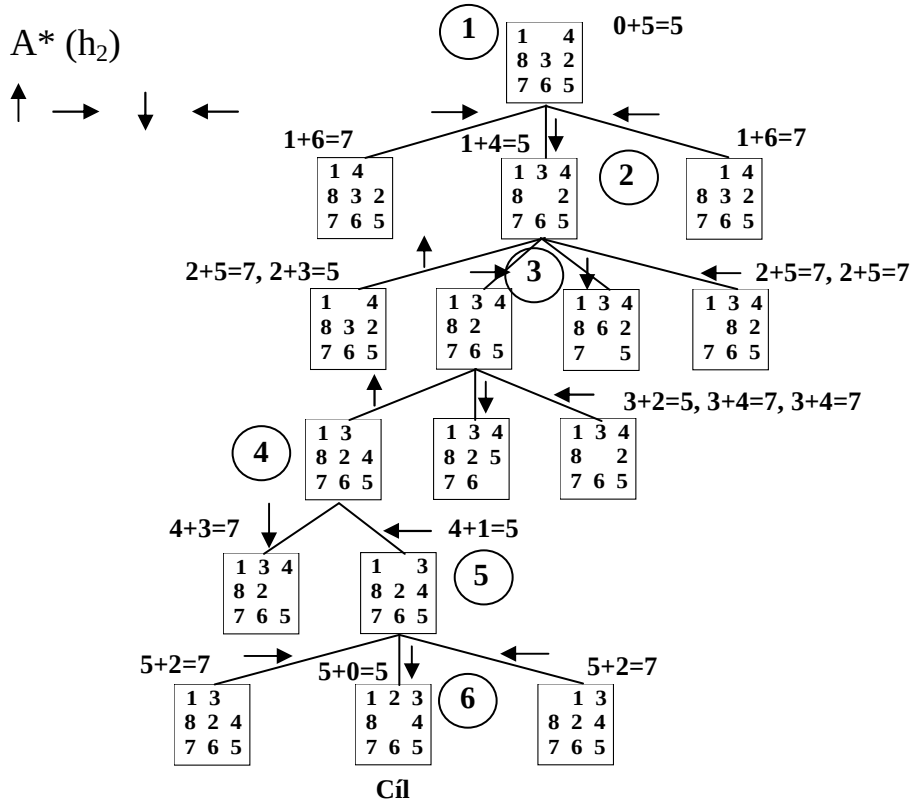
- počet kamenů na nesprávných pozicích (heuristiku označme jako h_1)
- součet vzdáleností kamenů (např. vzdálenost Manhattan) od jejich cílových pozic (heuristiku označme jako h_2)



Obr. 3.17 Prohledávání stavového prostoru hlavolamu "8" metodou A* s heuristikou h_1



Obr. 3.18 Prohledávání stavového prostoru hlavolamu "8" metodou A* s heuristikou h_1 (jiný počáteční stav než v Obr. 3.17)



Obr. 3.19 Prohledávání stavového prostoru hlavolamu “8” metodou A* s heuristikou h_2 (stejný počáteční stav jako v Obr. 3.18)

Na Obr. 3.17 je ukázáno řešení úlohy hlavolamu “8”, pro které je heuristika h_1 plně dostatečná – expanduje pouze pět uzlů a nalezne optimálních pět tahů. Na Obr. 3.18 je použita stejná heuristika pro řešení stejného hlavolamu, pouze s jiným počátečním stavem. Zde již tato heuristika není tak úspěšná – i když by algoritmus nalezl opět optimální řešení, potřeboval by k tomu expandovat více uzlů, než je nezbytně potřebné. Heuristika h_2 je lepším spodním odhadem skutečné ceny a proto i v tomto případě nalezne optimální řešení expanzí pouze nezbytně nutného počtu uzlů.

Závěr: Platí-li podmínka $h \geq h_2 > h_1$, kde h je skutečná cena, pak h_2 je lepší heuristika než h_1!

Úkol: Zvolenými metodami řešte úlohy:

1. "misionáři a kanibalové" (3 kanibalové a 3 misionáři se mají přepravit pomocí dvoumístné loďky z jednoho břehu řeky na druhý, ze zřejmých důvodů však nikdy nesmí být na žádném břehu více kanibalů než misionářů
2. úloha "vlk, koza, zelí" (problém podobný předchozímu: převozník má převézt z jednoho břehu řeky na druhý po jednom vlka, kozu a zelí, přičemž opět ze zřejmých důvodů nesmí nechat na žádném břehu bez dozoru vlka s kozou, nebo kozu ze zelím)

3.2.7 Metody lokálního prohledávání

Metody lokálního prohledávání

Existuje řada úloh, jejichž řešením je pouze nalezení cílového stavu a vlastní cesta je přitom bezvýznamná (rozmístění součástek na desce s plošnými spoji, rozmístění prvků v integrovaných obvodech, optimální rozložení zboží v regálech obchodů, optimalizace telekomunikačních sítí apod.). K řešení takových úloh se používají metody, které místo optimální cesty hledají optimální stav – tzv. metody lokálního prohledávání. Metody lokálního prohledávání neprohledávají a ani nemohou prohledávat stavový prostor systematicky, přesto mají dvě přednosti:

1. Mají zcela zanedbatelnou paměťovou náročnost.
2. Často dospějí k přijatelnému řešení v rozsáhlých (dokonce i ve spojitých, resp. nekonečných) stavových prostorech, kdy použití výše popsaných systematických algoritmů je nemožné.

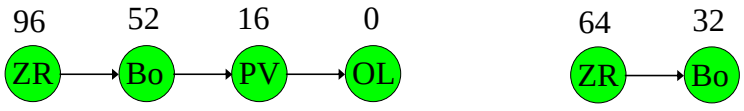
Metoda Hill-climbing

Metoda Hill-climbing (volně přeloženo metoda lezení do kopce) používá k ohodnocení uzlu *n*, podobně jako metoda Greedy search, pouze heuristiku $h(n)$. Obvykle je však touto heuristikou funkce, která ohodnocuje „kvalitu řešení“ spíše než vzdálenost od cílového řešení – pak čím lepší řešení ohodnocovaný uzel představuje, tím vyšší je mu přiřazena hodnota a algoritmus vybírá k expanzi uzel s nejvyšším ohodnocením (proto i název hill-climbing).

Vlastní algoritmus je zcela jednoduchý:

1. Vytvoř uzel *Current* totožný s počátečním uzlem s_0 (včetně jeho ohodnocení).
2. Expanduj uzel *Current*, ohodnoť jeho bezprostřední následníky a vyber z nich nejlépe ohodnoceného (nazvěme jej *Next*).
3. Je-li ohodnocení uzlu *Current* lepší než ohodnocení uzlu *Next*, ukonči řešení a vrať jako výsledek uzel *Current*. Jinak pokračuj.
4. Ulož uzel *Next* do uzlu *Current* a vrať se na bod 2.

Na následujícím obrázku (Obr. 3.20) jsou ukázána řešení získaná metodou Hill-climbing pro obě úlohy, řešené dříve metodou Greedy search. Zatímco první úlohu vyřeší metoda Hill-climbing velmi rychle a bez jakýchkoliv problémů, druhou úlohu nevyřeší – zůstane uvízlá v lokálním extrému, který představuje město Boskovice, z něhož je přímou čarou do Brna nejblíže, ale z něhož cesta do Brna (podle mapy z Obr. 3.14) nevede.



a) ze Žďáru n/S do Olomouce b) ze Žďáru n/S do Brna

Obr. 3.20 Lokální prohledávání stavového prostoru úlohy hledání cesty z Obr. 3.14 metodou Hill-climbing

Lokální extrémy jsou hlavním problémem metody Hill-climbing. Snaha o jejich překonání vedla ke vzniku několika modifikací, z nichž nejznámější je metoda Beam search, která v každém kroku ukládá do seznamu uzlů CURRENTS

k nejlépe ohodnocených bezprostředních následníků všech *k* uzlů-předchůdců ze seznamu CURRENTS.

Metoda simulovaného žíhání

Metoda simulovaného žíhání

Metoda simulovaného žíhání je stochastickou metodou, jejímž cílem je překonání lokálních extrémů vedoucích k častým neúspěchům metody Hill-climbing. Vychází z principů žíhání kovů a používá i stejnou terminologii. Algoritmus je následující:

1. Vytvoř tabulku s předpisem pro klesání teploty v závislosti na kroku výpočtu.
2. Vytvoř uzel *Current* totožný s počátečním uzlem s_0 (včetně jeho ohodnocení). Nastav krok výpočtu na nulu ($k = 0$).
3. Z tabulky zjisti aktuální teplotu T (T = f(k)). Je-li tato teplota nulová (T = 0) ukonči řešení a vrať jako výsledek uzel *Current*.
4. Expanduj uzel *Current* a z jeho bezprostředních následníků vyber náhodně jednoho z nich (nazvěme jej *Next*).
5. Vypočítej rozdíl ohodnocení uzlů *Current* a *Next*:
   ΔE = value(*Next*) – value(*Current*).
6. Jestliže $\Delta E > 0$, tak ulož uzel *Next* do uzlu *Current*, jinak ulož uzel *Next* do uzlu *Current* s pravděpodobností $e^{\Delta E/T}$.
7. Inkrementuj krok výpočtu *k* a vrať se na bod 3.

Z bodu 6) je zřejmé, že pro vysokou počáteční teplotu je pravděpodobnost „přijetí“ náhodně vybraného následníka s horším ohodnocením, než je ohodnocení uzlu *Current*, vysoká. Se snižující teplotou se snižuje, až konečně pro teplotu blízkou nule jde prakticky o algoritmus Hill-climbing (akceptuje se pouze následník s ohodnocením lepším, než je ohodnocení uzlu *Current*).

Metody založené na genetických algoritmech

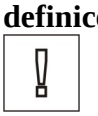
Metody založené na genetických algoritmech

Popis metod založených na genetických algoritmech přesahuje rámec předmětu IZU. Zájemce odkazujeme na volitelný předmět Softcomputing v magisterském studijním programu.

3.3 Metody řešení úloh s omezujícími podmínkami

U úloh, které byly řešeny metodami popsanými v kap. 3.2 , stačilo testovat, zda vybraný stav není stavem cílovým, a v opačném případě pak testovat použitelnost operátorů na tento stav. Úlohy s omezujícími podmínkami se od předcházejících úloh liší tím, že je nutné zkoumat i vnitřní strukturu jejich stavů.

3.3.1 Úvodní definice



Formálně je úloha s omezujícími podmínkami (CSP – *Constraint Satisfaction Problem*) definována takto:

- Nechť existuje množina proměnných $\{X_1, X_2, \ldots, X_n\}$, z nichž každá proměnná X_i může nabývat hodnot z neprázdné domény D_i.
- Nechť existuje množina omezujících podmínek $\{C_1, C_2, \ldots, C_m\}$, z nichž každá podmínka předepisuje vztah mezi nějakou podmnožinou proměnných z výše zmíněné množiny proměnných.
- Nechť stav úlohy je definován hodnotami přiřazenými jednotlivým

proměnným a nechť legální stav znamená, že proměnné s přiřazenými hodnotami vyhovují předepsaným podmínkám.
- V počátečním stavu není přiřazena hodnota žádné proměnné.
- Obecný operátor přiřazuje hodnotu libovolné volné proměnné tak, aby následný stav byl legálním stavem.
- Řešením úlohy je legální stav, ve kterém všechny proměnné mají přiřazené hodnoty.

Poznamenejme, že některé CSP vyžadují takové řešení (legální stav), ve kterém nabývá nějaká ohodnocující funkce extrémní hodnoty.

3.3.2 Typy úloh

Typy úloh - úlohy, které je možné řešit jako CSP

- Úloha N dam (*N Queens problem*) – byla již popsána v kap 3.2.
- Úlohy kryptoaritmetických hlavolamů – úkolem je přiřadit každému použitému písmenu jinou číslici tak, aby výsledná přiřazení vyhovovala předepsaným součtům, např.:

a)
```
  SEND
+ MORE
 MONEY
```

b)
```
 FORTY
+  TEN
+  TEN
 SIXTY
```

K praktickým úlohám, které jsou řešitelné jako CSP, patří například úlohy barvení map a rozmísťování prvků VLSI obvodů.

3.3.3 Formulace úloh

Formulace úloh

Řádná formulace řešené úlohy je opět velmi důležitá a opět na ní často závisí i úspěch či neúspěch vlastního řešení. Jako názorné příklady uvedeme formulace problému pro výše uvedené úlohy.

Úloha osmi dam:

- Množina proměnných $\{Q_1,Q_2,\ldots,Q_8\}$ označuje dámy jednotlivých sloupců.
- Domény D_i jsou všechny stejné a obsahují čísla řádků {1,2,…,8}.
- Omezující podmínky:
  1. Dámy nesmí být ve stejných sloupcích - zajišťuje definice proměnných.
  2. Dámy nesmí být ve stejných řádcích: Qi # Qj pro i # j.
  3. Dámy nesmí být ve stejných úhlopříčkách: |Qi – Qj| # |i – j| pro i # j.

Kryptoaritmetický hlavolam, ad a):

- Množina proměnných {S,E,N,D,M,O,R,Y} označuje jednotlivá písmena.
- Domény D_i jsou stejné a obsahují desítkové číslice {0,1,…,9}.
- Omezující podmínky:
  1. D + E = Y + 10·P1
  2. N + R + P1 = E + 10·P2
  3. E + O + P2 = N + 10·P3
  4. S + M + P3 = O + 10·P4
  5. M = P4
  6. M # 0

3.3.4 Přehled metod

Přehled metod řešení CSP

CSP lze řešit všemi metodami prohledávání stavového prostoru, které byly popsány v předchozí kapitole. Existují však speciální a výrazně efektivnější metody, z nichž si dále ukážeme principy tří nejznámějších (s původními anglickými názvy):

- Backtracking for CSP
- Forward checking
- Min-conflict

Metoda Backtracking for CSP

Metoda Backtracking for CSP

Princip metody (algoritmus) je velmi jednoduchý: v každém kroku přiřazuje hodnotu neoznačené proměnné a pokud nový stav není legálním stavem, přiřazuje další hodnotu atd. Není-li možné této proměnné přiřadit žádnou hodnotu, pak se okamžitě vrací o krok zpět (Obr. 3.21, úloha čtyř dam).

Metoda Forward checking

Metoda Forward checking

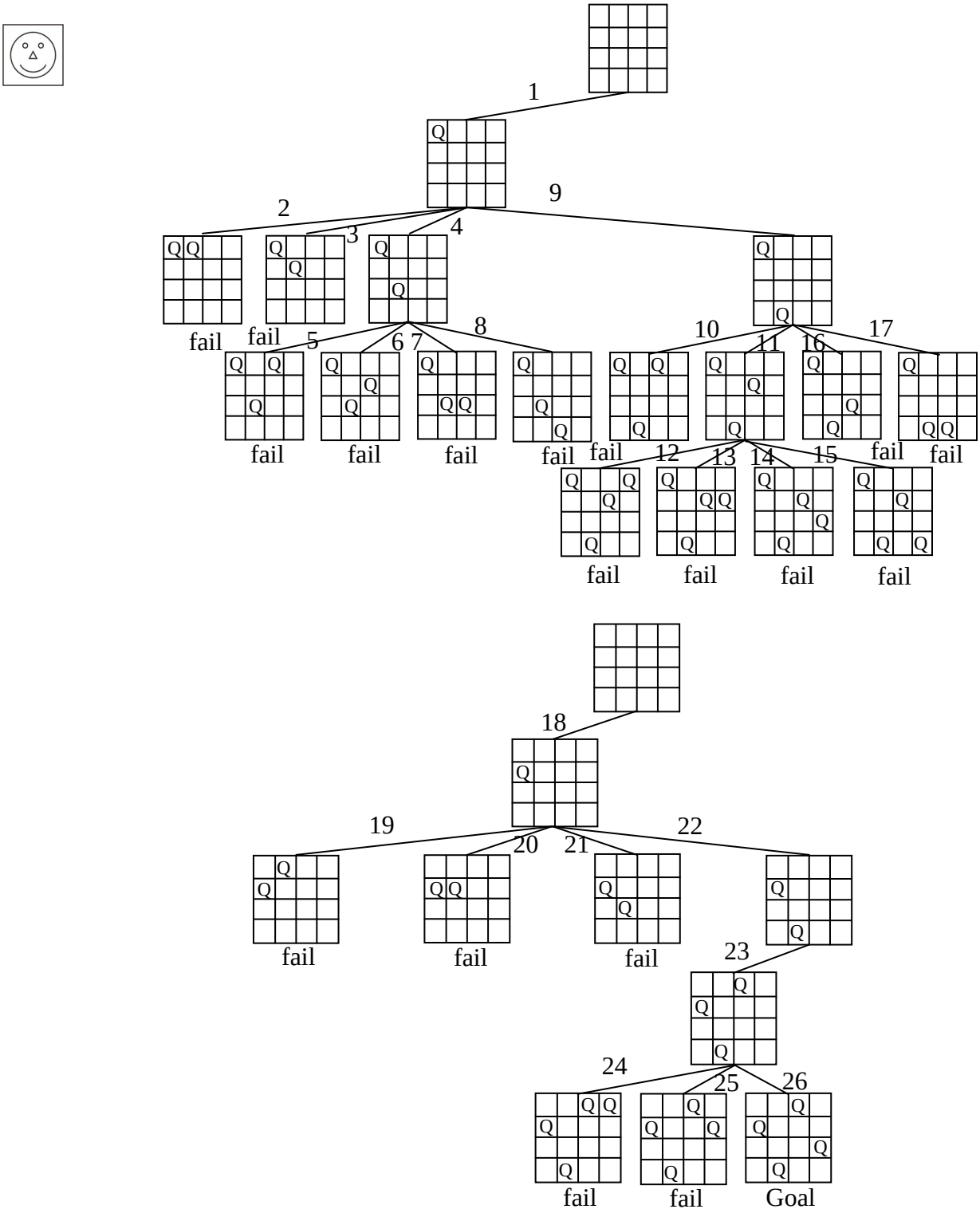
Princip algoritmu Forward checking spočívá v tom, že po každém přiřazení hodnoty nějaké proměnné vyřazuje z množin přípustných hodnot dosud volných proměnných ty hodnoty, které jsou s právě přiřazenou hodnotou v konfliktu. Algoritmus je dán procedurou, která rekurzivně volá sama sebe:

- Přiřaď každé proměnné i (i = 1,…,n) množinu přípustných hodnot S_i.
- Volej proceduru Forward_Checking pro první proměnnou (i = 1).

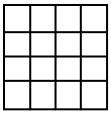
procedura Forward_Checking:

1. Odstraň první hodnotu z množiny S_i a přiřaď tuto hodnotu proměnné *i*, nechť *X* je nový stav.
2. Je-li *X* cílovým, tj. legálním stavem, skonči proceduru úspěchem (vrať *X*), jinak pokračuj.
3. Odstraň z množin S_j, (j = i + 1, ..., n) všechny hodnoty, které jsou v konfliktu s dosud přiřazenými hodnotami.
4. Jestliže je nějaká množina S_j (j = i + 1, ..., n) prázdná, obnov původní stav množin Sj (tj. stav před bodem 3) a jdi na bod 7. Jinak pokračuj.
5. Volej proceduru Forward_Checking pro proměnnou k = i + 1.
6. Skončí-li procedura Forward_Checking pro proměnnou k úspěchem, skonči také úspěchem (vrať vrácený stav). Jinak pokračuj.
7. Není-li množina S_i prázdná, vrať se na bod 1. Jinak skonči neúspěchem.

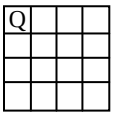
Příklad použití algoritmu Forward checking je ukázáno na obrázku Obr. 3.22. Z obrázku je zřejmé vyřazování hodnot, které jsou v konfliktu. Například po postavení 1. dámy na 1. políčko, jsou z přípustných hodnot pro 2. dámu vyřazeny hodnoty 1 a 2, z přípustných hodnot pro 3. dámu hodnoty 1 a 3, a z přípustných hodnot pro 4. dámu hodnoty 1 a 4. Dále je vidět mechanizmus navracení, například po postavení druhé dámy na třetí políčko, kdy seznam přípustných hodnot pro 3. dámu je prázdný – druhá dáma se z třetího políčka odstraní, hodnota 3 se vypustí ze seznamu přípustných hodnot pro 2. dámu a seznamy přípustných hodnot pro 3. a 4. dámu se obnoví na stav po postavení 1. dámy.



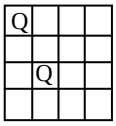
Obr. 3.21 Řešení úlohy čtyř dam metodou Backtracking for CSP (**fail** značí neúspěch, **Goal** znamená cíl)



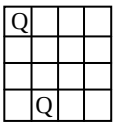
$S_1 = \{1,2,3,4\}, S_2 = \{1,2,3,4\}, S_3 = \{1,2,3,4\}, S_4 = \{1,2,3,4\}$



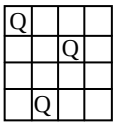
$i_1 = 1,$
$S_1 = \{2,3,4\}, S_2 = \{3,4\}, S_3 = \{2,4\}, S_4 = \{2,3\}$



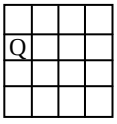
$i_1 = 1, i_2 = 3,$
$S_1 = \{2,3,4\}, S_2 = \{4\}, S_3 = \{\}, S_4 = \{2\}$ **fail !**
$S_3 = \{2,4\}, S_4 = \{2,3\}$



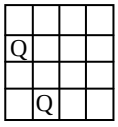
$i_1 = 1, i_2 = 4,$
$S_1 = \{2,3,4\}, S_2 = \{\}, S_3 = \{2\}, S_4 = \{3\}$



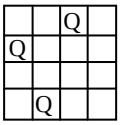
$i_1 = 1, i_2 = 4, i_3 = 2,$
$S_1 = \{2,3,4\}, S_2 = \{\}, S_3 = \{\}, S_4 = \{\}$ **fail !**
$S_2 = \{1,2,3,4\}, S_3 = \{1,2,3,4\}, S_4 = \{1,2,3,4\}$



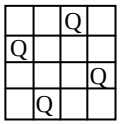
$i_1 = 2,$
$S_1 = \{3,4\}, S_2 = \{4\}, S_3 = \{1,3\}, S_4 = \{1,3,4\}$



i1 = 2, i2 = 4,
S1 = {3,4}, S2 = {}, S3 = {1}, S4 = {1,3}



i1 = 2, i2 = 4, i3 = 1,
S1 = {3,4}, S2 = {}, S3 = {}, S4 = {3}



i1 = 2, i2 = 4, i3 = 1, i4 = 3,
S1 = {3,4}, S2 = {}, S3 = {}, S4 = {} **Goal !**

Obr. 3.22 Řešení úlohy čtyř dam metodou Forward checking (**fail** značí neúspěch, **Goal** znamená cíl)

Metoda Min-conflict

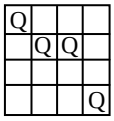
Metoda Min-conflict

Algoritmus Min-conflict je lokálním algoritmem, který vychází z libovolného úplného (všem proměnným jsou přiřazeny nějaké hodnoty), ale nelegálního (přiřazené hodnoty nesplňují podmínky) stavu a snaží se zmenšovat počet konfliktů (tj. počet porušených podmínek). Tento velmi jednoduchý přístup je překvapivě efektivní pro mnoho CSP.

Na Obr. 33 je ukázáno použití algoritmu Min-conflict opět na řešení problému čtyř dam.

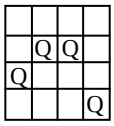
Úkol: Metodami Backtracking for CSP, Forward checking a Min-conflict řešte kryptoaritmetické úlohy uvedené na začátku této kapitoly.

1 2 3 4

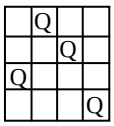


Počet ohrožení dámy *i* na jednotlivých řádcích sloupce *i*:

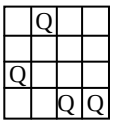
$Q_1 = \{2,2,1,2\}$



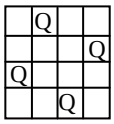
$Q_2 = \{1,3,2,2\}$



$Q_3 = \{2,1,2,1\}$

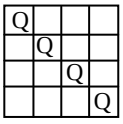


$Q_4 = \{1,0,3,1\}$

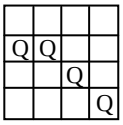


$Q_1 = \{1,3,0,1\}$, $Q_2 = \{0,2,2,3\}$, $Q_3 = \{3,2,2,0\}$, $Q_4 = \{1,0,3,1\}$
$\Rightarrow$ Cíl

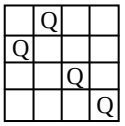
Obr. 3.23 a) Řešení úlohy čtyř dam metodou Min-conflict



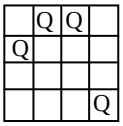
$Q_1 = \{3,1,2,1\}$



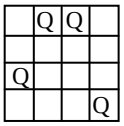
$Q_2 = \{1,3,2,2\}$



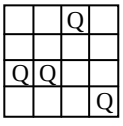
$Q_3 = \{1,2,1,2\}$



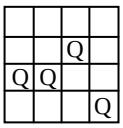
$Q_4 = \{2,2,1,0\}$
$Q_1 = \{3,1,1,1\}$



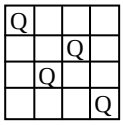
$Q_2 = \{1,3,1,2\}$



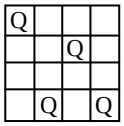
$Q_3 = \{1,1,3,2\}$



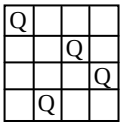
$Q_4 = \{2,1,3,0\}$
$Q_1 = \{1,2,1,3\}$



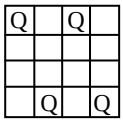
$Q_2 = \{2,3,1,1\}$



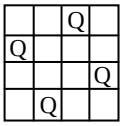
$Q_3 = \{1,0,3,2\}$
$Q_4 = \{2,2,1,2\}$



$Q_1 = \{0,1,2,2\}$
$Q_2 = \{3,2,2,0\}$
$Q_3 = \{1,1,3,2\}$



$Q_4 = \{2,2,0,2\}$
$Q_1 = \{1,0,3,1\}$



Cíl

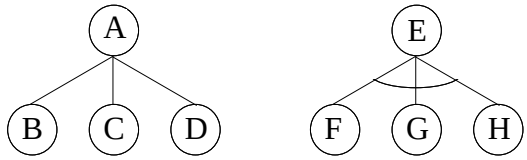
Obr. 3.23 b) Řešení úlohy čtyř dam metodou Min-conflict (jiný výchozí stav, než v Obr. 3.23 a)

3.4 Metody založené na rozkladu úloh na podproblé-my

Metody založené na rozkladu úloh na podproblémy

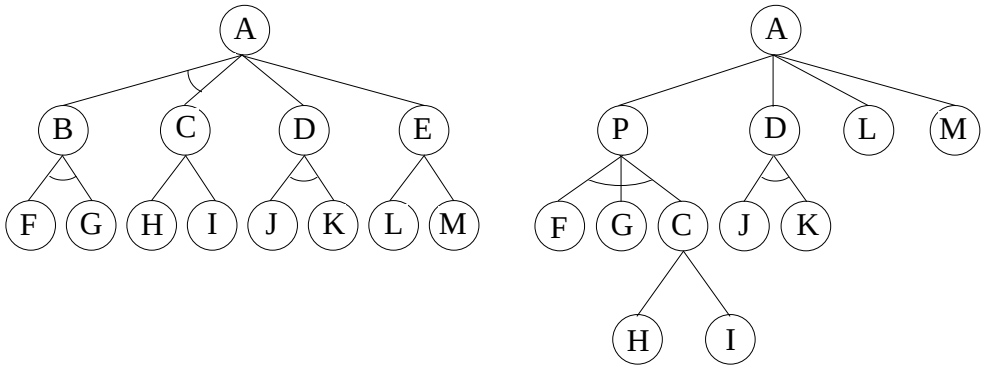
Metody řešení úloh založené na rozkladu úlohy na podproblémy jsou přirozenými metodami, které při řešení obtížných problémů používá i člověk.

Existují dva typy podproblémů (Obr. 3.24). Problém A na tomto obrázku je řešitelný, je-li řešitelný alespoň jeden z podproblémů B, C, D, problém E je řešitelný, jsou-li řešitelné všechny podproblémy F, G a H. Problém A budeme nazývat problémem OR a problém E problémem AND (poznamenejme, že někteří autoři označují tyto uzly právě naopak).



Obr. 3.24 Problémy OR a AND

Případný smíšený problém (uzel A na Obr. 3.25) můžeme převést na „čisté“ AND a OR problémy zavedením pomocných uzlů (uzel P na Obr. 3.25), resp. vynecháním nedůležitých uzlů (Uzel E na Obr. 3.25).



Obr. 3.25 Převod smíšených uzlů na uzly AND a OR

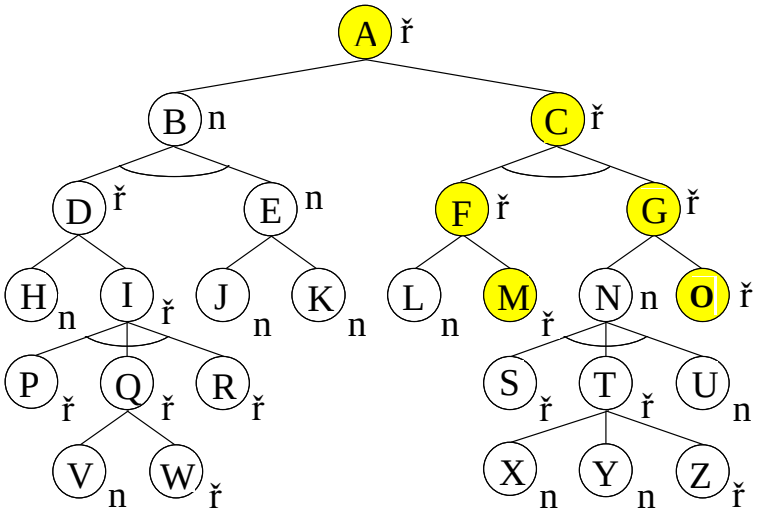
3.4.1 AO algoritmus

AO algoritmus

AO algoritmus je základním neinformovaným algoritmem (viz také Obr. 3.26):

1. Sestroj prázdné seznamy OPEN a CLOSED. Do seznamu OPEN ulož počáteční uzel (problém).
2. Vyjmi uzel zleva ze seznamu OPEN a označ jej jako uzel *X*.
3. a) Je-li uzel (problém) *X* řešitelný, přenes informaci o jeho řešitelnosti jeho předchůdcům. Je-li řešitelný počáteční problém, ukonči řešení jako úspěšné - vytvoř a vrať relevantní část AND/OR grafu.
   b) Není-li uzel (problém) *X* řešitelný a nelze-li jej rozložit na podproblémy, přenes informaci o jeho neřešitelnosti jeho předchůdcům. Není-li řešitelný počáteční problém, ukonči řešení jako neúspěšné.
   c) Expanduj *X* (rozlož *X* na podproblémy) a všechny jeho následníky ulož do OPEN.
4. Ulož *X* do CLOSED.
5. Je-li seznam OPEN prázdný, ukonči řešení jako neúspěšné, jinak se vrať na bod 2.

Poznamenejme, že podobně jako u prohledávání stavového prostoru můžeme i nyní vyšetřovat řešitelnost problému procházením AND/OR grafu do hloubky, nebo do šířky (podle toho, který uzel se vyjme v bodu 2 ze seznamu OPEN).

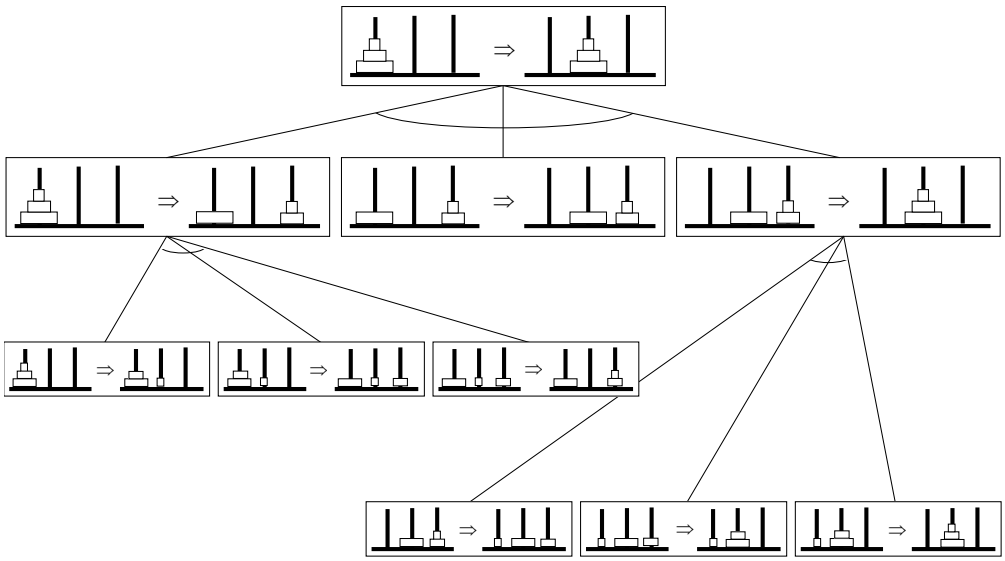


Obr. 3.26 Ukázka řešení demonstrační úlohy pomocí základního AO algoritmu

Na Obr. 3.27 je ukázán postup řešení známé úlohy Hanojských věží (přenes po jednom kroužky ze zdrojové tyčky na cílovou tyčku s použitím jedné pomocné tyčky tak, aby nikdy neležel větší kroužek na menším): Každý problém přenesení věže s n kroužky se rozkládá rekurzívně na tři podproblémy:

1. Přenes věž s n-1 kroužky na pomocnou tyčku za pomoci cílové tyčky.
2. Přenes největší kroužek ze zdrojové na cílovou tyčku (triviální a tudíž řešitelný problém – vede k použití jediného operátoru).
3. Přenes věž s n-1 kroužky z pomocné na cílovou tyčku za pomoci zdrojové tyčky.

Je zřejmé, že všechny (pod)problémy této úlohy jsou typu AND.



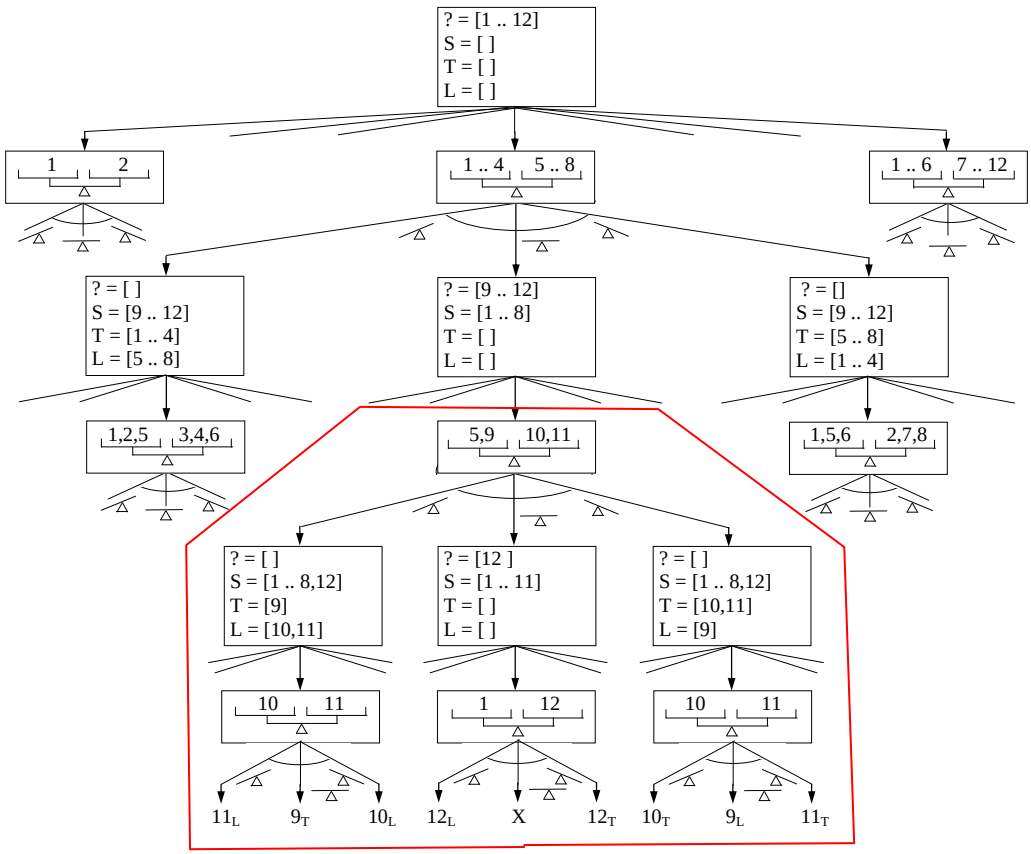
Obr. 3.27 Ukázka řešení úlohy Hanojských věží základním AO algoritmem

Klasickou úlohou řešitelnou pomocí rozkladu na AND a OR podproblémy je úloha 12 míčků:

V sadě 12 míčků má jeden z nich nepatrně rozdílnou váhu (je buď lehčí, nebo těžší, než každý z ostatních míčků). Úlohou je nalézt tento míček pomocí vážení na jazýčkových vahách a minimalizovat přitom potřebná vážení.

K reprezentaci problému vytvoříme 4 seznamy: Seznam **?** obsahuje čísla všech míčků, o kterých nemáme žádnou informaci (tj. v počátečním stavu obsahuje čísla všech dvanácti míčků), seznam **S** obsahuje čísla míčků, které mají správnou váhu, seznam **T** čísla míčků, mezi nimiž může být těžší míček a seznam **L** čísla míčků, mezi nimiž může být lehčí míček. Poslední tři seznamy jsou v počátečním stavu prázdné. Na Obr. 3.28 a Obr 3.29 je ukázán princip řešení úlohy (Obr 3.29 popisuje podrobněji část stromu, označenou na Obr. 3.28 červeným polygonem):

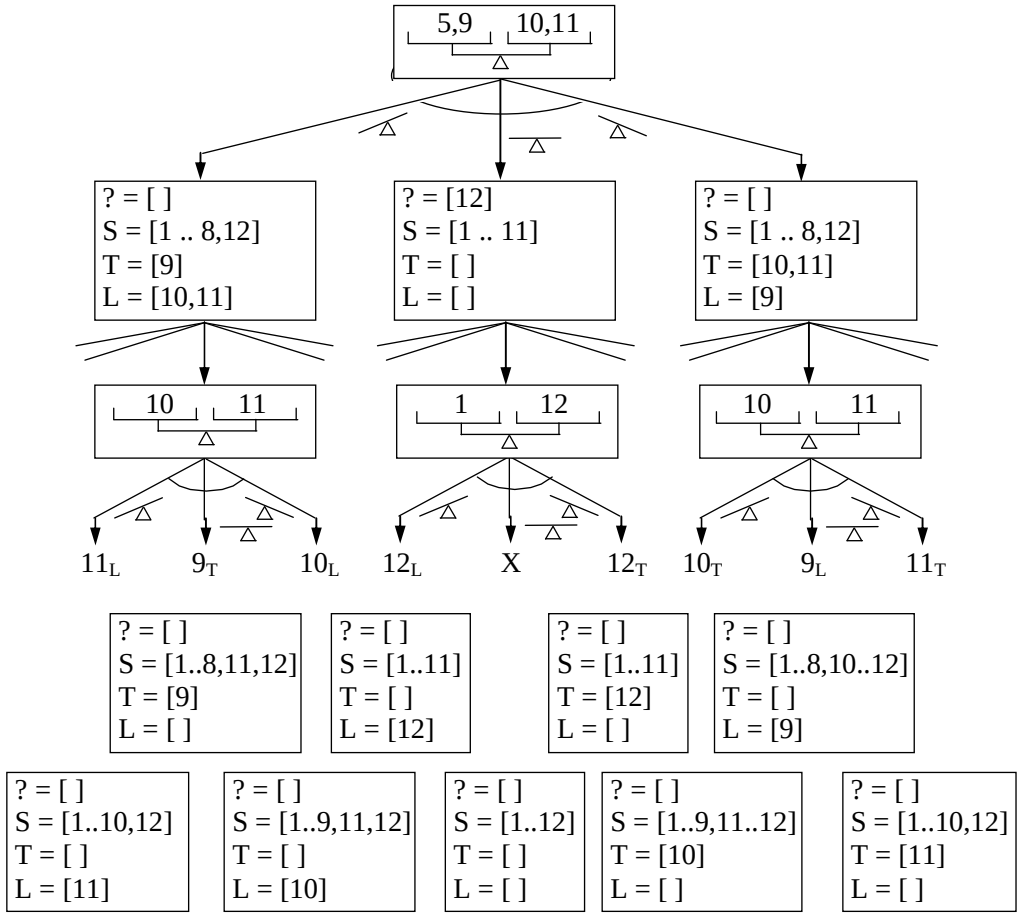
1. Ke každému vážení vybereme různé kombinace míčků (smysl mají pouze vážení, při kterých je na obou miskách jazýčkových vah stejný počet míčků). Jde tedy o problémy/uzly typu OR
2. Na základě výsledku každého vážení musíme být schopni zpracovat výsledek tohoto vážení (Levá miska je níže, levá miska je výše, misky jsou ve vodorovné poloze), tj. upravit obsah výše zmíněných seznamů. Jde tedy o problémy/uzly typu AND.



Obr. 3.28 Princip řešení úlohy dvanácti míčků pomocí základního AO algoritmu

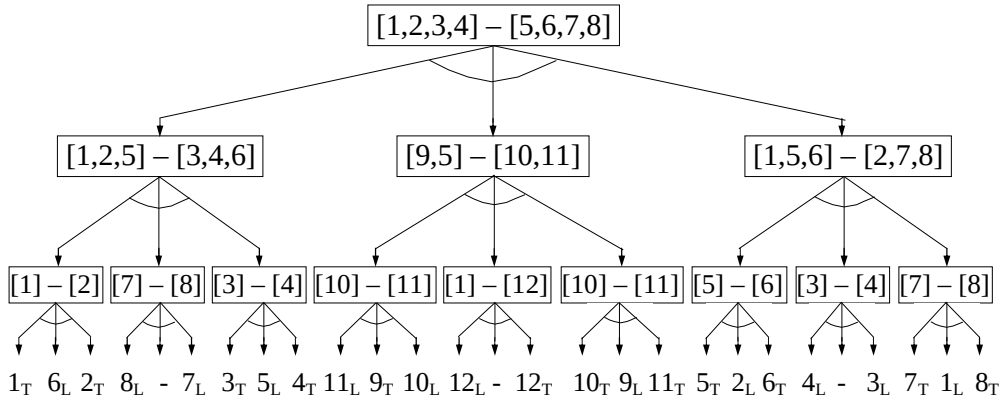
Princip řešení úlohy je zřejmý z obrázku – popišme si podrobněji situaci při vážení míčků 1,2,3,4 na levé misce a 5,6,7,8 na pravé misce:

- Pokud se jazýček váhy nevychýlí, upraví se pouze dva seznamy: Ze seznamu ? se odstraní čísla všech osmi vážených míčků a tato čísla se přesunou do seznamu S.
- Pokud výchylka jazýčku ukazuje, že je níže levá miska, pak buď mezi míčky na levé misce je těžší míček, nebo mezi míčky na pravé misce je lehčí míček, v obou případě dosud neválené míčky mají stejné váhy. Upraví se proto obsahy všech seznamů: Čísla míčků z levé misky se uloží do seznamu T, čísla míčků z pravé misky do seznamu L a všechna čísla míčků ze seznamu ? do seznamu S.
- Pokud výchylka jazýčku ukazuje, že je níže pravá miska, pak buď mezi míčky na levé misce je lehčí míček, nebo mezi míčky na pravé misce je těžší míček, v obou případě dosud neválené míčky mají stejné váhy. Upraví se proto obsahy všech seznamů: Čísla míčků z levé misky se uloží do seznamu L, čísla míčků z pravé misky do seznamu T a všechna čísla míčků ze seznamu ? do seznamu S.



Obr. 3.29 Detail označený na Obr. 3.28 červeným polygonem

Na Obr. 3.30 je uvedeno optimální řešení, kdy lze nalézt míček s odlišnou hmotností během pouhých tří vážení.



Obr. 3.30 Optimální řešení úlohy dvanácti míčků

Pokud lze jednotlivé podproblémy ohodnocovat pomocí nějaké heuristické funkce, můžeme v každém uzlu OR „přepínat“ mezi řešenými podproblémy a řešit tak nejnadějnější podstrom. Ohodnocení bývá číselné (0 znamená triviální, tj. řešitelný uzel, neřešitelný uzel se označuje jak FUTILITY (marnost)). Informovaný AO algoritmus se nazývá AO* algoritmem:

1. Sestroj strukturu G pro reprezentaci AND/OR stromu a umísti do ní počáteční uzel s označením INIT s jeho ohodnocením. Urči hodnotu FUTILITY, která určuje maximální povolenou cenu řešení.
2. Je-li uzel UNIT označen jako SOLVED, ukonči řešení jako úspěšné (řešení je dáno nejnadějnějším podstromem stromu G). Je-li hodnota uzlu INIT větší nebo rovna hodnotě FUTILITY, ukonči prohledávání jako neúspěšné. Jinak pokračuj.
3. Procházej nejnadějnějším podstromem až narazíš na neexpandovaný uzel. Tento uzel označ NODE.
4. Expanduj uzel NODE. Jestliže NODE nemá žádného následníka, přiřaď mu hodnotu FUTILITY (je ekvivalentní sdělení, že uzel není řešitelný) a přejdi na bod 7. Jinak pokračuj.
5. Představují-li někteří bezprostřední následníci elementární úlohy, označ je jako SOLVED (tj. přiřaď jim nulové ohodnocení), u zbývajících ohodnocení odhadni (vypočítej).
6. Připoj bezprostřední následníky uzlu NODE ke stromu G a přenes informace o jejich ohodnocení směrem nahoru k uzlu INIT takto:
   - ohodnocení uzlů AND je rovno nenižšímu ohodnocení jeho bezprostředních následníků.
   - ohodnocení uzlů OR je rovno nejvyššímu ohodnocení jeho bezprostředních následníků.
7. V uzlech OR označ nejnadějnější podstromy (vždy hranu k nejlépe ohodnocenému bezprostřednímu následníku).
8. Vrať se na bod 2.

3.5 Metody hraní her

Metody hraní her

Při popisu metod hraní her se omezíme pouze na hry se dvěma (proti)hráči, kteří se po jednotlivých tazích hry pravidelně střídají. Oba hráči mají úplnou informaci o stavu hry, hrají čestně a oba si přejí zvítězit.

Problém spočívá v nalezení tahu hráče, který je právě na tahu (hráč A). Pro

tohoto hráče bude problém považován za řešitelný, povede-li k jeho výhře alespoň jeden z jeho možných tahů (problém OR). Protože v dalším tahu táhne soupeř (hráč B), musí být pro každý tah vedoucí k výhře hráče A pro všechny tahy hráče B pro tohoto hráče neřešitelné, jinými slovy, všechny tyto tahy hráče B musí být řešitelné pro hráče A (problém AND). Hledání tahu vedoucího k výhře tak vede na klasické prohledávání AND/OR grafu.

Po výběru a realizaci tahu hráče A se „vše zapomene“, v dalším tahu hraje hráč B, a hráč A pak opět vybírá svůj tah z již nového konkrétního stavu úlohy!!!

Pozn.: Při výběru tahu hráče B, se tento hráč považuje za hráče A a řešitelnost stavů hry hodnotí přirozeně opačně.

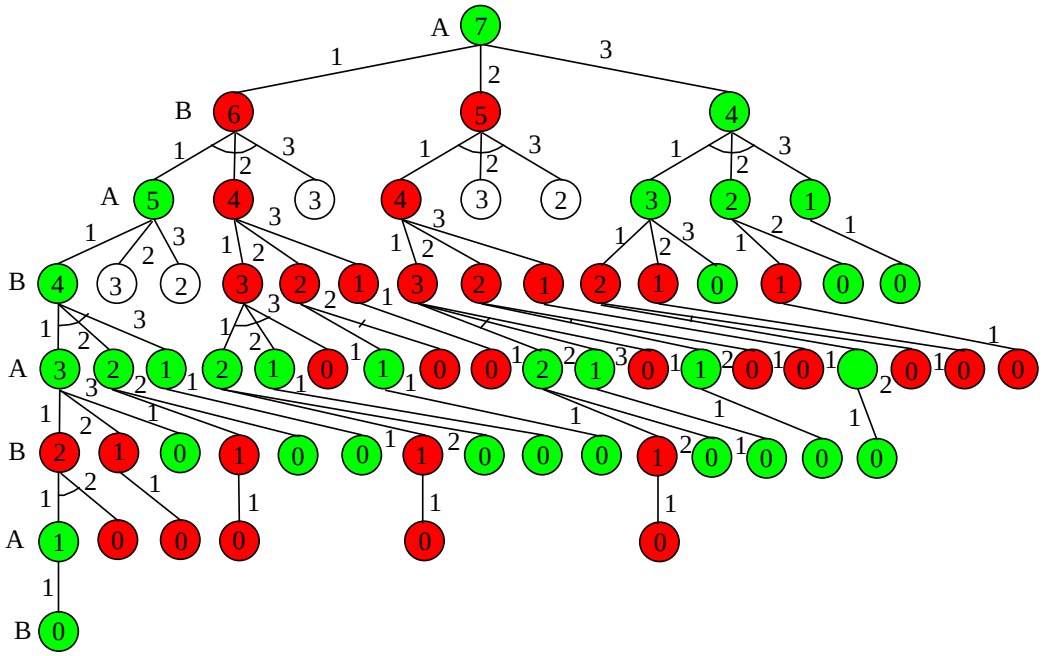
Právě popsané hry lze rozdělit na jednoduché hry, složité hry a hry s neurčitostí.

3.5.1 Jednoduché hry

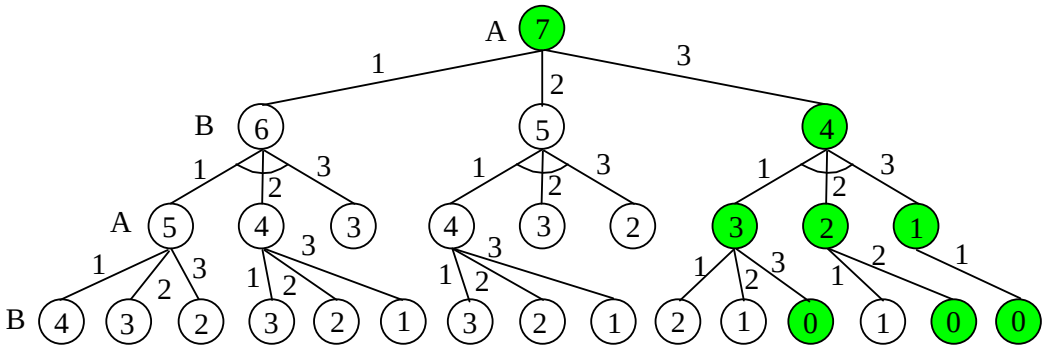
Jednoduché hry

Za jednoduché hry budeme považovat hry, u kterých lze v reálném čase prohledat celý jejich AND/OR graf. K řešení takových her lze použít algoritmus AO uvedený v kap. 3.4 s tím, že v případě řešitelnosti není nutné vracet celou část AND/OR grafu, ale pouze tah hráče A, který vede k jeho výhře.

K demonstraci hledání výběru tahu u jednoduché hry použijeme následující hru se zápalkami: Hráči střídavě odebírají z hromádky zápalek jednu, dvě, nebo tři zápalky, prohrává ten hráč, který již nemá co odebrat. Výchozím stavem hry je hromádka s libovolným počtem zápalek (dále uvažujme pro jednoduchost pouze 7 zápalek). Prohledávání AND/OR grafu do šířky a do hloubky jsou ukázána na následujících dvou obrázcích (Obr. 3.31 a Obr. 3.32). Červenou barvou jsou označeny neřešitelné uzly, zelenou barvou řešitelné uzly (vše pro hráče A). Čísla v uzlech značí aktuální stav hry (počet zápalek na hromádce), čísla u hran značí počet odebíraných zápalek.



Obr. 3.31 Prohledávání AND/OR grafu úlohy 7 zápalek AO algoritmem metodou DFS



Obr. 3.32 Prohledávání AND/OR grafu úlohy 7 zápalek AO algoritmem metodou BFS, resp. IDS

Nevybarvené uzly na Obr. 3.31 nemusí být vyšetřovány, protože na řešitelnost bezprostředního předchůdce již nemají vliv. Nevybarvené uzly na Obr. 3.32 jsou uzly, které dosud vyšetřovány nebyly a uzly, o jejichž řešitelnosti nebylo dosud rozhodnuto.

Je zřejmé, že výchozí stav je pro hráče A příznivý – tah spočívající v odebrání tří zápalek vede k jeho výhře.

Úkol: Řešte klasickou úlohu NIM: Máte 16 zápalek ve čtyřech řadách (1+3+5+7). Hráči střídavě odebírají libovolný nenulový počet zápalek, v jednom tahu vždy z jediné řady. Prohrává ten hráč, který již nemá co odebrat (v jiné variantě hry prohrává hráč, který musí odebrat poslední zápalku).

3.5.2 Složité hry

Složité hry

Za složité hry budeme považovat takové hry, u nichž je úplné prohledávání jejich AND/OR grafů nemožné (např. šachy). Proto se u těchto her prohledává pouze do předem dané hloubky, tj. zkoumá se pouze předem zadaný počet tahů. Pokud se v této hloubce nenachází koncové problémy, u kterých lze rozhodnout o řešitelnosti/neřešitelnosti problémů (uzlů grafu) je nutné tyto problémy/uzly nějakým způsobem hodnotit.

V dále popsaných metodách se k ohodnocení vnitřních problémů/uzlů používají celočíselné hodnotící funkce, jejíž kladné hodnoty znamenají příznivé stavy pro hráče A (čím větší, tím příznivější), záporné pak příznivý stav pro hráče B (čím menší, tím příznivější). Vítězství či prohra jsou hodnoceny maximálními hodnotami uvažovaného číselného intervalu (např. pro ShortInt na 1 Byte hodnotami +127, resp. –127).

Je zřejmé, že hráč A si vybírá tahy vedoucí ke stavům s maximálními ohodnoceními, a že hráč B si vybírá tahy vedoucí ke stavům s minimálními ohodnoceními. Základní algoritmus/metoda hraní her pracující na uvedeném principu se proto nazývá MiniMax.

Algoritmus MiniMax

Algoritmus MiniMax

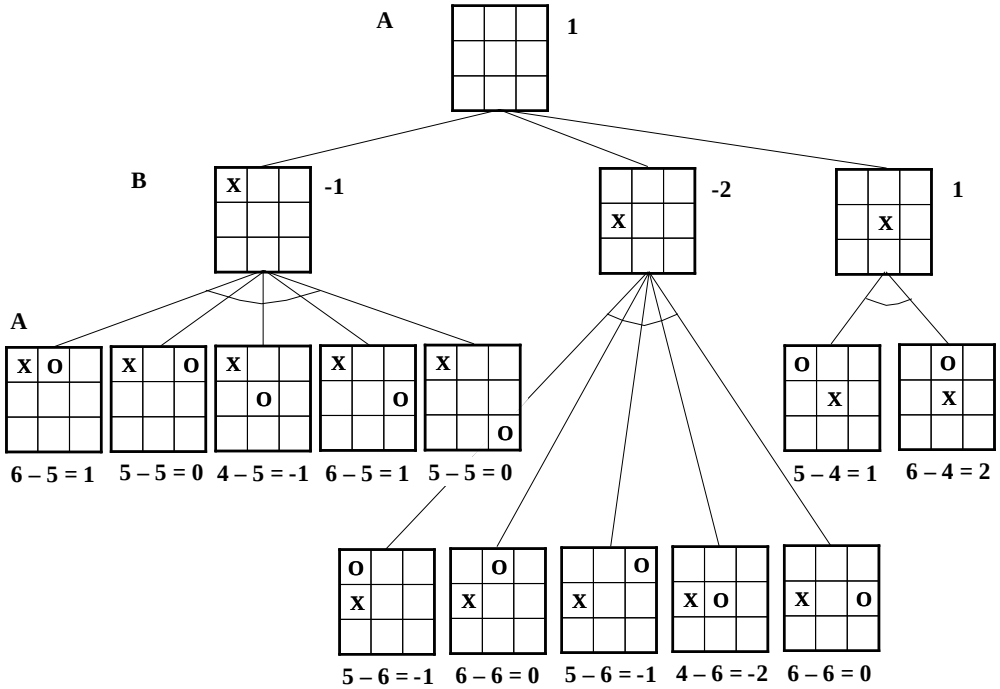
Základem algoritmu MiniMax je rekurzivní procedura, nazvěme ji také MiniMax, která se zavolá pro aktuální stav hry (kořenový uzel AND/OR grafu) a hráče A. Tato procedura vrací ohodnocení uzlu a pro hráče A i tah k uzlu s maximálním ohodnocením, tj. tah, který je v daném stavu hry pro tohoto hráče

nejvýhodnější. Procedura MiniMax předpokládá, že je zadána maximální hloubka prohledávání (počet zkoumaných tahů).

Procedura MiniMax:

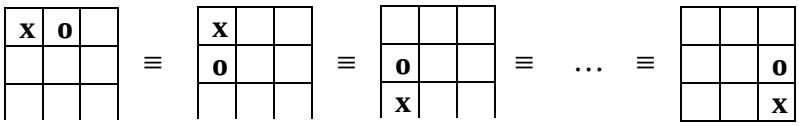
1. Nazvěme předaný vstupní uzel uzlem X.
2. Je-li uzel X listem (konečným stavem hry, nebo uzlem v maximální hloubce) vrať ohodnocení tohoto uzlu. Jinak pokračuj.
3. Je-li na tahu hráč A, tak postupně pro všechny jeho možné tahy (bezprostřední následníky uzlu X a hráče B) volej proceduru MiniMax a vrať maximální z navrácených hodnot. Je-li X kořenovým uzlem vrať i tah, který vede k nejlépe ohodnocenému bezprostřednímu následníkovi.
4. Je-li na tahu hráč B, tak postupně pro všechny jeho možné tahy (bezprostřední následníky uzlu X a hráče A) volej proceduru MiniMax a vrať minimální z navrácených hodnot.

Praktické použití procedury MiniMax ukážeme na hledání výchozího tahu pro hru Tic-Tac-Toe (křížky - kolečka); pro jednoduchost uvažujeme pouze desku o pouhých 3×3 políčcích (Obr. 3.33). Hráči kladou střídavě své značky a zvítězí ten hráč, kterému se podaří postavit souvislou linku (vodorovnou, svislou, nebo v úhlopříčce). Nechť hráč A používá křížky a hráč B kolečka.



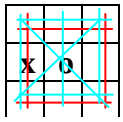
Obr. 3.33 Hledání nejvýhodnějšího tahu z počátečního stavu hry Tic-Tac-Toe metodou MiniMax

Pozn.: Na Obr. 3.33 nejsou pro jednoduchost uvedeny „zrcadlové“ stavy, například:



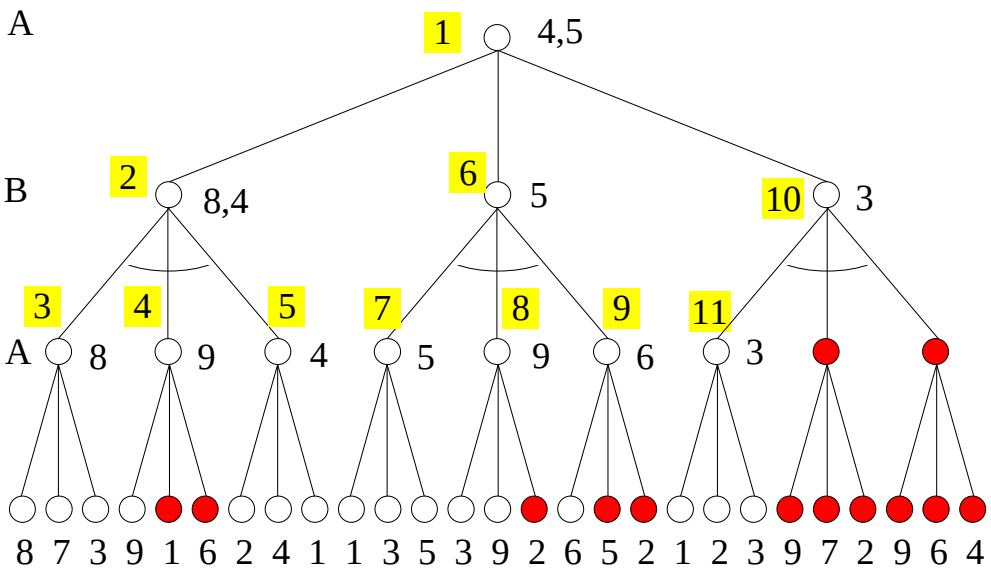
Pozn.: Příklad ohodnocení stavu:

4 – 6 = – 2



dosud možné linky:
x = 4
o = 6

Při použití procedury MiniMax dochází ke zbytečnému vyšetřování velké části AND/OR grafu z důvodu, který je blíže vysvětlen na Obr. 3.34.



Obr. 3.34 Příklad na zbytečná vyšetřování některých uzlů AND/OR grafu metodou MiniMax

Hráč A (kořenový uzel 1) volá proceduru MiniMax na svůj první tah a hráče B (uzel 2) a ten volá tuto proceduru na svůj první tah a hráče A (uzel 3). Hráč A (uzel 3) volá proceduru MiniMax postupně na všechny své možné tahy a hráče B - protože všichni jeho bezprostřední následníci jsou uzlovými listy procedura MiniMax pouze postupně vrací ohodnocení těchto listů a hráč A pak vybere (vrátí) maximální hodnotu z jejich ohodnocení (tj. číslo 8). Hráč B (uzel 2) tuto hodnotu akceptuje a volá proceduru MiniMax na svůj druhý tah a hráče A (uzel 4). První bezprostřední následník tohoto uzlu (listový uzel) vrací hodnotu 9. Je zřejmé, že prohledávání dalších následníků uzlu 4 je zbytečné, protože již nyní je jasné, že tento uzel vrátí hodnotu ≥ 9, a že hráč B (uzel 2), který si vybírá tah s minimálním ohodnocením, si tento tah nevybere, protože ohodnocení jeho prvního tahu je menší. Hráč B (uzel 2) pak volá proceduru MiniMax na svůj poslední tah a hráče A (uzel 5). Hráč A (uzel 5) opět volá postupně proceduru MiniMax na všechny své bezprostřední následníky a z vrácených hodnot vybere (vrátí) hodnotu maximální, tj. hodnotu 4. Protože tato hodnota je menší než hodnota 8, hráč B (uzel 2) si vybere a vrátí tuto hodnotu. Hráč A (uzel 1) hodnotu 4 akceptuje a volá proceduru MiniMax na svůj druhý možný tah a hráče B (uzel 6). Další postup pro tento tah je velmi podobný postupu pro první tah. Hráč B (uzel 6) volá postupně proceduru MiniMax na své bezprostřední následníky (uzly 7, 8 a 9) a vrátí minimum z vrácených hodnot, tj. číslo 5. Některé listové uzly, na obrázku označené červenou barvou, se opět vyšetřují zbytečně, z důvodů popsaných výše. Protože 5 > 4, akceptuje hráč A (uzel 1) tuto hodnotu (druhý tah je pro něj výhodnější, než tah první) a volá proceduru MiniMax na svůj třetí tah a hráče B (uzel 10). Hráč B (uzel 10) volá proceduru

MiniMax na svůj první tah a hráče A (uzel 11) a od tohoto uzlu dostane navrácenu hodnotu 3. Proto je již v tomto okamžiku zřejmé, že hráč B (uzel 10), který si vybírá minimum, vrátí kořenovému uzlu hodnotu ≤ 3 a hráč A (uzel 1) si tento tah nevybere. Další vyšetřování tahů hráče B (uzlu 10) je tak zbytečné.

Jak již bylo zmíněno, jsou zbytečně vyšetřované uzly metodou MiniMax označeny na obrázku červenou barvou – je jich přibližně 30% (13 ze 40ti).

Alfa a Beta řezy

Alfa a Beta řezy

Uvedenému zbytečnému vyšetřování lze zabránit velmi jednoduchým způsobem, a to zavedením tzv. alfa a beta řezů (zastavení). Alfa řezy zabrání zbytečnému vyšetřování tahů hráče A, beta řezy pak zbytečnému vyšetřování tahů hráče B. Ve skutečnosti je toto rozdělení formální, a vyšetřování se zastaví vždy, když platí $\alpha \geq \beta$ (viz níže uvedený algoritmus). Algoritmus MiniMax se zavedením alfa a beta řezů se často nazývá přímo AlfaBeta algoritmus (metoda, procedura).

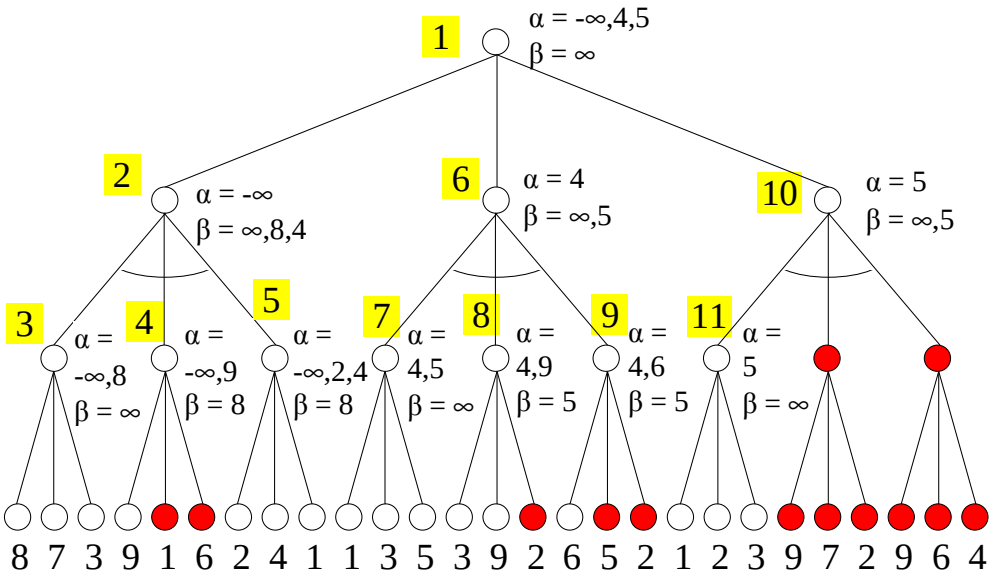
Procedura AlfaBeta:

1. Nazvěme předaný vstupní uzel uzlem X.
2. Je-li X počátečním/kořenovým uzlem, nastav $\alpha = -\infty$, $\beta = \infty$ (v praxi nastav hodnoty těchto proměnných na minimální a maximální možnou hodnotu).
3. Je-li uzel X listem (konečným stavem hry, nebo uzlem v maximální hloubce) ukonči proceduru a vrať ohodnocení tohoto uzlu.
4. Je-li uzel typu AND (na tahu je hráč B) jdi na bod 5, jinak pokračuj (uzel je typu OR, na tahu je hráč A):
   4.1. Dokud je $\alpha < \beta$, tak postupně pro první/další tah (bezprostředního následníka uzlu X a hráče B) volej proceduru AlfaBeta s aktuálními hodnotami proměnných α a β. Po každém vyšetřeném tahu nastav hodnotu proměnné α na maximum z aktuální a navrácené hodnoty .
   4.2. Ukonči proceduru, vrať aktuální hodnotu proměnné α a pro kořenový uzel vrať i tah, který vede k nejlépe ohodnocenému bezprostřednímu následníkovi.
5. (Uzel je typu AND, na tahu je hráč B):
   5.1. Dokud je $\alpha < \beta$, tak postupně pro první/další tah (bezprostředního následníka uzlu X a hráče A) volej proceduru AlfaBeta s aktuálními hodnotami proměnných α a β. Po každém vyšetřeném tahu nastav hodnotu proměnné β na minimum z aktuální a navrácené hodnoty .
   5.2. Ukonči proceduru a vrať aktuální hodnotu proměnné β.

Na Obr. 3.35 je ukázán princip alfa a beta řezů při vyšetřování grafu z Obr. 3.34. Při zavolání procedury AlfaBeta na počáteční/kořenový uzel (uzel 1) se nastaví hodnoty $\alpha = -\infty$, $\beta = \infty$. Hráč A (uzel 1) volá proceduru AlfaBeta na svůj první tah a hráče B (uzel 2) s hodnotami $\alpha = -\infty$, $\beta = \infty$, a hráč B (uzel 2) volá tuto proceduru na svůj první tah a hráče A (uzel 3), opět s hodnotami $\alpha = -\infty$, $\beta = \infty$. Hráč A (uzel 3) volá proceduru AlfaBeta postupně na všechny své možné tahy a hráče B - protože všichni jeho bezprostřední následníci jsou uzlovými listy procedura AlfaBeta pouze postupně vrací ohodnocení těchto listů a hráč A upravuje hodnotu své proměnné α na maximum z původní a navrácených hodnot ($\alpha = -\infty$, 8). Poznamenejme, že hodnota $\beta = \infty$ je stále větší než hodnota proměnné α! Hráč A (uzel 3) nakonec vrátí hodnotu 8. Hráč B (uzel 2) tuto

hodnotu akceptuje a nastaví na tuto hodnotou hodnotu proměnné β = min(8, ∞). Zdůrazněme, že hodnota proměnné α se v tomto uzlu nemění (α = -∞). Hráč B pak volá proceduru AlfaBeta na svůj druhý tah a hráče A (uzel 4), s aktuálními hodnotami proměnných α = -∞, β = 8. Hráč A (uzel 4) volá proceduru AlfaBeta na svůj první tah. Vrácená hodnota je 9 a tato hodnota bude novou hodnotou proměnné α uzlu 4. V tomto okamžiku je však (α = 9) > (β = 8) a vyšetřování uzlu 4 je „násilně“ ukončeno (alfa řez) – vrací se hodnota 9 a hodnota β = 8 uzlu 2 se nemění (Hráč B si vybírá minimum). Hráč B pak volá proceduru AlfaBeta na svůj třetí/poslední tah a hráče A (uzel 5), s aktuálními hodnotami proměnných α = -∞, β = 8. Hráč A (uzel 5) volá postupně proceduru AlfaBeta na všechny své bezprostřední následníky a postupně upravuje aktuální hodnotu proměnné α = -∞, 2, 4 (Hodnota β = ∞ je stále větší než hodnota proměnné α!). Konečnou hodnotu proměnné α, tj. hodnotu 4 vrací hráč A (uzel 5) hráči B (uzlu 2). Hráč B (uzel 2) přepíše touto hodnotou hodnotu proměnné β (β = 4) a tuto hodnotu vrátí hráči A (uzlu 1). Nová hodnota proměnné α uzlu 1 je tak 4. Hráč A (uzel 1) volá proceduru AlfaBeta a hráče B (uzel 6) s aktuálními hodnotami α = 4, β = ∞ a postup uvedený výše se v podstatě opakuje. Aktuální hodnota proměnné α hráče A (uzlu 1) se pak zvýší na 5 (α = 5).

Za povšimnutí stojí vyšetřování uzlů 10 a 11. Hráči B (uzlu 10) jsou hráčem A (uzlem 1) předány hodnoty α = 5, β = ∞. Tento hráč (uzel 10) předá tyto hodnoty pro svůj první tah hráči A (uzlu 11). Tento hráč (uzel 11) volá s těmito hodnotami proceduru AlfaBeta postupně na všechny své bezprostřední následníky. Protože tito následníci vrací hodnoty menší než 5, hodnota α hráče A (uzel 11) se nezmění a nakonec je vrácena hráči B (uzlu 10). Ten upraví hodnotu své proměnné β (β = 5) a protože v tomto okamžiku se hodnota β = 5 rovná hodnotě α = 5, vyšetřování uzlu 10 okamžitě končí (beta řez). Hráči A (uzlu 1) je vrácena hodnota 5 a protože tato hodnota je stejná jako aktuální hodnota α uzlu 1, tak se již nic nemění (ukazatel na nejlepší tah hráče A (uzlu 1) ukazuje na druhý tah !!!).



Obr. 3.35 Příklad práce procedury AlfaBeta (alfa a beta řezů)

Úkol: Metodami Minimax a AlfaBeta vyšetřete grafy z Obr. 3.35, jestliže ohodnocení listových uzlů bude následující:

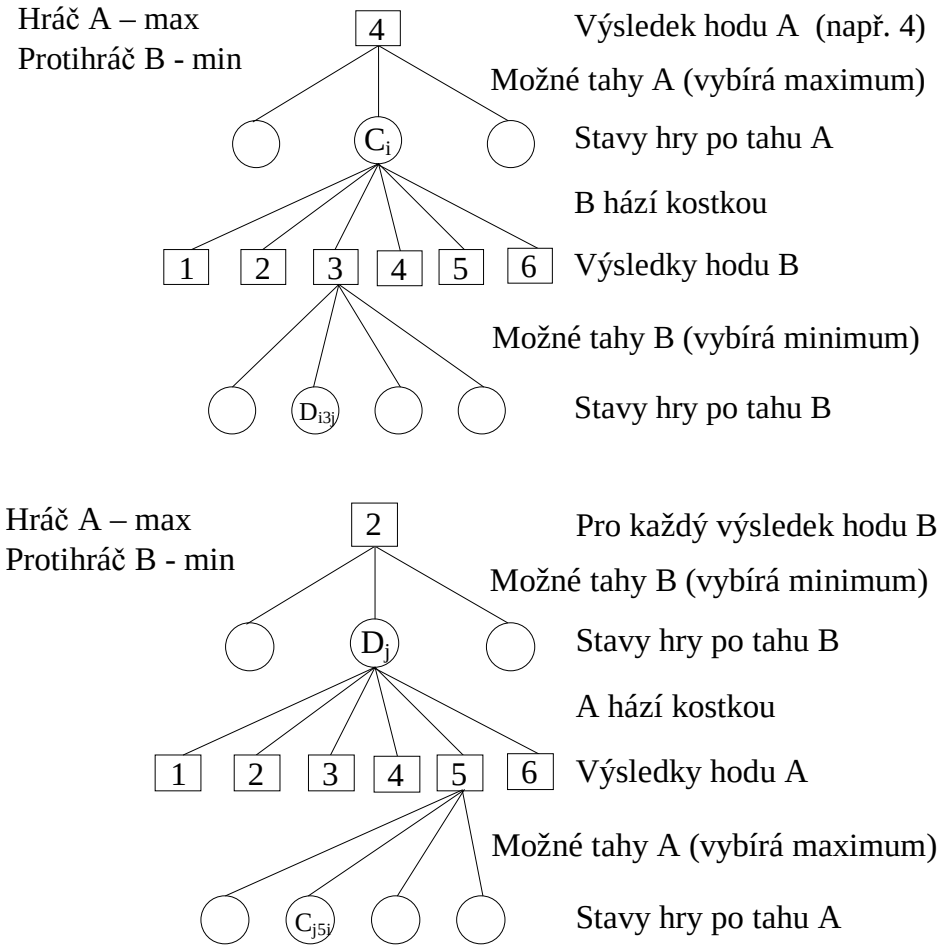
a) 8 7 3 6 9 6 8 4 1 1 3 5 3 9 2 6 5 2 1 2 4 3 7 2 9 6 4
b) 1 6 2 6 9 6 4 4 1 1 3 4 3 9 2 6 5 2 1 2 4 3 7 2 9 6 4
c) 2 2 2 1 2 3 4 4 1 1 1 1 3 2 2 6 5 2 1 3 3 3 6 2 8 2 6

3.5.3 Hry s neurčitostí

Hry s neurčitostí

Existuje řada podobných her, které opět hrají dva protihráči, kteří se po jednotlivých tazích hry pravidelně střídají, mají úplnou informaci o stavu hry, hrají čestně a oba si přejí zvítězit. Na rozdíl od výše uvedených her však při hře používají kostku, resp. kostky, a do hry tak vstupuje neurčitost – náhoda.

Základní princip her s kostkou je naznačen na Obr. 3.36. Hráč A je na tahu a právě hodil kostkou (uvažujeme klasickou šestistrannou kostku - na obrázku je výsledkem hodu číslo 4, tato skutečnost je však pro další úvahy nepodstatná).



Obr. 3.36 Princip her (výběru nejlepšího tahu) pro hry s kostkou

Hráč A nyní ví, které své tahy C_i může uskutečnit. Může samozřejmě přímo ohodnotit jednotlivé stavy (bezprostřední následníky) a vybrat tah vedoucí ke stavu s maximálním ohodnocením. Tento přístup sice uplatňuje řada lidí, ale pro nás je zajisté nezajímavý.

Hráč A proto bude postupovat při hodnocení každého stavu C_i složitějším způsobem (horní část Obr. 3.36). Vyjde z úvahy, že hráč B by pro známý výsledek hodu vybral tah do stavu D_j s minimálním ohodnocením. Protože však hráč B výsledek svého hodu nezná, může pracovat pouze s ohodnocením očekávaným, které se v literatuře označuje jako *expectimin* (očekávané minimum):

$$expectimin(C_i) = \sum_k P(h_k) * \min_j (D_{ikj}),$$

kde značí

- h_k … *k*-tý výsledek hodu (1, 2, 3, 4, 5, nebo 6),
- $P(h_k)$ … pravděpodobnost *k*-tého výsledku (pro hry s jednou kostkou je pravděpodobnost všech výsledků stejná: $P(h_k) = 1/6$, pro hry se dvěma kostkami jsou pravděpodobnosti výsledků se stejnými čísly na kostkách 1/36 a pravděpodobnosti výsledků s různými čísly na kostkách 1/18),
- D_{jki} … ohodnocení stavu D_j dosažitelného ze stavu C_i po *k*-tém výsledku hodu kostkou.

Ohodnocení *expectimin* je tedy dáno součtem ohodnocení po všech možných výsledcích hodu kostky, kdy každé jednotlivé ohodnocení je dáno součinem pravděpodobnosti daného výsledku hodu kostky a následného minimálního ohodnocení stavu, kterého je možné po daném hodu dosáhnout. Hráč A si pak vybírá tah do stavu C_i s maximální hodnotou *expectimin*.

Podobným způsobem se postupuje při vyšetřování očekávaného ohodnocení uzlu D_{ijk}, který pro jednoduchost označujeme dále i v dolní části Obr. 3.36 pouze jako D_j. Protože hráč A vybírá maximum z možných ohodnocení, označuje se toto hodnocení jako *expectimax* (očekávané maximum):

$$expectimax(D_j) = \sum_k P(h_k) * \max_i (C_{jki}),$$

kde značí

- h_k … *k*-tý výsledek hodu (1, 2, 3, 4, 5, nebo 6),
- $P(h_k)$ … pravděpodobnost *k*-tého,
- C_{jki} … ohodnocení stavu C_i dosažitelného ze stavu D_j po *k*-tém výsledku hodu kostkou.

Přestože se zdá uvedený postup na první pohled složitý, lze opět snadno realizovat rekurzivní proceduru.

Procedura ExpectMiniMax:

1. Nazvěme předaný vstupní uzel uzlem X.
2. Je-li uzel X listem (konečným stavem hry, nebo uzlem v maximální hloubce) vrať ohodnocení tohoto uzlu. Jinak pokračuj.
3. Je-li na tahu hráč A, tak postupně pro všechny jeho možné tahy (bezprostřední následníky uzlu X a hráče B) volej proceduru ExpectMiniMax a vrať maximální hodnotu z hodnot *expectimax*. Je-li X kořenovým uzlem vrať i tah, který vede k nejlépe ohodnocenému bezprostřednímu následníkovi.
4. Je-li na tahu hráč B, tak postupně pro všechny jeho možné tahy

(bezprostřední následníky uzlu X a hráče A) volej proceduru ExpectMiniMax a vrať minimální hodnotu z hodnot *expectimin*.

Pozn: Pro kořenový uzel je již znám výsledek hodu kostkou. Proto při vyčíslování vztahů *expectimax* je pouze jedna z hodnot $P(h_i)$ jedničková, zatímco ostatní jsou nulové.

3.6 Shrnutí

Shrnutí

V této kapitole byly popsány nejznámější metody řešení úloh UI: metody založené na prohledávání stavového prostoru (neinformované, informované, metody lokálního prohledávání), metody řešení úloh, metody založené na rozkladech úloh na podproblémy a metody hraní her dvou protihráčů.

Pro úspěšné absolvování předmětu IZU je nutné znát principy/algoritmy metod BFS, UCS, DFS, DLS, IDS, Backtracking, A^*, Hill-climbing, Forvard checking, AO, MiniMax (včetně Alfa a Beta řezů) a jejich ohodnocení podle kritérií uvedených v kapitole 3.2.3.

3.7 Kontrolní otázky

Kontrolní otázky

1. Znáte skupiny metod řešení úloh?
2. Znáte kritéria pro hodnocení jednotlivých metod
3. Znáte metody založené na prohledávání stavového prostoru?
4. Můžete popsat princip metod BFS, UCS, DFS, DLS a IDS?
5. Můžete popsat princip metody Backtracking?
6. Můžete popsat princip metod BestFS (A^*)?
7. Můžete popsat princip metody Hill-climbing?
8. Znáte metody řešení úloh s omezujícími podmínkami?
9. Můžete popsat princip metody Forvard checking?
10. Můžete popsat princip AO algoritmu?
11. Můžete popsat princip algoritmu MiniMax a Alfa a Beta řezů?

4 LOGIKA A UMĚLÁ INTELIGENCE

10 hod

Cílem této kapitoly je seznámit studenty se základními principy výrokové a predikátové logiky v rozsahu potřebném pro další výklad problematiky UI. Je zde vysvětleno především odvozování závěrů z konjunkcí podmínek/formulí prováděním důkazů nesplnitelnosti negací dokazovaných formulí, postup při unifikaci proměnných v případě predikátové logiky a použití rezoluční metody při strojovém dokazování.

4.1 Úvodní informace

Úvodní informace

Logika je považována ze jeden ze základních „stavebních kamenů“ klasické umělé inteligence. Umožňuje jak snadnou reprezentaci znalostí, tak i práci s těmito znalostmi. Má však také některé nedostatky, z nichž nejzávažnějším je velmi obtížná práce s nejistými, neúplnými a dočasnými znalostmi.

V dalším textu bude nejprve stručně popsána výroková logika. Ta má sice poměrně slabou vyjadřovací sílu a v UI se prakticky nepoužívá, dovoluje nám však snadněji pochopit některé principy, které používá i predikátová logika prvního řádu, jejíž popis je základním cílem této kapitoly. V závěru bude uveden princip základní metody pro odvozování znalostí, tzv. rezoluční metody, a bude také naznačeno použití této metody při řešení úloh.

4.2 Výroková logika

Výroková logika

Formální systém každé logiky, a tedy i výrokové logiky, tvoří tři složky:

1) *jazyk* - množiny symbolů pro označení logických konstant, atomických formulí, spojek a pomocných objektů,
2) *axiomy* - formule představující základní formule jazyka,
3) *odvozovací pravidla* - pravidla umožňující na základě syntaxe základních formulí odvodit nové formule.

Axiomy výrokové logiky a odvozovací pravidlo (ve výrokové logice je jediné) nebudeme dále používat a uvádíme je pouze pro informaci:

- Axiómy výrokové logiky:
  - $A \Rightarrow (B \Rightarrow A)$,
  - $A \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$,
  - $(\neg A \Rightarrow \neg B) \Rightarrow (B \Rightarrow A)$.
- Odvozovací pravidlo výrokové logiky (modus ponens):
  - z formulí A, $A \Rightarrow B$ odvoď formuli B.

Základními prvky jazyka výrokové logiky jsou prvotní formule (atomy), logické spojky $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ (negace, disjunkce, konjunkce, implikace a ekvivalence) a kulaté závorky.

S prvotními formulemi je vždy spojena jedna ze dvou logických hodnot T/F (True/False – pravdivá/nepravdivá).

Z prvotních formulí se vytvářejí výrokové formule (dále jen formule) takto:

- každá prvotní formule je formule,
- jsou-li A a B formule, pak i $\neg A$, $A \vee B$, $A \wedge B$, $A \Rightarrow B$ a $A \Leftrightarrow B$ jsou formule. Logická hodnota formulí je dána následující tabulkou:

A	B	¬A	¬B	A ∨ B	A ∧ B	A ⇒ B	A ⇔ B
F	F	T	T	F	F	T	T
F	T	T	F	T	F	T	F
T	F	F	T	T	F	F	F
T	T	F	F	T	T	T	T

V případě složitějších výrazů se respektují priority logických spojek podle výše uvedeného pořadí (nejvyšší prioritu má spojka ¬ a nejnižší spojka ⇔), kulaté závorky umožňují prioritu standardním způsobem měnit.

Binární spojky ∨ , ∧ , ⇔ jsou symetrické, a proto pořadí argumentů nemá na hodnotu formule vliv, spojka ⇒ je však nesymetrická a výsledná hodnota formule na pořadí argumentů závisí. (formule A se nazývá antecedent a formule B konsekvent formule A ⇒ B).

Pravdivostní ohodnocení prvotních formulí (přiřazení logických hodnot) v dané formuli se nazývá interpretací této formule.

Formule se nazývá tautologií (logicky pravdivou formulí), je-li pravdivá ve všech interpretacích, a kontradikcí (nesplnitelnou formulí), je-li ve všech interpretacích nepravdivá.

Formule je splnitelná, existuje-li alespoň jedna interpretace ve které je pravdivá. Formule není logicky pravdivá (není tautologií), existuje-li alespoň jedna interpretace ve které je nepravdivá.

Dvě formule jsou ekvivalentní, nabývají-li stejných logických hodnot ve všech interpretacích.

Příklady: formule (A ∨ ¬A) je tautologií
formule (A ∧ ¬A) je kontradikcí
formule (A ∨ ¬B) je splnitelnou formulí
formule ¬(A ∨ B) a (¬A ∧ ¬B) jsou ekvivalentní

Literálem se rozumí prvotní formule, nebo její negace.

Formule je v disjunktivní normální formě, je-li disjunkcí několika formulí, které jsou konjunkcemi literálů. V žádné z těchto formulí se však nesmí vyskytovat některá prvotní formule společně se svou negací.

Formule je v konjunktivní normální formě, je-li konjunkcí několika formulí, které jsou disjunkcemi literálů. V žádné z těchto formulí se opět nesmí vyskytovat některá prvotní formule společně se svou negací.

Disjunkce literálů $L_1 \vee L_2 \vee, ..., \vee L_n$ se nazývá klauzulí. Jednotková klauzule obsahuje jeden literál, prázdná klauzule neobsahuje žádný literál a označuje se symbolem □.

Klauzule s komplementárním párem literálů (např. A ∨ ¬A) je tautologií.

Formule G je logickým důsledkem (logicky vyplývá z) formulí $\{F_1, F_2, \ldots F_n\}$, je-li pravdivá v každé interpretaci I, v níž je pravdivá formule $F_1 \wedge F_2 \wedge, ..., \wedge F_n$, tj. je-li formule $F_1 \wedge F_2 \wedge, ..., \wedge F_n \Rightarrow G$ tautologií.

Formule $F_1 \wedge F_2 \wedge, ..., \wedge F_n \Rightarrow G$ se pak nazývá teorémem, formule G tvrzením a formule $\{F_1, F_2, \ldots F_n\}$ premisami (postuláty) tohoto teorému.

Konečná posloupnost formulí A_1, A_2, ..., A_n je důkazem formule A jestliže A_n je A a pro libovolné $i \in <1,n>$ je A_i buď axiómem, nebo bezprostředním důsledkem aplikace pravidla modus ponens na nějaké dva prvky z množiny $\{A_1, ..., A_{i-1}\}$.

Formule dokazatelné ve výrokové logice jsou právě tautologie a výroková logika proto tvoří bezesporný (korektní / konzistentní) formální systém.

Pozn.: Formální sytém je sporný, je-li v něm dokazatelná libovolná formule.

Následující dokazatelné formule jsou považovány za zákony výrokové logiky:

1.	zákon ekvivalence	$A \Leftrightarrow B$	$\Leftrightarrow$	$(A \Rightarrow B) \wedge (B \Rightarrow A)$
2.	zákon implikace	$A \Rightarrow B$	$\Leftrightarrow$	$\neg A \vee B$
3.	zákony komutativní	$A \vee B$	$\Leftrightarrow$	$B \vee A$
		$A \wedge B$	$\Leftrightarrow$	$B \wedge A$
4.	zákony asociativní	$A \wedge (B \wedge C)$	$\Leftrightarrow$	$(A \wedge B) \wedge C$
		$A \vee (B \vee C)$	$\Leftrightarrow$	$(A \vee B) \vee C$
5.	zákony distributivní	$A \vee (B \wedge C)$	$\Leftrightarrow$	$(A \vee B) \wedge (A \vee C)$
		$A \wedge (B \vee C)$	$\Leftrightarrow$	$(A \wedge B) \vee (A \wedge C)$
6.	zákony zjednodušení disjunkce			
		$A \vee T$	$\Leftrightarrow$	T
		$A \vee F$	$\Leftrightarrow$	A
		$A \vee A$	$\Leftrightarrow$	A
		$A \vee (A \wedge B)$	$\Leftrightarrow$	A
7.	zákony zjednodušení konjunkce			
		$A \wedge T$	$\Leftrightarrow$	A
		$A \wedge F$	$\Leftrightarrow$	F
		$A \wedge A$	$\Leftrightarrow$	A
		$A \wedge (A \vee B)$	$\Leftrightarrow$	A
8.	zákon vyloučeného třetího			
		$A \vee \neg A$	$\Leftrightarrow$	T
9.	zákon sporu	$\neg(A \vee \neg A)$	$\Leftrightarrow$	F
10.	zákon identity	A	$\Leftrightarrow$	A
11.	zákon negace	$\neg(\neg A)$	$\Leftrightarrow$	A
12.	zákony De Morganovy	$\neg(A \wedge B)$	$\Leftrightarrow$	$\neg A \vee \neg B$
		$\neg(A \vee B)$	$\Leftrightarrow$	$\neg A \wedge \neg B$
13.	eliminace AND	$A_1 \wedge A_2 \wedge ... \wedge A_n \Rightarrow A_i$		$i \in <1, n>$
14.	zavedení OR	$A_i \Rightarrow A_1 \vee A_2 \vee ... \vee A_n$		$i \in <1, n>$

Pomocí uvedených zákonů lze každou formuli, která není kontradikcí, převést do disjunktivní (DNF), resp. konjunktivní (KNF) normální formy takto: nejprve se eliminují spojky $\Leftrightarrow$ a $\Rightarrow$ (zákony 1 a 2), poté se přesunou spojky $\neg$ k atomům (zákony 11 a 12) a úprava se dokončí (pomocí zbývajících zákonů).

Pokud je formule G je logickým důsledkem formulí $\{F_1, F_2, \ldots F_n\}$, tj. pokud je formule $(F_1 \wedge F_2 \wedge, ..., \wedge F_n \Rightarrow G)$ tautologií, musí být negovaná formule, tj. formule $\neg(F_1 \wedge F_2 \wedge, ..., \wedge F_n \Rightarrow G)$, kontradikcí. S použitím zákona č. 2 lze psát

$$\neg(F_1 \wedge F_2 \wedge, ..., \wedge F_n \Rightarrow G) \Leftrightarrow (F_1 \wedge F_2 \wedge, ..., \wedge F_n \wedge \neg G)$$

a proto formule $(F_1 \wedge F_2 \wedge, ..., \wedge F_n \wedge \neg G)$ musí být také nesplnitelná. Zapíše-li

se předchozí formule ve tvaru množiny $\{F_1, F_2, ..., F_n, \neg G\}$, pak tato formule je nesplnitelná, právě když je nesplnitelná uvedená množina (tj. neexistuje žádné ohodnocení prvotních formulí vyskytujících se v této množině takové, aby všechny formule množiny byly pravdivé současně).

Důkaz nesplnitelnosti množiny $\{F_1, F_2, ..., F_n, \neg G\}$ je základem strojového dokazování a bude podrobněji popsán v části věnované rezoluční metodě.

4.3 Predikátová logika

Predikátová logika

Formální systém predikátové logiky prvního řádu (dále jen predikátová logika) tvoří opět tři složky:

- jazyk predikátové logiky (JPL),
- axiómy predikátové logiky:
  - axiómy výrokové logiky,
  - schéma specifikace $\forall x(A) \Rightarrow Ax[t]$ (Ax[t] je formule vzniklá substitucí termu t za proměnnou x ve formuli A),
  - schéma kvantifikace implikace $\forall x(A \Rightarrow B) \Rightarrow (A \Rightarrow \forall x(B))$ (proměnná x nesmí být ve formuli A volná),
- odvozovací pravidla:
  - modus ponens,
  - pravidlo generalizace: pro libovolnou proměnnou x odvoď z formule A formuli $\forall x(A)$.

Axiomy predikátové logiky a odvozovací pravidlo nebudeme dále používat, a stejně jako u výrokové logiky jsme je uvedli pouze pro informaci.

Jazyk predikátové logiky je bohatší nežli jazyk výrokové logiky a jeho základními složkami jsou:

- individuové konstanty: a, b, c, ...,
- individuové proměnné: x, y, z,...,
- funkční symboly: f, g, h, ...,
- predikátové symboly: P, Q, R, ...,
- logické spojky, stejné jako u výrokové logiky ($\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$),
- kvantifikátory (univerzální $\forall$a existenční $\exists$),
- pomocné symboly (kulaté závorky, hranaté závorky a čárka).

S každým funkčním a predikátovým symbolem je spojeno přirozené číslo, které udává počet jeho argumentů (míst).

Za term jazyka predikátové logiky se považuje individuová konstanta, individuová proměnná a struktura $f(t_1, t_2,..., t_n)$, kde f je n-místný funkční symbol a t_i jsou termy.

Atomickou formulí jazyka predikátové logiky je struktura $P(t_1, t_2, ..., t_n)$, kde P je n-místný predikátový symbol a t_i jsou termy. Za speciální případ atomických formulí se považují pravdivostní symboly T a F.

Atomickým formulím a jejich negacím se říká literály.

Formule jazyka predikátové logiky se vytvářejí takto:

- Každá atomická formule je formule,
- jsou-li A a B formule, pak i $\neg A$, $A \vee B$, $A \wedge B$, $A \Rightarrow B$ a $A \Leftrightarrow B$ jsou

formule – pro jejich vyhodnocení lze použít tabulku uvedenou u výrokové logiky,

- je-li x individuová proměnná a A formule, pak i $\forall x(A)$ a $\exists x(A)$ jsou formule.

Pozn.: Poslední pravidlo, tj. pravidlo pro tvorbu formulí kvantifikací proměnných, omezuje obecný jazyk predikátové logiky na jazyk predikátové logiky 1. řádu. Pro predikátové jazyky vyšších řádů, které povolují kvantifikovat i predikátové a funkční symboly, však dosud neexistují efektivní dokazovací metody.

Individuová proměnná je vázaná, pokud se vyskytuje v oblasti působnosti některého z kvantifikátorů, jinak je volná.

Kvantifikátor $\forall x$ znamená "pro každé x" a pravdivostní hodnota formule $\forall x(A)$ je proto dána konjunkcemi formulí A pro všechny možné hodnoty proměnné x.

Kvantifikátor $\exists x$ znamená "existuje x" a pravdivostní hodnota formule $\exists x(A)$ je proto dána disjunkcemi formulí A pro všechny možné hodnoty proměnné x.

Formule je otevřená, pokud neobsahuje žádnou vázanou proměnnou, resp. uzavřená, pokud neobsahuje žádnou volnou proměnnou.

Formule $\forall x_1 \forall x_2 ... \forall x_n(A)$ se nazývá univerzálním uzávěrem formule A, jsou-li všechny volné proměnné ve formuli A z množiny $\{x_1, x_2, ..., x_n\}$. Obdobně je definován i existenční uzávěr formule A.

Interpretací I formule A nad libovolnou neprázdnou množinou D (oborem interpretace - univerzem) se rozumí:

- přiřazení pevně zvoleného prvku z D každé individuové konstantě,
- přiřazení libovolného prvku z D každé individuové proměnné,
- přiřazení n-ární operace každému n-místnému funkčnímu symbolu ($D^n \rightarrow D$),
- přiřazení n-ární relace každému n-místnému predikátovému symbolu ($D^n \rightarrow \{T,F\}$).

Příklad: Důkaz, že formule $\forall x(P(a,x) \Rightarrow Q(f(x)))$ je pravdivá v interpretaci I:

Interpretace (například):

D	= {1,2,3}	a	= 2	f(x)	= 4-x
P(1,1)	= F	P(1,2)	= T	P(1,3)	= T
P(2,1)	= T	P(2,2)	= F	P(2,3)	= F
P(3,1)	= T	P(3,2)	= T	P(3,3)	= F
Q(1)	= T	Q(2)	= F	Q(3)	= T

Postup odvození důkazu:

$$\begin{aligned}
&\forall x(P(a,x) \Rightarrow Q(f(x))) && \Leftrightarrow \\
&(P(2,1) \Rightarrow Q(f(1))) \wedge (P(2,2) \Rightarrow Q(f(2))) \wedge (P(2,3) \Rightarrow Q(f(3))) && \Leftrightarrow \\
&(T \Rightarrow T) \wedge (F \Rightarrow F) \wedge (F \Rightarrow T) && \Leftrightarrow \\
&T \wedge T \wedge T && \Leftrightarrow \\
&T
\end{aligned}$$

Definice logicky pravdivých, logicky nepravdivých, splnitelných a nesplnitelných formulí jsou stejné jako ve výrokové logice. Rovněž stejné jsou definice klauzulí, ekvivalentních formulí, konjunktivní i disjunktivní normální formy a logických důsledků.

Formule A je v prenexní normální formě, je-li ve tvaru $A \equiv (Q_1x_1...Q_nx_n(M))$, kde Q_i je buď univerzální, nebo existenční kvantifikátor, x_i jsou navzájem různé proměnné a M je otevřená formule vzniklá z atomických formulí použitím logických spojek. Formuli M se pak říká jádro (matice) a posloupnosti kvantifikátorů $Q_1x_1...Q_nx_n$ prefix formule A.

Libovolnou formuli lze převést do prenexní normální formy pomocí zákonů výrokové logiky, doplněných schématy (zákony) pro práci s kvantifikátory:

15. přesun kvantifikátorů doleva (x se nevyskytuje v B!)

$$Qx(A) \vee B \quad \Leftrightarrow \quad Qx(A \vee B)$$
$$Qx(A) \wedge B \quad \Leftrightarrow \quad Qx(A \wedge B)$$

16. přesun negace dovnitř

$$\neg(\forall x(A)) \quad \Leftrightarrow \quad \exists x(\neg A))$$
$$\neg(\exists x(A)) \quad \Leftrightarrow \quad \forall x(\neg A))$$

17. distributivní zákony pro kvantifikátory

$$\forall x(A) \wedge \forall x(B) \quad \Leftrightarrow \quad \forall x(A \wedge B)$$
$$\exists x(A) \vee \exists x(B) \quad \Leftrightarrow \quad \exists x(A \vee B)$$

18. přejmenování vázaných proměnných

$$\forall x(A(x)) \vee \forall x(B(x)) \Leftrightarrow \quad \forall x \forall y(A(x) \vee B(y))$$
$$\exists x(A(x)) \wedge \exists x(B(x)) \Leftrightarrow \quad \exists x \exists y(A(x) \wedge B(y))$$

Pozn.: Symbol A(x) znamená, že se ve formuli A vyskytuje volná proměnná x, ap.

Pozn.: Poslední zákon říká, že se ve formuli B nahradí všechny výskyty proměnné x novou proměnnou y, která se v této formuli původně nevyskytovala!

Při převodu formule do prenexní normální formy se nejprve eliminují spojky $\Leftrightarrow$ a $\Rightarrow$ (zákony 1, 2), poté se přesouvají všechny negace těsně k atomům (zákony 11, 12), provede se nezbytné přejmenování proměnných a s použitím zákonů (15, 17, 18) se přesunou všechny kvantifikátory do prefixu formule. S použitím ostatních zákonů lze navíc převést matici formule do konjunktivní nebo disjunktivní normální formy.

Příklad: Převedení formule $\forall x \forall y(\exists z(A(x,z) \wedge B(y,z)) \Rightarrow \exists v(C(v,x,y)))$ do disjunktivní prenexní normální formy:

$$\forall x \forall y(\exists z(A(x,z) \wedge B(y,z)) \Rightarrow \exists v(C(v,x,y))) \quad \Leftrightarrow$$
$$\forall x \forall y(\neg \exists z(A(x,z) \wedge B(y,z)) \vee \exists v(C(v,x,y))) \Leftrightarrow$$
$$\forall x \forall y \forall z(\neg A(x,z) \vee \neg B(y,z) \vee \exists v(C(v,x,y))) \quad \Leftrightarrow$$
$$\forall x \forall y \forall z \exists v(\neg A(x,z) \vee \neg B(y,z) \vee C(v,x,y))$$

Odstranění existenčních kvantifikátorů ve formuli A, která je v prenexní normální formě a která má navíc matici M v konjunktivní normální formě, se provádí postupem zvaným skolemizace:

1. Nechť $\exists x_1$ je prvním kvantifikátorem v prefixu formule A zleva. Pak se vybere libovolná individuová konstanta c, tzv. Skolemova konstanta, která se dosud v matici M nevyskytuje, touto konstantou se nahradí všechny výskyty proměnné x_1 v matici M a kvantifikátor $\exists x_1$ se z prefixu formule odstraní.
2. Nechť $\exists x_{m+1}$ je první existenční kvantifikátor v prefixu formule A zleva a nechť se před ním (zleva) nachází m univerzálních kvantifikátorů. Pak se vybere libovolná m-místná funkce $f(x_1, ..., x_m)$, tzv. Skolemova funkce, která

dosud v matici M není, touto funkcí se nahradí všechny výskyty proměnné x_{m+1} v matici M a kvantifikátor $\exists x_{m+1}$ se z prefixu formule odstraní.

Po odstranění všech existenčních kvantifikátorů z prefixu formule získáme formuli v tzv. Skolemově normální formě, nebo krátce v normální formě. Díky pravidlu generalizace pak není nutné dokazovat formuli, ale pouze její matici.

Formule F v normální formě je vlastně konjunkcí klauzulí a proto se také říká, že je v klauzulárním tvaru nebo v klauzulární formě. Zapisuje se pak často ve tvaru množiny klauzulí S a zřejmě platí, že formule F je splnitelná právě tehdy, když je splnitelná množina klauzulí S a naopak.

x+y

Příklad: Převedení formule $\exists x \forall y \exists z(\neg A(x,y) \wedge (B(x,z) \vee C(x,y,z)))$ do Skolemovy normální formy:

$$\exists x \forall y \exists z(\neg A(x,y) \wedge (B(x,z) \vee C(x,y,z))) \quad \Leftrightarrow$$
$$\forall y \exists z(\neg A(a,y) \wedge (B(a,z) \vee C(a,y,z))) \quad \Leftrightarrow$$
$$\forall y(\neg A(a,y) \wedge (B(a,f(y)) \vee C(a,y,f(y))))$$

Odpovídající množina klauzulí: $S = \{\neg A(a,y), (B(a,f(y)) \vee C(a,y,f(y))\}$

Důkazem formule A je, podobně jako ve výrokové logice, konečná posloupnost formulí $A_1, A_2, ..., A_n$, jestliže A_n je A a pro libovolné $i \in \langle 1,n \rangle$ je A_i buď axiómem, nebo bezprostředním důsledkem aplikace odvozovacích pravidel na nějaké dva prvky z množiny $\{A_1, ..., A_{i-1}\}$. Přestože jde o úlohu podobnou hledání důkazu ve výrokové logice, nelze v případě predikátové logiky podobný důkaz jednoduše provést. Predikátové logika je obecně nerozhodnutelná, což jinými slovy znamená, že proces důkazu nepravdivé formule nemusí nikdy skončit (není možné dokázat logickou pravdivost, nebo nesplnitelnost formulí predikátové logiky ověřováním jejich interpretací, protože počet oborů těchto interpretací není konečný). Nedokáže-li se tedy pravdivost formule, nelze tento neúspěch nikdy považovat za důkaz její nepravdivosti, protože proces dokazování mohl být ukončen příliš brzy.

V běžné praxi se naštěstí nepotřebují dokazovat libovolné formule, ale formule jistým způsobem omezené (syntakticky i sémanticky), které jsou pak dokazatelné.

4.3 Rezoluční metoda

Rezoluční metoda

!

Rezoluční metoda je nejznámější metodou automatického dokazování. Umožňuje dokázat nesplnitelnost dané množiny klauzulí procesem postupného rozšiřování této množiny o odvozené klauzule (rezolventy). Daná množina klauzulí je nesplnitelná, podaří-li se odvodit prázdnou klauzuli □ (Robinsonův rezoluční princip, J.A.Robinson, 1965).

Pravidlo základní rezoluce je určeno k dokazování nesplnitelnosti množiny klauzulí, ve kterých se nevyskytují žádné proměnné, a je proto vhodné především pro výrokovou logiku:

Nechť C_1 a C_2 jsou klauzule (tzv. rodičovské klauzule) a nechť jedna z nich, např. C_1, obsahuje literál L a druhá klauzule (C_2) negaci tohoto literálu $\neg L$. Pak rezolventou klauzulí C_1 a C_2 je klauzule

$$C = (C_1 - L) \vee (C_2 - \neg L),$$

pro kterou platí, že je logickým důsledkem klauzulí C_1 a C_2. Důkaz je jednoduchý:

Nechť $C_1 = (A_1 \vee A_2 \vee A_n \vee L)$ a $C_2 = (B_1 \vee B_2 \vee B_m \vee \neg L)$.

Protože obě klauzule (disjunkce literálů) jsou pravdivé, pak je-li nepravdivý literál L musí být pravdivá klauzule $(A_1 \vee A_2 \vee A_n)$. Naopak, je-li nepravdivý literál $\neg L$ musí být pravdivá klauzule $(B_1 \vee B_2 \vee B_m)$. Z této jednoduché úvahy bezprostředně vyplývá, že disjunkce klauzulí $(A_1 \vee A_2 \vee A_n) \vee (B_1 \vee B_2 \vee B_m)$ musí být rovněž pravdivou klauzulí.

Pozn.: Pravidlo modus ponens je speciálním případem pravidla základní rezoluce: {A, (A ⇒ B)} lze pomocí zákona implikace upravit na {A, (¬A ∨ B)) a rezolventou těchto dvou klauzulí je literál B.

Příklad: Dokažte nesplnitelnost množiny klauzulí:

1) P(a,b)
2) ¬P(a,b) ∨ ¬Q ∨ R(c)
3) ¬R(c)
4) ¬T(a,c) ∨ Q
5) T(a,c)

Důkaz:

6)	¬Q ∨ R(c)	rezolventa z 1 a 2
7)	¬Q	rezolventa z 6 a 3
8)	¬T(a,c)	rezolventa z 7 a 4
9)	□	rezolventa z 8 a 5

Podařilo se odvodit prázdnou klauzuli a tím je důkaz proveden.

V případě množiny klauzulí s proměnnými je situace složitější. Nejprve se musí provést tzv. unifikace, která spočívá ve vhodné substituci termů za proměnné:

- substitucí σ je konečná množina dvojic$\{t_1/x_1, ..., t_n/x_n\}$, kde t_i jsou termy a x_i proměnné,
- instancí klauzule C je klauzule C_σ získaná z klauzule C náhradou každého výskytu proměnné x_i termem t_i ze substituce σ,
- unifikátorem λ množiny literálů $\{L_1, L_2, ...,L_k\}$ je substituce, pro kterou platí $L_{1\lambda} = L_{2\lambda} = ... = L_{k\lambda} = L_\lambda$.

Jestliže unifikátor λ existuje, nazývá se množina unifikovatelnou.

Je nutné poznamenat, že v řadě případů může být unifikace provedena více unifikátory. Proto se zavádí pojem nejobecnějšího unifikátoru, což je unifikátor, ze kterého jdou všechny jiné unifikátory dalšími substitucemi odvodit.

Například klauzule P(a,x) a P(y,z) lze unifikovat unifikátory {a/y, a/x, a/z} a {a/y, x/z} – je na první pohled zřejmé, že druhý unifikátor je nejobecnějším unifikátorem.

Následující algoritmus pro unifikaci dvou klauzulí (obecně dvou výrazů E_1 a E_2) předpokládá jejich zápis ve tvaru seznamů, například:

Klauzule:	Odpovídající seznam:
p(a, b)	(p a b)

p(f(a), g(X, Y))	(p (f a) (g X Y))
p(X) ∨ q(Y)	((p X) ∨ (q Y))

Tento algoritmus buď nalezne nejobecnější unifikátor obou výrazů, nebo zjistí, že dané výrazy nejsou unifikovatelné.

Procedura Unify: {unifikuje E_1 a E_2, vrací nejobecnější unifikátor, nebo Fail}

1. E_1 i E_2 jsou buď konstanty nebo prázdné seznamy (současně):
   - Jestliže $E_1 = E_2$, vrať { } (tj. prázdnou substituci),
     jinak vrať Fail (výrazy nelze unifikovat).
2. E_1 je proměnná:
   - Jestliže E_1 je proměnnou v E_2, vrať Fail,
     jinak vrať substituci $\{E_2/E_1\}$.
3. E_2 je proměnná:
   - Jestliže E_2 je proměnnou v E_1, vrať Fail,
     jinak vrať substituci $\{E_1/E_2\}$.
4. jinak:
   - 4.1. Nechť HE_1 je první prvek E_1 a HE_2 první prvek E_2,
   - 4.2. Volej proceduru Unify s argumenty HE_1 a HE_2,
   - 4.3. Vrátí-li procedura volaná v bodě 4.2 Fail, vrať také Fail, jinak pokračuj,
   - 4.4. Nechť $SUBST_1$ je vrácená substituce procedurou volanou v bodě 4.2.
   - 4.5. Nechť TE_1 je zbytek E1 (bez prvku HE_1) a TE_2 zbytek E_2 (bez prvku HE_2), s respektováním substituce $SUBST_1$,
   - 4.6. Volej proceduru Unify s argumenty TE_1 a TE_2,
   - 4.7. Vrátí-li procedura volaná v bodě 4.6 Fail, vrať také Fail, jinak pokračuj,
   - 4.8. Nechť $SUBST_2$ je vrácená substituce procedurou volanou v bodě 4.6.
   - 4.9. Vrať kompozici substitucí $SUBST_1$ a $SUBST_2$.

Existence unifikačního algoritmu umožňuje zobecnit rezoluční metodu uvedenou výše. Základní rezoluční princip zůstává v platnosti, avšak pravidlo pro odvozování rezolvent je pozměněno takto:

Nechť C_1 a C_2 jsou klauzule (množiny literálů). Nechť klauzule C_1 obsahuje unifikovatelnou podmnožinu literálů M_1, kterou unifikátor λ_1 unifikuje do jediného literálu L_1, a nechť klauzule C_2 obsahuje unifikovatelnou podmnožinu literálů M_2, kterou unifikátor λ_2 unifikuje do jediného literálu L_2 a nechť tyto literály tvoří komplementární pár ($L1 = \neg L2$). Pak rezolventou klauzulí C_1 a C_2 je klauzule

$$C = (C_{1\lambda 1} - M_{1\lambda 1}) \vee (C_{2\lambda 2} - M_{2\lambda 2}).$$

Rezoluční metoda zaručuje odvození prázdné klauzule z nesplnitelné množiny výchozích klauzulí (ve smyslu, že prázdnou klauzuli je možné odvodit). Závažnou otázkou však zůstává optimální výběr rodičovských klauzulí, neboť jinak může docházet k odvozování značného počtu zbytečných rezolvent. Tím se samozřejmě zvyšují nároky na paměť počítače a prodlužuje se i doba výpočtu. Je známo několik doporučovaných strategií, z nichž korektní a úplné jsou např.

strategie podpůrné množiny a lineární strategie.

Strategie podpůrné množiny je založena na předpokladu, že v každé výchozí množině klauzulí je možné nalézt bezespornou podmnožinu. Rezolventy z klauzulí této podmnožiny nemohou vést ke sporu a nejsou tak zbytečně odvozovány.

Lineární strategie používá při odvozování nové rezolventy vždy poslední odvozenou rezolventu. Praktické použití této strategie je ukázáno na následujících příkladech.

Příklad: Dokažte nesplnitelnost množiny klauzulí:

1) $\neg A(x) \vee B(x) \vee C(x,f(x))$
2) $\neg A(x) \vee B(x) \vee D(f(x))$
3) $E(a)$
4) $A(a)$
5) $\neg C(a,y) \vee E(y)$
6) $\neg E(x) \vee \neg B(x)$
7) $\neg E(x) \vee \neg D(x)$

Důkaz:

8)	$B(a) \vee C(a,f(a))$	rezolventa z 1 a 4, substituce {a/x}
9)	$C(a,f(a)) \vee \neg E(a)$	rezolventa z 8 a 6, substituce {a/x}
10)	$\neg E(a) \vee E(f(a))$	rezolventa z 9 a 5, substituce {f(a)/y}
11)	$E(f(a))$	rezolventa z 10 a 3
12)	$\neg D(f(a))$	rezolventa z 11 a 7, substituce {f(a)/x}
13)	$\neg A(a) \vee B(a)$	rezolventa z 12 a 2, substituce {a/x}
14)	$B(a)$	rezolventa z 13 a 4
15)	$\neg E(a)$	rezolventa z 14 a 6, substituce {a/x}
16)	□	rezolventa z 15 a 3

Podařilo se odvodit prázdnou klauzuli a tím je důkaz proveden.

Rekapitulace:

Pokud potřebujeme dokázat, že nějaká klauzule logicky vyplývá z množiny klauzulí jiných, doplníme tuto množinu o negaci dokazované klauzule a s použitím unifikace a rezoluce se snažíme z této rozšířené množiny odvodit prázdnou klauzuli. Pokud tuto klauzuli odvodíme, je důkaz proveden, jinak nelze učinit žádný relevantní závěr.

Příklad: Z následujících výroků:

1. Bohatí a zdraví lidé jsou šťastní.
2. Lidé, kteří sportují, jsou zdraví.
3. Jirka sportuje a je bohatý.
4. Šťastní lidé mají spokojený život.

odvoďte, že

5. Někdo má spokojený život (tj. existuje někdo, kdo má spokojený život).

Postup důkazu:

a) Nejprve se výše uvedené výroky zapíší ve formě logických formulí

1. $\forall x(\text{Bohatý}(x) \wedge \text{Zdravý}(x) \Rightarrow \text{Šťastný}(x))$
2. $\forall y(\text{Sportuje}(y) \Rightarrow \text{Zdravý}(y))$
3. $\text{Sportuje}(\text{jirka}) \wedge \text{Bohatý}(\text{jirka})$
4. $\forall z(\text{Šťastný}(z) \Rightarrow \text{Spokojený\_život}(z))$
5. $\exists v(\text{Spokojený\_život}(v))$

b) Poté se všechny výše uvedené formule převedou do množiny klauzulí – dokazovaná formule se však musí nejprve negovat.

1. $\forall x(\text{Bohatý}(x) \wedge \text{Zdravý}(x) \Rightarrow \text{Šťastný}(x)) \Leftrightarrow$
   $\neg\text{Bohatý}(x) \vee \neg\text{Zdravý}(x) \vee \text{Šťastný}(x)$ (1)
2. $\forall y(\text{Sportuje}(y) \Rightarrow \text{Zdravý}(y)) \Leftrightarrow$
   $\neg\text{Sportuje}(y) \vee \text{Zdravý}(y)$ (2)
3. $\text{Sportuje}(\text{jirka}) \wedge \text{Bohatý}(\text{jirka}) \Leftrightarrow$
   $\text{Sportuje}(\text{jirka})$ (3a)
   $\text{Bohatý}(\text{jirka})$ (3b)
4. $\forall z(\text{Šťastný}(z) \Rightarrow \text{Spokojený\_život}(z)) \Leftrightarrow$
   $\neg\text{Šťastný}(z) \vee \text{Spokojený\_život}(z)$ (4)
5. $\neg(\exists v(\text{Spokojený\_život}(v)) \Leftrightarrow$
   $\forall v(\neg\text{Spokojený\_život}(v)) \Leftrightarrow$
   $\neg\text{Spokojený\_život}(v)$ (5)

Výsledné klauzule jsou číslovány vpravo. Stojí za povšimnutí, že formule 3. není klauzulí (neobsahuje disjunkci literálů), ale dvojicí klauzulí (3a) a (3b)!

c) Vlastní důkaz:

6.	$\neg\text{Šťastný}(v)$	rezolventa z (5) a (4), substituce {v/z}
7.	$\neg\text{Bohatý}(v) \vee \neg\text{Zdravý}(v)$	rezolventa z 6 a (1), substituce {v/x}
8.	$\neg\text{Zdravý}(\text{jirka})$	rezolventa z 7 a (3b), substituce{jirka/v}
9.	$\neg\text{Sportuje}(\text{jirka})$	rezolventa z 8 a (2), substituce {jirka/y}
10.	□	rezolventa z 9 a (3a)

Podařilo se odvodit prázdnou klauzuli a tím je důkaz proveden.

x+y

Příklad: Z následujících výroků:

1. Každý má rodiče.
2. Prarodič je rodič rodiče.

odvoďte, že

3. Karel má prarodiče (i když již třeba nežijí)

Postup důkazu:

a) Nejprve se výše uvedené výroky zapíší ve formě logických formulí

1. $\forall x_1 \exists y_1(\text{Rodič}(x_1, y_1))$
2. $\forall x_2 \forall y_2 \forall z_2(\text{Rodič}(x_2, y_2) \wedge \text{Rodič}(y_2, z_2) \Rightarrow \text{Prarodič}(x_2, z_2))$
3. $\text{Prarodič}(\text{karel}, x_3)$

b) Pak se všechny výše uvedené formule převedou do množiny klauzulí, dokazovaná formule se neguje

1. $\forall x_1 \exists y_1(\text{Rodič}(x_1, y_1) \Leftrightarrow$
   $\forall x_1(\text{Rodič}(x_1, \text{přímý\_předek}(x_1)) \Leftrightarrow$
   $\text{Rodič}(x_1, \text{přímý\_předek}(x_1))$ (1)

2. $\forall x_2 \forall y_2 \forall z_2 (\text{Rodič}(x_2, y_2) \wedge \text{Rodič}(y_2, z_2) \Rightarrow \text{Prarodič}(x_2, z_2)) \quad \Leftrightarrow$
   $\forall x_2 \forall y_2 \forall z_2 (\neg\text{Rodič}(x_2, y_2) \vee \neg\text{Rodič}(y_2, z_2) \vee \text{Prarodič}(x_2, z_2)) \quad \Leftrightarrow$
   $\neg\text{Rodič}(x_2, y_2) \vee \neg\text{Rodič}(y_2, z_2) \vee \text{Prarodič}(x_2, z_2) \quad (2)$
3. $\neg\text{Prarodič}(\text{karel}, x_3)$

Stojí za povšimnutí, že Skolemova funkce, pomocí které byl odstraněn existenční kvantifikátor v klauzuli 1, má zřejmý význam bezprostředního předka (při vyhodnocení by vracela bezprostředního předka osoby, která by byla jejím argumentem) – proto byla funkce nazvána přímý_předek.

c) Vlastní důkaz:

4. $\neg\text{Rodič}(\text{karel}, y_2) \vee \neg\text{Rodič}(y_2, x_3)$ rezolventa z 3 a (2), substituce$\{\text{karel}/x_2, x_3/z_2\}$
5. $\neg\text{Rodič}(\text{přímý\_předek}(\text{karel}), x_3)$ rezolventa z 4 a (1), substituce $\{\text{karel}/x_1, \text{přímý\_předek}(\text{karel})/y_2\}$
6. $\square$ rezolventa z 5 a (1), substituce $\{\text{přímý\_předek}(\text{karel})/x_1, \text{přímý\_předek}(\text{přímý\_předek}(\text{karel}))/x_3\}$

Podařilo se odvodit prázdnou klauzuli a tím je důkaz proveden.

Úkol: Z následující databáze

- Kdo se neučil, neumí rezoluční metodu.
- Kdo do umí rezoluční metodu, získá body.
- Kdo získá body, složí zkoušku z IZU.
- Někdo složí zkoušku z IZU, přestože se neučil.

dokažte pomocí resoluční metody tvrzení:

- Někdo složí zkoušku z IZU, přestože neumí rezoluční metodu.

4.4 Hornovy klauzule

Hornovy klauzule

Závěrem této kapitoly se stručně zmíníme o Hornových klauzulích, které jsou základem jazyka PROLOG. Hornovy klauzule jsou klausule, která obsahují nejvíce jeden positivní literál, tj. klauzule, které mohou být zapsány buď takto

$$((L_1 \wedge L_2 \wedge, ..., \wedge L_n) \Rightarrow L) \quad \Leftrightarrow \quad (\neg L_1 \vee \neg L_2 \vee, ..., \vee \neg L_n \vee L)$$

nebo takto

$$((L_1 \wedge L_2 \wedge, ..., \wedge L_n) \Rightarrow \square) \quad \Leftrightarrow \quad (\neg L_1 \vee \neg L_2 \vee, ..., \vee \neg L_n)$$

kde $n \geq 0$.

První z uvedených klauzulí představuje pro $n = 0$ fakt (L) a pro $n > 0$ pravidlo.

Pomocí druhé klauzule se zadává cíl, který se PROLOG snaží splnit.

Dokazování nesplnitelnosti množiny Hornových klauzulí je výrazně jednodušší, než dokazování nesplnitelnosti množiny obecných formulí. Většinou se používá klasické prohledávání do hloubky se zpětným navracením (backtracking).

4.5 Shrnutí

Shrnutí

V této kapitole jsou popsány základní principy výrokové a predikátové logiky a princip rezoluční metody. Pro úspěšné absolvování předmětu je nutné znát význam pojmů *formule*, *atomická formule*, *logické spojky*, *interpretace formule*, *tautologie*, *kontradikce*, *literál*, *klauzule*, *logický důsledek*, *teorém*, *normální forma*, *unifikace*, *Skolemova normální forma*, *rezolventa*, *rezoluce*. Dále je nutné mít praktické dovednosti při převodech formulí do normální formy, unifikace proměnných a při důkazu nesplnitelnosti množiny klauzulí pomocí rezoluční metody.

4.6 Kontrolní otázky

Kontrolní otázky

1. Jaké jsou základní prvky jazyka výrokové logiky?
2. Jak se tvoří formule z atomických formulí?
3. Jaký je vztah mezi tautologií a kontradikcí?
4. Co se rozumí pod pojmem *interpretace formule*?
5. Co se rozumí pod pojmy *literál*, *klauzule*, *normální forma*?
6. Jak se převede formule do Skolemovy normální formy?
7. Co je *unifikace* a jak se provádí?
8. V čem spočívá princip rezoluční metody?

5 JAZYK PROLOG

Cílem této kapitoly je seznámit studenty s jazykem PROLOG a ukázat, jak výhodně lze v tomto jazyku programovat prohledávací metody z kapitoly 3.

10 hod

Postupně zde bude popsána syntax jazyka, postup při plnění cílů, některé důležité zabudované predikáty, postup při definici nových klauzulí, zvláště klauzulí pro abstraktní datové struktury (zásobník, frontu, množinu), a pro základní prohledávací algoritmy (BFS, DFS, A*).

5.1 Úvodní informace

Úvodní informace

PROLOG je považován za jeden ze dvou základních jazyků klasické UI (druhým je jazyk LISP). Jak již bylo zmíněno v úvodu této opory, základní principy jazyka PROLOG prezentoval v roce 1972 A. Colmerauer. Název PROLOG pak pochází z francouzského *Programmation en logique*, i když v literatuře se často uvádí i anglický původ tohoto názvu *Programming in Logic* (programování v logice, logické programování).

PROLOG patří mezi tzv. deklarativní programovací jazyky, ve kterých programátor pouze zadá aktuální znalosti a cíl výpočtu, přičemž postup, jakým se tento výpočet provádí, je ponechán zcela na interpretu jazyka (systému). Základními přístupy, které systém při výpočtu používá, jsou unifikace, rezoluce a backtracking.

5.2 Syntax jazyka

Syntax jazyka

Syntax jazyka PROLOG je poměrně jednoduchá:

atom:	posloupnost alfanumerických znaků začínající malým písmenem
konstanta:	číslo (integer / real) | atom
proměnná:	posloupnost alfanumerických znaků začínající velkým písmenem nebo znakem '_'
term:	konstanta | proměnná | seznam | řetěz | predikát
argumenty:	term [, argumenty]
seznam:	[ ] | [argumenty] | [argumenty | proměnná]
řetěz:	posloupnost znaků uzavřená apostrofy (nebo znaky $)
predikát:	jméno(argumenty) | [term] operátor [term] | jméno
jméno:	atom
operátor:	definovaná posloupnost znaků
predikáty:	[not] predikát [, predikáty] | [not] predikát [; predikáty]
pravidlo:	hlava :- tělo.
hlava:	predikát
tělo:	predikáty
fakt:	predikát.
klauzule:	fakt | pravidlo
databáze:	množina klauzulí
otázka/cíl:	predikáty.
poznámka:	% libovolný řetěz znaků na zbytku řádku

Pomocné symboly (kulaté a hranaté závorky, čárka, středník, tečka) nejsou ve předchozím výpisu syntaxe uvedeny samostatně, ale přímo na místech, kde se mohou používat. Mezery jsou nevýznamné.

Pozn.: Posloupnost znaků uzavřená mezerami je ekvivalentní seznamu ordinálních čísel těchto znaků.

5.3 Plnění cílů

Plnění cílů

PROLOG je interpretovaný jazyk, který se ohlásí symbolem '?-' a očekává zadání cíle (otázky):

?- cíl.

Zadání cíle se ukončí znakem CR (klávesa Enter), který je pokynem pro zahájení výpočtu - činnosti spočívající ve splnění cíle. Obvykle je prvním cílem načtení předem připravené databáze.

Pokud je již databáze v systému uložena, další cíle spočívají v odvozování nových znalostí ze znalostí uložených v databázi. PROLOG pak pro každý další cíl začíná systematicky prohledávat databázi od jejího počátku.

Při splnění cíle vypíše PROLOG hodnoty všech proměnných cíle (případné volné proměnné vypíše ve tvaru adresy _xxxx, kde x značí hexadecimální číslici) a nakonec napíše buď „yes“ (cíl splněn) nebo „–>“. Po výpisu „–>“ může uživatel buď ukončit plnění daného cíle (stiskem klávesy ENTER), nebo může vyvolat pokus o jiné splnění tohoto cíle zápisem středníku. Středník chápe PROLOG jako logický operátor „nebo“ a proto pokračuje v dalším prohledávání databáze od místa, ve kterém se při předchozím splnění cíle nacházel.

Pokud cíl nelze splnit, vypíše PROLOG symbol „no“ (cíl nesplněn).

Plnění zadaného cíle se řídí následujícími pravidly:

1. Cílem je predikát:
   PROLOG se snaží ztotožnit cíl s faktem, nebo s hlavou pravidla (hledá stejné jméno, stejný počet argumentů a stejné argumenty - volná proměnná se může vázat na libovolný term).
   Cíl je splněn, jestliže se ztotožní:
   a) s faktem
   b) s hlavou pravidla a současně lze splnit tělo pravidla (tělo pravidla je chápáno jako nový cíl).
2. Cílem je konjunkce predikátů:
   Jestliže je cílem konjunkce predikátů (predikáty oddělené čárkami), snaží se PROLOG postupně, zleva doprava, splnit tyto predikáty všechny. Každý predikát představuje nový cíl a proměnné se považují pro všechny predikáty cíle za globální (stejné proměnné ve všech predikátech musí být vázané na stejný term!). Při prvním pokusu o splnění každého cíle se databáze prochází od počátku. Pokud se cíl, např. C_i, nesplní, vrátí se PROLOG k předcházejícímu C_{i-1} a pokouší se tento splnit jiným způsobem (pokračuje v prohledávání databáze od aktuálního místa pro tento cíl). Pokud se cíl C_{i-1} splní, přechází PROLOG opět k plnění cíle C_i a databázi pro tento cíl začne procházet znovu od začátku. Pokud se cíl C_{i-1} nesplní, vrátí se PROLOG k předcházejícímu cíli C_{i-2}, neuspěje-li cíl C_{i-2}, vrátí se PROLOG k cíli C_{i-3} atd. Pokud jsou splněny všechny cíle je původní cíl splněn, jinak tento cíl neuspěje. Popsaný způsob plnění cílů

je typickou aplikací metody zpětného navracení (Backtracking).

3. Jestliže je cílem disjunkce predikátů (predikáty oddělené středníky), snaží se PROLOG postupně (zleva doprava) splnit některý z nich. Databázi prochází pro každý predikát od počátku a proměnné jsou pro každý predikát cíle lokální.

Je nutné si uvědomit, že nesplnění cíle neznamená jeho negaci, ale pouze skutečnost, že z dané databáze nelze zadaný cíl dokázat (viz následující příklad)!

5.4 Jednoduchá konverzace

Jednoduchá konverzace

Nechť v systému je následující databáze (každý fakt i pravidlo ukončuje tečka!):

```
zena(zuzana).                 % fakty o osobách:
zena(anna).
muz(jan).
muz(jiri).
muz(robert).
umi(Kazdy, pocitat).          % fakt o dovednosti (možná příliš optimistický)
ma_rad(anna, jogurt).         % fakty o tom, co kdo má rád
ma_rad(anna, tango).
ma_rad(jan, anna).
par(M, Z) :- muz(M), zena(Z).     % pravidlo pro „tvorbu“ párů
```

Příklad konverzace (každá otázka musí být ukončena tečkou!):

```
?- muz(robert).
yes
?- muz(karel).
no                      Karel je jistě muž, z dané databáze to však nelze dokázat!
?- par(jan,  anna).     mezery mezi argumenty jsou nevýznamné
yes
?- par(anna,jan).
no                      na pořadí argumentů samozřejmě záleží
?-muz(X).
X = jan ->;             symbol -> oznamuje možnost volby:
X = jiri ->;            středníkem se ptáme na další možné splnění cíle
X = robert ->;
no
?- muz(X).
X = jan ->              ENTER znamená konec - s odpovědí jsme spokojeni
yes
?- zena(Zena).          Zena je normální proměnná
Zena = zuzana ->
yes
?- par(X, Y).       Dále je ukázáno postupné plnění cíle se dvěma proměnnými
X = jan
Y = zuzana ->;
X = jan
Y = anna ->;
X = jiri
Y = zuzana ->;
X = jiri
```

```
Y = anna ->;
X = robert
Y = zuzana ->;
X = robert
Y = anna ->;
no
?- umi(X, Y).
X = _0038          proměnná X se naváže na proměnnou Kazdy  a je stále volná!
Y = pocitat ->
yes
?- ma_rad(X, Y), ma_rad(Y,tango).          konjunkce cílů (musí být splněny oba)
X = jan
Y = anna -> ;
no
?- ma_rad(X, Y); ma_rad(Y,tango).          disjunkce cílů (stačí splnit jeden)
X = anna
Y = jogurt ->;
X = anna
Y = tango ->;
X = jan
Y = anna ->;
X = _0038
Y = anna ->;
no
?-
```

5.5 Práce se seznamy

Práce se seznamy

Práce se seznamy by měla být zřejmá z následující přímé konverzace:

x+y

```
?- S = [abc, 1, X, b(c, d), [5,6,7], "hallo"].          PROLOG zobrazuje znaky
S = [abc,1, _004C,b(c,d),[5,6,7],[104,97,108,108,111]]  v uvozovkách seznamem
X = _004C ->                                             jejich ordinálních čísel
yes
?- [a,b,c] = [H | T].
H = a
T = [b,c] ->;
no
?- [a,b,c] = [H1,H2,H3|T].
H1 = a
H2 = b
H3 = c
T = [ ] ->
yes
?- [a,b,c] = [H1,H2,H3,H4 | T].
no
?- T = [b,c,d], S = [a|T].
T = [b,c,d]
S = [a,b,c,d] ->
yes
?- [a,b,c,d] = [_ | T].      samostatný znak '_'  reprezentuje anonymní proměnnou
```

```
T = [b,c,d] ->
yes
?- [a,b,c,d] = [H | _]
H = a ->;
no
?- [a,b,c,d] = [H | _A].
H = a
_A = [b,c,d] ->          _A je normální proměnná
yes
?-
```

5.6 Definice nových klauzulí

Definice nových klauzulí (prvků databáze):

- **pro prvek seznamu** (member(X,S))

```
                                  prvek X je prvkem seznamu S:
member(X,[X | _]).                - pokud je jeho prvním prvkem
member(X,[_ |T]) :- member(X,T).  - pokud je prvkem jeho těla
```

- **pro spojení dvou seznamů** (append(S1,S2,S3))

```
                                  seznam S3 je spojení seznamu S1 a
                                  seznamu S2:
append([ ], S, S).                - pokud je první seznam prázdný
append([H | T1], S2, [H | T3]) :- - jinak
                        append(T1,S2,T3).
```

Pozn.: Předchozí pravidlo říká: třetí seznam [H | T3], tj. seznam složený z hlavičky H a těla T3, je spojením prvního seznamu [H | T1], tj. seznamu složeného z hlavičky H a těla T1, a seznamu S2, jestliže seznam T3 je spojením seznamů T1 a S2.

- **pro předka osoby** (predek(X,Y))

```
                                  Y je předkem osoby X, jestliže Y je:
predek(X,Y) :- otec(X,Y).         - otcem X
predek(X,Y) :- matka(X,Y).        - matkou X
predek(X,Y) :- otec(X,Z), predek(Z,Y).  - předkem otce X
predek(X,Y) :- matka(X,Z), predek(Z,Y). - předkem matky X
```

Uvažujme databázi, která obsahuje všech 8 předchozích klauzulí - member, append a predek (dvou faktů a šesti pravidel), a následujících 8 faktů:

```
otec(jiri,kamil).
otec(kamil,robert).
otec(robert,jan).
matka(jiri,eva).
matka(kamil,jana).
matka(robert,marie).
matka(eva,anna).
otec(eva,pavel).
```

Pak možná konverzace je tato:

```
?- member(3,[1,2,3,4,5]).
yes
?- member(X,[a,b,c]).
```

```
X = a ->;
X = b ->;
X = c ->;
no
?- append([a,b,c],[e,f,g],S).
S = [a,b,c,e,f,g] ->
yes
?- append(S,[e,f,g],[1,2,e,f,g]).
S = [1,2] ->
yes
?- predek(jiri,X).
X = kamil ->;
X = eva ->;
X = robert ->;
X = jana ->;
X = jan ->;
X = marie ->;
X = pavel ->;
X = anna ->;
no
?- predek(eva,Y).
Y = pavel ->;
Y = anna ->;
no
?-
```

5.7 Některé zabudované predikáty

Některé zabudované predikáty

Pozn.: Dále označuje symbol +X vázanou proměnnou a symbol -X proměnnou, která musí být volná. Pokud není uvedeno jinak, níže uvedené predikáty při návratech neuspějí!

5.7.1 Predikáty pro práci s databází

Predikáty pro práci s databází (přidávání, přepis a rušení klauzulí):

assert(+K)	***přidá*** *klauzuli K na konec databáze*
asserta(+K)	***přidá*** *klauzuli K na začátek databáze*
retract(+K)	*odstraní první výskyt klauzule K z databáze*
consult(+F)	*čte klauzule ze souboru F (F je jméno souboru uzavřené v apostrofech) a* ***přidává*** *je do databáze*
reconsult(+F)	*čte klauzule ze souboru F (F je jméno souboru uzavřené v apostrofech) a* ***zařazuje*** *je do databáze*
listing	*vypíše aktuální databázi na standardní výstupní zařízení*

Pozn.: Přidání klauzule znamená, že se tato klauzule k aktuální databázi skutečně přidá, tj. pokud v databázi již byla klauzule se stejným jménem, pak tato klauzule v databázi zůstane. Tato skutečnost je velmi „nebezpečná“ a může vést k chybným výpočtům! Zařazení klauzule do databáze znamená, že pokud již byla v databázi klauzule se stejným jménem, pak se tato klauzule odstraní (přepíše se novou klauzulí).

Příklady použití výše uvedených predikátů (nechť soubor d:\prolog\pokus.ari obsahuje dvě klauzule (fakty): pes(rek). pes(alik).):

```
?- consult('d:\prolog\pokus.ari').      přidá obsah souboru do databáze
yes
```

```
?- listing.
pes(rek).
pes(alik).
yes.
?- reconsult('d:\prolog\pokus.ari').      zařadí obsah souboru do databáze
yes
?- listing.
pes(rek).
pes(alik).
yes.
?- consult('d:\prolog\pokus.ari').        přidá obsah souboru do databáze
yes
?- listing.
pes(rek).
pes(alik).
pes(rek).
pes(alik).
yes.
?- assert(pes(zeryk)).                    přidá klauzuli na konec databáze
yes
?- asserta(pes(alan)).                    přidá klauzuli na začátek databáze
yes
?- retract(pes(alik)).                    odstraní klauzuli z databáze
yes
?- retract(pes(P)).
P = alan ->;
P = rek ->;
P = rek ->;
P = alik - >;
P = zeryk ->;
no
?-
```

Pozn.: Pokud je klauzulí pravidlo, musí být tato klauzule v predikátech assert, resp. retract uzavřena do kulatých závorek (v praxi se uvedené predikáty používají pouze pro práci s fakty, vkládání klauzulí je zcela výjimečné).

5.7.2 Predikáty pro V/V operace

Predikáty pro vstupní a výstupní operace (čtení a výpis termů):

see(+F)	*otevře soubor F pro čtení (F je jméno souboru uzavřené v apostrofech)*
read(-T)	*čte term T ze souboru otevřeného pro čtení (standardně z klávesnice - čtený term musí být ukončen tečkou!)*
seen	*uzavírá soubor, který je otevřen pro čtení*
tell(+F)	*otevře soubor F pro zápis (F je jméno souboru uzavřené v apostrofech)*
write(+T)	*zapíše term T do souboru otevřeného pro zápis (standardně na obrazovku)*
nl	*přejde na nový řádek*
told	*uzavírá soubor, který je otevřen pro zápis*

Příklady použití výše uvedených predikátů (nechť obsah souboru bbb.txt je: hallo. [1,2,3]. a(b,c).):

```
?- tell(aaa),write(abc),write('.'),write([1,2,3]),nl,write(a(b,c)),told.
yes
?- see('bbb.txt'),read(X),read(Y),read(Z),read(V),read(W),seen.
X = hallo
Y = [1,2,3]
Z = a(b,c)
V = end_of_file
W = end_of_file ->
yes
?
```

Obsah souboru aaa po výše uvedené konverzaci je:
abc.[1,2,3]
a(b,c)

Pozn.: Při čtení termů ze souboru musí být každý term ukončen tečkou a mezi jednotlivými termy musí být oddělovače – mezery, nebo nové řádky. Pokus o čtení termů ze souboru aaa by proto způsobil chybu!

5.7.3 Predikáty – operátory

Predikáty – operátory:

Operátor pro vyhodnocení aritmetického výrazu (vyhodnotí výraz s operátory: + , – , * , / , // , mod (sečítání, odečítání, násobení, dělení a celočíselné dělení)):

is

Příklady:

?- X is (15 – 6) * 3.	*vyhodnotí výraz a jeho hodnotu přiřadí volné proměnné X*
X = 27 ->	
yes	
?- 25 is 50 / 2.	*výsledkem vyhodnocení výrazu je reálné číslo 25.0*
no	
?- 25 is 50 // 2.	*výsledkem vyhodnocení výrazu je celé číslo 25*
yes	
?-	

Operátory pro porovnání hodnot výrazů V1 a V2, porovnání výrazů musí být cílem - oba výrazy se nejprve vyhodnotí a pokud je relace splněna, pak cíl uspěje, jinak neuspěje.

V1 < V2	*menší než*
V1 =< V2	*menší nebo rovno*
V1 > V2	*větší než*
V1 >= V2	*větší nebo rovno*
V1 =:= V2	*rovno*
V1 =\= V2	*nerovno*

Příklady:

?- 4 > 3.	*cíl splněn*
yes	
?- 3 > 4.	*cíl nesplněn*
no	
?- S = 3 > 4.	*cíl splněn (operátor '=' navázal volnou proměnnou S na term*

```
S = 3 > 4 ->                 3 > 4, podrobněji viz dále
yes
?- 3 + 4 =:= 9 – 2.
yes
?-
```

Operátory pro porovnání (unifikaci) termů:

T1 = T2	*uspěje, jde-li termy ztotožnit (volné proměnné se naváží)*
T1 = = T2	*uspěje, jsou-li termy stejné*
T1 \= T2	*uspěje, pokud termy nelze ztotožnit*
T1 \= = T2	*uspěje, pokud termy nejsou stejné*

Příklady:

?- L = [a,b,c], L = = X.	*proměnná L je vázána ne seznam, proměnná L je volná,*
no	*tj. termy (proměnné) nejsou stejné*
?- L = [a,b,c], L = X, L = = X.	*proměnná X byla navázána na proměnnou L, tj.termy*
L = [a,b,c]	*(proměnné) jsou stejné*
X = [a,b,c] ->	
yes	
?-	

5.7.4 Některé další užitečné predikáty

Některé další užitečné predikáty:

!	*tzv. řez (cut); uspěje, ale při návratu neuspěje, tj. nepovolí návrat*
repeat	*vždy uspěje (i při návratu)*
fail	*nikdy neuspěje*
bagof(T,+C,-S)	*vrací seznam S všech instancí termu T, pro které uspěje cíl C*
halt	*ukončí konverzaci*

Příklad - uvažujme stejnou databázi, jako v první ukázce konverzace, s odlišně definovaným pravidlem a výše uvedený soubor bbb.txt:

```
zena(zuzana).
zena(anna).
muz(jan).
muz(jiri).
muz(robert).
ma(karel,marii).
ma(honza,evu).
ma(karel,alika).
ma(jiri,zeryka).
ma(karel,nuz).
ma(honza,sirky).
ma(karel,zizen).
par(M, Z) :- muz(M), !, zena(Z).          % pravidlo používá řez (predikát !)
co_ma(X,S) :- bagof(Y, ma(X, Y), S).      % pravidlo používá predikát bagof
retract_all :- retract(X),fail.           % pravidlo používá predikát fail
retract_all.
```

Možná konverzace:

```
?- par(X,Y).
X = jan
Y = zuzana ->;
X = jan
Y = anna ->;                          řez nepovolí návrat na druhého muže
no
?- see('bbb.txt'), repeat, read(X), write(X), nl, X = end_of_file, seen.
hallo                                 ukázka použití predikátu repeat
[1,2,3]
a(b,c)
end_of_file
X = end_of_file ->
yes
?- co_ma(karel,S).
S = [marii,alika,nuz,zizen] ->
yes
?- halt.
```

Úkol: Definujte klauzule pro určení indexu maximálního prvku lineárního číselného seznamu, klauzule pro obrácení pořadí prvků (reverzi) lineárního seznamu, klauzule pro odstranění posledního prvku seznamu a klauzule pro stanovení počtu prvků obecného seznamu (t.j seznamu, jehož prvky mohou být i seznamy, jejímiž prvky mohou být další seznamy, atd.).

5.8 ADS — Abstraktní datové struktury (ADS) - zásobník, fronta, množina

```
empty([ ]).          vytvoření prázdného seznamu, nebo test na prázdný seznam
                     (stejné pro zásobník, frontu i množinu)
```

V následujících definicích klauzulí pro vložení a výběr prvku je prvním argumentem příslušného predikátu vkládaný, resp. vybíraný prvek (E - element), druhým argumentem aktuální seznam (zásobník, fronta, množina)) a třetím argumentem nový seznam po provedené operaci.

Zásobník (Stack)

```
push(E, T, [E | T]).                    vložení prvku
pop(E, [E | T], T).                     výběr prvku
```

Fronta (Queue)

```
enqueue(E, [ ], [E]).                   vložení prvku
enqueue(E, [H | T2], [H | T3]) :- enqueue(E, T2, T3).
dequeue(E, [E | T], T).                 výběr prvku
```

Fronta s prioritou (Priority Queue)

```
insert(E, [ ], [E]).                    vložení prvku
insert(E, [H | T], [E,H | T]) :- precedes(E, H).
insert(E, [H | T2], [H | T3]) :- precedes(H, E), insert(E, T2, T3).
precedes(A, B) :- A < B.                platí pouze pro čísla!
dequeue(E, [E | T], T).                 výběr prvku
```

Množina (Set)

```
add(E, S, S) :- member(E, S), !.                    vložení prvku
add(E, S, [E | S]).

delete(E, [ ], [ ]).                                výběr prvku
delete(E, [E | T], T) :- !.
delete(E, [H | T2], [H | T3]) :- delete(E, T2, T3).
```

V následujících klauzulích pro operace se dvěma množinami (průnik, sjednocení, rozdíl) jsou prvními dvěma argumenty predikátů původní množiny a třetím argumentem je výsledná množina.

```
intersection([ ], _, [ ]).                          průnik dvou množin
intersection([H | T1], S2, [H | T3]) :- member(H, S2), !, intersection(T1, S2, T3).
intersection([H | T1], S2, S3) :- intersection(T1, S2, S3).

union([ ], S, S).                                   sjednocení dvou množin
union([H | T], S2, S3) :- union(T, S2, S4), add(H, S4, S3).

difference([ ], _ , [ ]).                           rozdíl dvou množin
difference([H | T ], S2, S3) :- member(H, S2), difference(T, S2, S3).
difference([H | T1 ], S, [H | T3]) :- not(member(H, S)), difference(T1, S, T3).
```

V následujících klauzulích pro porovnání dvou množin (podmnožina, rovnost) jsou argumenty příslušných predikátů obě původní množiny.

```
subset([ ], _).                  testuje, zda první množina je podmnožinou druhé
subset([H | T], S) :- member(H, S), subset(T, S).

equal(S1, S2) :- subset(S1, S2), subset(S2, S1).    testuje rovnost množin
```

Ukázka konverzace

Předpokládejme, že všechny výše uvedené klauzule jsou v databázi. Pak jejich použití si můžeme snadno ověřit, doplníme-li databázi o následující klauzule (jde již o „program“, který se spustí zadáním cíle *do*.):

```
do :- remove_list, repeat,nl, write('Zadejte prikaz (ukonceny teckou!)'),nl,
      write('(new empty push pop enqueue insert dequeue add delete end): '),
      read(X),command(X).

remove_list:-retract(list(_)).
remove_list.

error_nl:- write('Chyba - zadny seznam v databazi neexistuje!'),nl,fail.
error_el:- write('Chyba - seznam je prazdny!'),nl,fail.
msg_e(E):- write('Vybrany prvek: '),write(E),nl.
msg_l(L):- write('Novy seznam:   '),write(L),nl.

command(new):- remove_list,empty(L),assert(list(L)),msg_l(L),!,fail.

command(empty):- list(L),!,com_empty(L).
command(empty):- error_nl.

com_empty(L):- empty(L),write('Seznam je prazdny.'),nl,!,fail.
com_empty(L):- write('Seznam neni prazdny.'),nl,fail.

command(push):- list(L),write('Zadejte prvek: '),read(E),retract(list(S)),
```

```
                    push(E,S,Snew),assert(list(Snew)),msg_l(Snew),!,fail.
command(push):- error_nl.

command(pop):- list(L),!,com_pop(L).
command(pop):- error_nl.

com_pop(L):- empty(L),!,error_el.
com_pop(L):- retract(list(S)),pop(E,S,Snew),assert(list(Snew)),msg_e(E),
             msg_l(Snew),!,fail.

command(enqueue):- list(L),write('Zadejte prvek: '),read(E),retract(list(Q)),
                   enqueue(E,Q,Qnew),assert(list(Qnew)),msg_l(Qnew),!,
                   fail.
command(enqueue):- error_nl.

command(insert):- list(L),write('Zadejte prvek: '),read(E),retract(list(Q)),
                  insert(E,Q,Qnew),assert(list(Qnew)),msg_l(Qnew),!,fail.
command(insert):- error_nl.

command(dequeue):- list(L),!,com_deq(L).
command(dequeue):- error_nl.

com_deq(L):- empty(L),!,error_el.
com_deq(L):- retract(list(Q)),dequeue(E,Q,Qnew),assert(list(Qnew)),msg_e(E),
             msg_l(Qnew),!,fail.

command(add):- list(L),write('Zadejte prvek: '),read(E),retract(list(S)),
               add(E,S,Snew), assert(list(Snew)),msg_l(Snew),!,fail.
command(add):- error_nl.

command(delete):- list(L),write('Zadejte prvek: '),read(E),retract(list(S)),
                  delete(E,S,Snew),assert(list(Snew)),msg_l(Snew),!,fail.
command(delete):- error_nl.

command(end).

member(H,[H|_]).
member(E,[H|T]):-member(E,T).
```

Základem předchozího programu je predikát *list(L)*, který obsahuje seznam *L* (chápaný podle uvažovaných ADS buď jako zásobník, nebo jako fronta, nebo jako množina). Předchozí klauzule jsou poměrně jednoduché a s odkazem na následující příklad konverzace předpokládáme, že nepotřebují podrobnější vysvětlení. Všechny klauzule mohou pracovat s libovolnými argumenty ADS, pouze klauzule *insert* pracují výhradně s čísly, která vkládají do uspořádaných číselných seznamů (jinak by se musely předefinovat klauzule *precedes*).

```
?- do.
Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): push.
Chyba - zadny seznam v databazi neexistuje!

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): new.
Novy seznam:   []
```

```
Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): push.
Zadejte prvek: 4.
Novy seznam:   [4]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): push.
Zadejte prvek: 1.
Novy seznam:   [1,4]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): enqueue.
Zadejte prvek: 8.
Novy seznam:   [1,4,8]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): insert.
Zadejte prvek: 6.
Novy seznam:   [1,4,6,8]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): pop.
Vybrany prvek: 1
Novy seznam:   [4,6,8]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): dequeue.
Vybrany prvek: 4
Novy seznam:   [6,8]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): empty.
Seznam neni prazdny.

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): add.
Zadejte prvek: 6.
Novy seznam:   [6,8]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): add.
Zadejte prvek: a.
Novy seznam:   [a,6,8]

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): new.
Novy seznam:   []

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): empty.
```

```
Seznam je prazdny.

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): pop.
Chyba - seznam je prazdny!

Zadejte prikaz (ukonceny teckou!)
(new empty push pop enqueue insert dequeue add delete end): end.
yes
?- intersection([1,2,3,[a,b],c],[2,4,6,[a,b],d,e],L).
L = [2,[a,b]] ->;
no
?- A=[1,2,3,[a,b],c],B=[2,4,6,[a,b],d,e],union(A,B,C).
A = [1,2,3,[a,b],c]
B = [2,4,6,[a,b],d,e]
C = [1,3,c,2,4,6,[a,b],d,e] ->
yes
?- subset([d,a,f],[a,b,c,d,e,f,g]).
yes
?- subset([a,d,h],[a,b,c,d,e,f,g]).
no
?- equal([a,b,c],[b,c,a]).
yes
?- halt.
```

5.9 Základní prohledávací algoritmy

Základní prohledávací algoritmy

Se znalostmi získanými z předchozího textu již lze porozumět níže uvedeným programům implementujícím základní prohledávací algoritmy BFS a DFS a A*.

5.9.1 BFS

BFS (se seznamem CLOSED) - úloha dvou džbánů:

Prvky/argumenty seznamů *Open a Closed* jsou dvouprvkové seznamy – prvním prvkem je daný uzel a druhým prvkem jeho bezprostřední předchůdce, např.: [[3,0],[0,0]], nebo [[0,0],nil]]. Symbolem *Start* je označen počáteční uzel, seznam *Goals* označuje seznam cílů.

```
% část programu nezávislá na konkrétní úloze:
bfs(Start,Goals):- path([[Start,nil]],[ ],Goals).                    % 1

path([ ],_,_):- write('No solution was found.'),!.                   % 2
path(Open,Closed,Goals):- dequeue([State|Parent],Open,_),            % 2a
        member(State,Goals),write('Solution has been found! '),
        write('The path is:'),nl,print_solution([State|Parent],Closed).
path(Open,Closed,Goals):- dequeue([State|Parent],Open,Rest_open),    % 2b
        get_children(State,Rest_open,Closed,Children),
        append(Rest_open,Children,New_open),
        append([[State|Parent]],Closed,New_closed),
        path(New_open,New_closed,Goals).

dequeue(E,[E|T],T).                                                  % 3
```

```
get_children(State,Rest_open,Closed,Children):-                    % 4
        bagof(Child,moves(State,Rest_open,Closed,Child),Children).
get_children(State,Rest_open,Closed,[]).                           % 4a

moves(State,Rest_open,Closed,[Next,State]):- move(State,Next),     % 5
        not member([Next,_],Rest_open), not member([Next,_],Closed).

member(H,[H|_]).                                                   % 6
member(H,[_|T]):- member(H,T).

append([],S,S).                                                    % 7
append([H|T1],T2,[H|T3]):- append(T1,T2,T3).

print_solution([State,nil],_):- write(State),nl.                   % 8
print_solution([State,Parent],Closed):-
        member([Parent,Grandparent],Closed),
        print_solution([Parent,Grandparent],Closed), write(State),nl.

% část závislá na konkrétní úloze, úloha dvou džbánů (4 litry, 3 litry):
move([V,M],[4,M]):- V<4.                                           % 9
move([V,M],[V,3]):- M<3.
move([V,M],[0,M]):- V>0.
move([V,M],[V,0]):- M>0.
move([V,M],[0,MN]):- V>0,M<3,3-M>=V,MN is M+V.
move([V,M],[VN,3]):- V>0,M<3,3-M<V,VN is V-3+M.
move([V,M],[VN,0]):- M>0,V<4,4-V>=M,VN is M+V.
move([V,M],[4,MN]):- M>0,V<4,4-V<M,MN is M-4+V.
```

Program se spustí zadáním predikátu/cíle *bfs*, jehož prvním argumentem bude počáteční stav a druhým argumentem seznam cílových stavů úlohy, např.

?- bfs([0,0],[[0,2],[2,0]]).

Tento cíl bude splněn, bude-li splněn cíl/predikát *path(Open,Closed,Goals)* pro počáteční stav úlohy *[Start,nil]* uložený v seznamu *Open* (klauzule 1).

Je-li (později, v průběhu výpočtu) seznam *Open* prázdný, řešení končí neúspěchem a vypíše se zpráva *No solution was found.* (klauzule 2).

Obsahuje-li první prvek *[State|Parent]* zleva v seznamu *Open* uzel cílový (zjistí se pomocí predikátů *dequeue([State|Parent],Open,_),member(State,Goals)*), vypíše se zpráva *Solution has been found! The path is:* a pomocí predikátu *print_solution([State|Parent],Closed)* i nalezená cesta (klauzule 2a).

Jinak se vybere první prvek *[State|Parent]* zleva ze seznamu *Open* (opět pomocí predikátů *dequeue([State|Parent],Open,Rest_open)*), pro uzel/stav *State* se určí všichni jeho bezprostřední následníci, kteří dosud nejsou v seznamech *Open*, nebo *Closed* (pomocí predikátu *get_children(State,Rest_open,Closed,Children)*), všichni tito následníci se připojí k seznamu *Open* zprava (pomocí predikátu *append(Rest_open,Children,New_open)*), expandovaný uzel se uloží pomocí predikátu *append([[State|Parent]],Closed,New_closed)* do seznamu *Closed* a nový cíl/predikát *path(New_open,New_closed,Goals)* se volá rekurzivně pro nové seznamy *Open* a *Closed* (klauzule 2b). Poznamenejme, že seznam

Children obsahuje dvouprvkové seznamy *[potomek$_i$,State]*.

Klauzule 3, 6 a 7 již byly vysvětleny dříve.

Klauzule 4 *get_children(State,Rest_open,Closed,Children)* provede expanzi uzlu *State* a všechny jeho bezprostřední následníky, kteří nejsou v seznamech *Rest_open* nebo *Closed*, vrátí v seznamu *Children (ve tvaru zmíněném výše)*. To zajistí predikát *bagof(Child,moves(State,Rest_open,Closed,Child),Children)*. Pokud žádný bezprostřední následník expandovaného uzlu neexistuje, vrátí klauzule 4a prázdný seznam (klauzule *get_children(State,Rest_open,Closed,[]*).

Splnění predikátu *bagof* v klauzuli 4 závisí na splnění jeho argumentu/predikátu *moves*. Klauzule *moves(State,Rest_open,Closed,[Next,State])* (klauzule 5) hledá možné tahy (pridikát *move(State,Next)*) a tyto stavy akceptuje, pokud dosud nejsou ani v seznamu *Open* (predikát *not member([Next,_],Rest_open)*), ani v seznamu *Closed* (predikát *not member([Next,_],Closed)*).

Klauzule 8 (*print_solution([State,Parent],Closed)*) zajistí výpis jednotlivých stavů nalezené cesty (predikáty *write(State),nl*) postupným hledáním předků těchto stavů v seznamu *Closed* (*member([Parent,Grandparent],Closed)*). Prohledávání je ukončeno nalezením počátečního stavu *[State,nil]* – výpis se provádí v opačném pořadí, tzn., že poslední nalezený stav se vypíše jako první a cesta se tak vypisuje od počátku k cíli.

Klauzule 9 popisují danou úlohu popisem možných, resp. přípustných tahů *move(Odkud,Kam)*. První klauzule *move([V,M],[4,M]):- V<4.* popisuje plnění velkého džbánu: pokud je původní obsah velkého džbánu $V < 4$ (džbán není plný), bude jeho nový obsah 4 a obsah malého džbánu *M* se přitom nezmění. Výklad dalších klauzulí *move(Odkud,Kam)* je stejně jednoduchý.

Úkol: Definujte klauzule *move(Odkud,Kam)* pro jiné velikosti džbánů. Pokuste se definovat obdobné tahy i pro některé jiné úlohy.

5.9.2 DFS

DFS (se seznamem CLOSED) - úloha dvou džbánů:

Program implementující algoritmus DFS se získá z předchozího programu:

- záměnou predikátu *pop* za *dequeque* (jde o formální záměnu, protože činnost je naprosto stejná) v klauzulích 2a, 2b a 3.
- záměnou pořadí připojení následníků expandovaného uzlu do seznamu *Open* v klauzuli 2b. Tato záměna se provede přehozením argumentů v predikátu append – místo

  append(Rest_open,Children,New_open),

  se použije

  append(Children,Rest_open,New_open).

Úkol: Napište program (klauzule) pro algoritmus DFS bez seznamu *Closed*, s eliminací stejných stavů v *Open* a s eliminací následníků, kteří jsou již předchůdci uzlu. Nápověda: Uzly musí nést veškerou informaci o svých předchůdcích (například [[3,3],[3,0],[0,3],[0,0]]).

5.9.3 A*

Algoritmus A* - úloha „hlavolam 8“:

x+y

```
% každý stav je popsán seznamem [Node,Parent,G,H,F], F = G+H.

% část nezávislá na konkrétní úloze
a_star(Start,Goal):- estimate(Start,Goal,0,_,H),                    % 1
        path([[Start,nil,0,H,H]],[],Goal).
path([ ],_,_):-  write('No solution was found.'),!.                 % 2
path(Open,Closed,Goal):- dequeue([Goal,Parent|_],Open,_),           % 2a
        write('Solution was found. The path is:'),nl,
        print_solution([Goal,Parent|_],Closed).
path(Open,Closed,Goal):- dequeue([State,Parent,G,H,F],Open,Rest_open), % 2b
        get_children(State,G,F,Rest_open,Closed,Goal,[],Children),
        insert_to_open(Rest_open,Children,New_open),
        append([[State,Parent,G,H,F]],Closed,New_closed),
        path(New_open,New_closed,Goal).

dequeue(E,[E|T],T).                                                 % 3

get_children(State,G,F,Rest_open,Closed,Goal,Temp,Children):-       % 4
        move(State,Next),
        not(member([Next|_],Closed)),not(member([Next|_],Temp)),
        estimate(Next,Goal,G,Gnew,H),Ftemp is Gnew+H,
        max(F,Ftemp,Fnew),
        get_children(State,G,F,Rest_open,Closed,Goal,
        [[Next,State,Gnew,H,Fnew]|Temp],Children),!.
get_children(_,_,_,_,_,_,L,L).                                      % 4a

max(A,B,A):- A>=B.                                                  % 5
max(A,B,B):- A<B.

insert_to_open(Open,[],Open).                                       % 6
insert_to_open(Open,[H|T],Open_new):- insert(Open,H,Temp_open),
        insert_to_open(Temp_open,T,Open_new).

insert([],H,[H]).                                                   % 7
insert(Open,[Next,P,G,H,F],New_open):-member([Next,_,_,_,_],Open),!,
        insert_change(Open,[Next,P,G,H,F],New_open).
insert([[S1,P1,G1,H1,F1]|T1],[S2,P2,G2,H2,F2],
                [[S2,P2,G2,H2,F2],[S1,P1,G1,H1,F1]|T1]):- F1>=F2.
insert([[S1,P1,G1,H1,F1]|T1],[S2,P2,G2,H2,F2],[[S1,P1,G1,H1,F1]|T3]):-
        F1<F2,insert(T1,[S2,P2,G2,H2,F2],T3).

insert_change(Open,[Node,Pnew,Gnew,Hnew,Fnew],Open):-               % 8
        remove(Open,[Node,Pnew,Gnew,Hnew,Fnew],
                [Node,Pold,Gold,Hold,Fold],Open_temp),Fold=<Fnew.
insert_change(Open,[Node,Pnew,Gnew,Hnew,Fnew],Open_new):-
        remove(Open,[Node,Pnew,Gnew,Hnew,Fnew],
                [Node,Pold,Gold,Hold,Fold],Open_temp),Fold>Fnew,
        insert(Open_temp,[Node,Pnew,Gnew,Hnew,Fnew],Open_new).
```

```
remove([[Node,Pold,Gold,Hold,Fold]|T],[Node,_,_,_,_],[Node,Pold,Gold,Hold,
                Fold],T).                                          % 9
remove([H|T],[Node,_,_,_,_],[Node,Pold,Gold,Hold,Fold],[H|Ttemp]):-
        remove(T,[Node,_,_,_,_],[Node,Pold,Gold,Hold,Fold],Ttemp).

member(H,[H|_]).                                                   % 10
member(H,[_|T]):- member(H,T).

append([],S,S).                                                    % 11
append([H|T1],T2,[H|T3]):- append(T1,T2,T3).

print_solution([State,nil|_],_):- write1(State),nl.                % 12
print_solution([State,Parent|_],Closed):-
        member([Parent,Grandparent|_],Closed),
        print_solution([Parent,Grandparent|_],Closed),write1(State),nl.

% část závislá na konkrétní úloze (úloha „hlavolam N“)
estimate(State,Goal,G,Gnew,H):- size(N),NN is N*N,                 % 13
        evaluate(State,Goal,OK),H is NN-1-OK,Gnew is G+1.

evaluate([o],[o],0).                                               % 14
evaluate([A],[B],0):- A\=B.
evaluate([H],[H],1):- H\=o.
evaluate([o|TState],[o|TGoal],OK):- evaluate(TState,TGoal,OK).
evaluate([A|TState],[B|TGoal],OK):- A\=B,evaluate(TState,TGoal,OK).
evaluate([H|TState],[H|TGoal],OK):- H\=o,
        evaluate(TState,TGoal,TOK),OK is TOK+1.

write1([]).                                                        % 15
write1(L):-size(N),write2(L,N).

write2([H|T],M):-M>0,write(H),MM1 is M-1,write2(T,MM1).
write2(L,M):-nl,write1(L).

move(State,Next):- N=3,find_pos(State,SP),SP>N,NP is SP-N,         % 16
      change(State,NP,SP,Next).                  % tah nahoru
move(State,Next):- N=3,find_pos(State,SP),R is SP mod N,R\=0,NP is SP+1,
      change(State,NP,SP,Next).                  % tah doprava
move(State,Next):- N=3,find_pos(State,SP),NN is N*(N-1)+1,SP<NN,
        NP is SP+N,change(State,NP,SP,Next).     % tah dolů
move(State,Next):- N=3,find_pos(State,SP),R is SP mod N,R\=1,
        NP is SP-1,change(State,NP,SP,Next).     % tah doleva

find_pos([o|_],1).                                                 % 17
find_pos([H|T],SP):- H\=o,find_pos(T,SPM1),SP is SPM1+1.

change(State,NP,SP,Next):- move_sp(State,NP,Piece,Temp_State),     % 18
        move_piece(Temp_State,SP,Piece,Next).

move_sp([H|T],1,H,[o|T]).                                          % 19
move_sp([H|T],N,Piece,[H|TN]):- N>1,NM1 is N-1,
        move_sp(T,NM1,Piece,TN).
```

```
move_piece([_|T],1,H,[H|T]).                                    % 20
move_piece([H|T],N,Piece,[H|TN]):- N>1,NM1 is N-1,
        move_piece(T,NM1,Piece,TN).

size(3).          % nebo size(4) pro „hlavolam 15“, ap.          % 21
```

Program se spustí zadáním predikátu/cíle *a_star*, jehož prvním argumentem bude počáteční stav a druhým argumentem cílový stav úlohy, např.

?- a_star([1,4,2,7,8,3,o,6,5],[1,2,3,8,o,4,7,6,5]).

Pozn.: Kameny jsou v seznamech opět uváděny po řádcích, tzn., že zápis [1,4,2,7,8,3,o,6,5] označuje stav, kdy prázdné políčko je na pozici 7:

1	4	2
7	8	3
	6	5

Klauzule 1 nejprve ohodnotí počáteční stav úlohy (*f=g+h; g=0*, *f=h*) pomocí predikátu *estimate(Start,Goal,_,_,H)*, a poté se snaží nalézt řešení úlohy splněním predikátu *path([[Start,nil,0,H,H]],[],Goal)*. Poznamenejme, že tento predikát je formálně stejný jako u algoritmu BFS (*path(Open,Closed,Goal)*), má však pouze jediný (optimální) cíl a složitější zápis uzlu *[Node,Parent,G,H,F]*; proměnné G, H, F obsahují hodnoty ohodnocující funkce a obou jejích částí.

Klauzule 2, 2a a 2b jsou téměř stejné jako u algoritmu BFS a provádějí i stejnou činnost. Klauzule 2a je dokonce jednodušší díky jedinému cílovému stavu úlohy, v klauzuli 2b je predikát *append* (viz BFS) nahrazen predikátem *insert_to_open* (jde o frontu s prioritou!).

Klauzule 3, 10, 11 a 12 jsou stejné jako v metodě BFS.

Klauzule 4 (*get_children(State,G,F,Rest_open,Closed,Goal,Temp,Children)*) provede expanzi uzlu *State* a vrátí v seznamu *Children* všechny jeho bezprostřední následníky, kteří nejsou v seznamech *Rest_open* nebo *Closed*. Na rozdíl od BFS však nepoužívá predikát *bagof*, ale ukazuje, jak je možné činnost tohoto predikátu nahradit rekurzivním voláním. Používá jednodušší predikáty *not member* a pomocný seznam *Temp* (který je při prvním volání prázdný). Při každém volání hledá jediný nový stav, který dosud není v ani jediném seznamu (Open,Closed,Temp). Pokud takový stav nenalezne, končí svoji činnost (a pomocí predikátu 4a *get_children(_,_,_,_,_,_,L,L)* přepíše obsah seznamu *Temp* do seznamu *Children*). Pokud takový stav nalezne, provede nejprve jeho ohodnocení (pomocí predikátu *estimate(Next,Goal,G,Gnew,H)*, pak případně opraví hodnotu funkce *F* tak, aby tato funkce byla monotónní (pomocí predikátu *max(F,Ftemp,Fnew)*) a poté volá rekurzivně sebe sama s upraveným seznamem *Temp* (je do něj přidán nový stav):

get_children(State,G,F,Rest_open,Closed,Goal,
[[Next,State,Gnew,H,Fnew]|Temp],Children)

Klauzule 5 jsou jednoduché klauzule, které vrátí větší ze dvou čísel.

Klauzule 6 (*insert_to_open(Rest_open,Children,New_open)*) zařadí postupně všechny následníky expandovaného uzlu, vrácené v seznamu *Children* klauzulemi 4, do fronty *Open* (fronta s prioritou). K tomu používá predikát *insert(Open,H,Temp_open)*.

Klauzule 7 zařazují nový stav do prioritní fronty *Open*. První z těchto klauzulí zařazuje stav do prázdné fronty, třetí a čtvrtá pak do neprázdné fronty, ve které se zařazovaný stav nevyskytuje. Druhá klauzule zařazuje nový stav do fronty, ve které se již stejný stav nachází a může mít jak lepší, tak i horší ohodnocení. Správné zařazení zajišťuje predikát *insert_change(Open,New_state,New_open).*

Klauzule 8 plní nejprve predikát *remove(Open,New_state,Old_state,New_open)*, který pomocí klauzulí 9 odstraní z fronty *Open* stav *Old_state*, který je stejný, jako zařazovaný stav *New_state*, a vrací frontu bez tohoto stavu (*New_ open*). Pokud má nový stav horší ohodnocení než starý stav, pak první klauzule 8 vrací původní frontu *Open*, jinak pomocí druhé klauzule 8 zařadí do fronty *Open* nový stav (pomocí predikátu *insert*).

Klauzulí 13 začínají klauzule závislé na úloze. Klauzule 13 zjistí rozměr hracího pole (predikát *size(N)*), spočítá počet políček desky (*NN=N∗N*), zjistí počet kamenů na správných pozicích (ten vrátí predikát *evaluate* v proměnné *OK*), spočítá hodnotu heuristické funkce *H* (jde o funkci h_1 ze str. 34, tj. počet kamenů na nesprávných pozicích: *H is NN-1-OK*) a inkrementuje hodnotu G (jeden tah)

Klauzule 14 *evaluate(State,Goal,OK)* počítají počet kamenů na správných pozicích. Postupně se rekurzivně zanořují, až v seznamech *State* a *Goal* zbývají poslední políčka, pro která nastaví výchozí stav pro vrácenou hodnotu (1 pro stejné kameny, 0 pro různé kameny a prázdná políčka). Pak se postupně vynořují a pro stejné kameny inkrementují vrácenou hodnotu.

Klauzule 15 zajišťují výpis stavu ve zřejmém tvaru, např. pro rozměr 3 ve tvaru:

1	5	2
7	8	3
6		4

Klauzule 16 *move(State,Next)* realizují možné tahy: nejprve zjistí pozici prázdného políčka *SP* (pomocí predikátu *find_pos* definovaného klauzulemi 17), pak zjistí, zda je tah možný (například tah nahoru pro *N*=3 je možný pouze tehdy, je-li prázdné políčko na pozici větší než 3), vypočítá pozici kamene *NP*, se kterým si má vyměnit pozici prázdné políčko a pomocí predikátu *change(State,NP,SP,Next)* tah/záměnu provede.

Klauzule 18 *change(State,NP,SP,Next)* provedou záměnu kamene a prázdného políčka pomocí dvou predikátů: Predikát *move_sp(State,NP,Piece,Temp_State)* definovaný klauzulemi 19 posune prázdné políčko na pozici kamene, se kterým se má vyměnit. Číslo tohoto kamene vrátí v proměnné *Piece* a pomocný stav v proměnné *Temp_state*. Predikát *move_piece(Temp_State,SP,Piece,Next)*, definovaný klauzulemi 20, pak výměnu dokončí uložením kamene na původní pozici prázdného políčka.

Klauzule 17, 19 a 20 jsou jednoduché a nepotřebují bližšího vysvětlení.

Klauzule/fakt 21 definuje rozměr hracího pole.

Úkol: Proveďte nutné úpravy programu (klauzulí závislých na úloze) tak, aby řešil úlohu dvou džbánů s minimalizaci množství vody (buď vody ze zdroje, nebo vody, se kterou se manipuluje).

5.10 Shrnutí

Shrnutí

V kapitole byly vysvětleny principy jazyka PROLOG a bylo zde ukázáno, jak je možné v tomto jazyku implementovat prohledávací metody BFS, DFS a A^*. Z principů jazyka je důležité znát jeho syntax, postup interpretu při plnění zadaných cílů, a také některé zabudované predikáty. Je samozřejmě nutné umět definovat nové klauzule, tj. psát jednoduché programy v tomto jazyku.

5.11 Kontrolní otázky

Kontrolní otázky

1. Jaký je rozdíl mezi faktem a pravidlem?
2. Jak postupuje interpret jazyka PROLOG při plnění cílů?
3. Jak vypadá program v jazyku PROLOG?
4. Jaký operátor vyhodnocuje aritmetické výrazy?
5. Jaký je rozdíl mezi operátory '=' a ,'= ='?
6. Jaké znáte způsoby zápisu seznamů?

6 JAZYK LISP

Cílem této kapitoly je seznámit studenty s jazykem LISP a ukázat, jak lze v tomto jazyku programovat některé prohledávací metody z kapitoly 3.

10 hod

Postupně zde bude popsána syntax jazyka, postup při vyhodnocování S-výrazů, některé důležité zabudované funkce, postup při definici nových funkcí, zvláště funkcí pro abstraktní datové struktury (zásobník, frontu, množinu), a pro základní prohledávací algoritmy (DFS, BFS).

6.1 Úvodní informace

Úvodní informace

LISP je základním jazykem klasické UI a patří mezi nejstarší programovací jazyky vůbec. Jak bylo uvedeno v úvodu této opory, základní principy jazyka LISP prezentoval v roce 1958 J. McCarthy a tento jazyk, později doplněný o řadu užitečných funkcí, se stal nejpoužívanějším programovacím jazykem pro umělou inteligenci v USA.

Název LISP je zkratka vzniklá z anglických slov LISt Procesor, která říká, že jde o prostředek určený k práci se seznamy. LISP patří mezi tzv. funkcionální programovací jazyky a nerozlišuje data od výkonného kódu – vše se zapisuje ve formě symbolických výrazů, tzv. S-výrazů (*Symbolic expression, S-expression*).

6.2 Syntax jazyka

Syntax jazyka

Zjednodušená syntax jazyka LISP je následující:

atom:	číslo | řetěz | symbol
číslo:	integer | real (standardní notace)
řetěz:	“libovolná posloupnost znaků uzavřená uvozovkami“
symbol:	posloupnost znaků, která nepředstavuje standardní zápis čísla a která neobsahuje znaky ( ) ' ` , " ;
seznam:	(S-výrazy uzavřené v kulatých závorkách, které jsou od sebe oddělené mezerami nebo konci řádků) | NIL
S-výraz:	atom | seznam

Pozn.: Poznámka začíná středníkem a končí koncem řádku.
Pozn.: Velikost písmen je nevýznamná, ve výpisech používá LISP výhradně velká písmena (mimo písmen v řetězech).
Pozn.: Počet S-výrazů v seznamu je nevýznamný a v případě prázdného seznamu je nulový. Prázdný seznam má pak dva různé způsoby zápisu: ( ), nebo NIL.

6.3 Činnost interpretu

Činnost interpretu

LISP je podobně jako PROLOG interpretovaný jazyk. Po spuštění se ohlásí symbolem '>' a očekává zadání S-výrazu:

> S-výraz.

Činnost interpretu pro zadaný S-výraz pak lze popsat následovně:

Čtení S-výrazu → Vyhodnocení S-výrazu → Tisk hodnoty S-výrazu

Čtení a tisk jsou z hlediska programátora triviální operace, vyhodnocení daného S-výrazu probíhá takto:

1) Je-li S-výrazem číslo, pak hodnotou tohoto S-výrazu je stejné číslo.
2) Je-li S-výrazem řetěz, pak hodnotou tohoto S-výrazu je stejný řetěz.
3) Je-li S-výrazem symbol, pak hodnotou tohoto S-výrazu je S-výraz navázaný na tento symbol. Je-li tento symbol volný, ohlásí interpret LISPu chybu.
4) Je-li S-výrazem seznam, pak hodnotou tohoto S-výrazu je výsledná hodnota funkce určené prvním prvkem, která je aplikovaná na vyhodnocené ostatní prvky ((f x y z) ≡ f(x y z)).

Pozn.: Prvním prvkem vyhodnocovaného seznamu musí být symbol vázaný na nějakou funkci. V opačném případě ohlásí interpret LISPu chybu!

Hodnoty S-výrazů mohou být numerické, symbolické a pravdivostní. Pravdivostní hodnoty jsou reprezentovány symboly T a NIL.

Pozn.: Protože symbol NIL reprezentuje také prázdný seznam, je současně jak atomem, tak i seznamem!

6.4 Některé zabudované funkce

Některé zabudované funkce

Původní LISP uvažoval pouze pět základních funkcí: *atom*, *eq*, *car*, *cdr*, a *cons*, a dvě další funkce: *quote* a *cond*. Současné verze LISPu zahrnují desítky až stovky zabudovaných funkcí, které výrazně usnadňují programování v tomto jazyku.

6.4.1 Základní funkce

Základní funkce:

(atom x)	Vrací hodnotu T, pokud argumentem *x* je atom (číslo, řetěz, symbol). Jinak vrací hodnotu NIL.
(eq x y)	Vrací hodnotu T, pokud argumenty *x* a *y* jsou stejná čísla nebo stejné symboly. Jinak vrací hodnotu NIL.
(car x)	Vrací první prvek seznamu *x*. Není-li *x* seznamem je hlášena chyba.
(cdr x)	Vrací seznam *x* bez jeho prvního prvku. Je-li *x* jednoprvkovým nebo prázdným seznamem, vrací hodnotu NIL (prázdný seznam). Není-li *x* seznamem je hlášena chyba.
(cons x y)	Vrací seznam, jehož prvním prvkem je S-výraz *x* a jehož dalšími prvky jsou všechny prvky seznamu *y*. Není-li *y* seznamem, vytvoří se seznam obsahující tečku-dvojici (*x* **.** *y*), jejíž popis jde nad rámec předmětu.
(quote x)	Vrací S-výraz *x*. Funkce *quote* je tak často používaná, že jako jediná může používat zkrácený zápis: *(quote x)* ≡ *'x*
(cond (s_{11} s_{12}) (s_{21} s_{22}) … (s_{n1} s_{n2}))	Vyhodnocuje postupně S-výrazy s_{i1} (podmínky) a vrací hodnotu prvního S-výrazu s_{j2}, jehož podmínka s_{j1} je rozdílná od hodnoty NIL. Jinak vrací hodnotu NIL.

Příklady použití výše uvedených funkcí:

```
> 2                          ; hodnotou čísla je toto číslo
 2
 > nil                      ; hodnotou symbolu NIL je tento symbol
```

```
 NIL
> auto                                  ; symbol auto není vázaný
error: unbound variable - AUTO
> (quote  auto)                         ; funkce quote vrací nevyhodnocený argument
AUTO
> ‘auto                                 ; totéž
AUTO
> (atom  6)                             ; číslo je atomem
T
> (atom  “moto”)                        ; řetěz je atomem
T
> (atom  ‘a123)                         ; symbol je atomem
T
> (atom  ‘(a  b  c  d))                 ; seznam není atomem
NIL
> (atom  (a  b  c  d))                  ; LISP se nejprve snaží vyhodnotit funkce,
error: unbound function - A
> (atom  (1  b  c  d))                  ; které jsou dány jeho argumenty
error: bad function  - 1
> (car  ‘(a  b  c  d))                  ; vrací první prvek seznamu
A
> (cdr  ‘(a  b  c  d))                  ; vrací seznam bez prvního prvku
(B C D)
> (cdr  ‘( ))          ; prázdný seznam bez prvního prvku je opět prázdný seznam
NIL
> (car  (cdr  ‘(a  (b  c)  d  e)))      ; nejprve se vyhodnocují argumenty
(B C)
> (cons  ‘a  (cons  ´b  nil))           ; NIL představuje prázdný seznam
(A B)
> (cons  ‘a  (cons  ‘b  ‘( )))          ; ( ) je také prázdný seznam
(A B)
> (eq  ‘ab  (car  ‘(ab  c  d)))         ; porovnávané symboly ab jsou stejné
T
> (eq  “kolo”  “kolo”)                  ; řetězy představují různé objekty
NIL
> (eq  ‘( )  nil)                       ; prázdný seznam a symbol NIL jsou stejné
T
> (cond  ((eq  ‘a  ‘b)  ‘c)             ; první podmínka má hodnotu NIL
         (‘d  ‘e)                       ; druhá podmínka má hodnotu D (# NIL)
         (T  “hallo”)  )                ; třetí podmínka se již nevyhodnocuje
E
>
```

6.4.2 Některé další funkce

Některé další funkce (platí pro XLISP):

(exit) Končí činnost interpretu, vrací řízení operačnímu systému.

x+y

Aritmetické funkce (příklady použití):

```
> (+  –3  4  6)                         ; sečítání ( –3 + 4 + 6 )
7
```

```
> (–  2  –1  4  5)                ; odečítání ( 2 – (–1) – 4 – 5 )
– 6
> (∗  1.5  8)                     ; násobení
12
> (/  60  10  2)                  ; celočíselné dělení ( 60 / 10 / 2 )
3
> (/  6  4)                       ; celočíselné dělení
1
> (/  6.0  4)                     ; reálné dělení
1.5
> (/  (float  6)  4)              ; reálné dělení
1.5
> (∗  (+  4  5)  –2  –3)          ; složený výraz (4 + 5) * (–2) * (–3)
54
> (truncate  2.7)                 ; oseknutí na celé číslo
2
> (max  5  6  –8  1)              ; maximum
6
> (min  5  6  –8  1)              ; minimum
-8
> (abs  –99.99)                   ; absolutní hodnota
99.99
>
```

Funkce – predikáty (příklady použití):

```
> (numberp  5)                    ; test na číslo
T
> (numberp  'john)
NIL
> (symbolp  'john)                ; test na symbol
T
> (listp  24)                     ; test na seznam
NIL
> (listp  ‘(a b c))
T
> (minusp  –7.5)                  ; test na  záporné číslo
T
> (minusp  ‘a)                    ; argumentem musí být číslo
error: bad argument type - A
> (minusp  (∗  –3  –2))           ; test na zápornou hodnotu výrazu
NIL
> (zerop  5)                      ; test na nulu
NIL
> (evenp  24)                     ; test na sudé číslo
T
> (>  65  64)                     ; porovnání dvou čísel
T
> (>  50  42.2  8  0  –7.1)       ; porovnání posloupnosti čísel
T
```

```
> (< 1 2 2 5)
NIL
> (<= 56 56 57)
T
> (>= 56 56 57)
NIL
> (= 5 6)
NIL
> (= 4 4.0 4 4 4.0 4 4)
T
> (/= 5 6)                          ; porovnání na nerovnost sousedních čísel
T
> (/= 2 8 6 1 9 2 8 6)
T
> (/= 2 8 4 1 2 2 4 8)
NIL
> (not (< 5 6))                     ; negace
NIL
>(and (< 5 6) (or (> 1 5) (>= 8 7)))      ; logické operace, výsledkem je buď
T
> (and 'a 'b 'c)                           ; NIL nebo hodnota posledního
C
> (and 'a (= 5 4) 'c)                      ; prvku vyhodnocovaného výrazu
NIL
> (null '( ))                       ; test na prázdný seznam
T
> (null '(a b c))
NIL
> (equal 7.0 7.0)                   ; hodnota (eq 7.0 7.0) je NIL !
T
> (equal ‘(a b c) ‘(a b c))         ; hodnota (eq ‘(a b c) ‘(a b c)) je NIL !
T
> (equal “ab_12” “ab_12’)           ; hodnota (eq “ab_12” “ab_12’) je NIL !
T
```

x+y

Funkce pro práci se seznamy (příklady použití):

```
> (cadr '(a b c))                   ; zkrácený zápis pro (car (cdr '(a b c)))
B
> (cddr '(a b c))                   ; zkrácený zápis pro (cdr (cdr '(a b c)))
(C)
                                    ; možné zkrácené zápisy jsou: caar, cadr,
                                    ; cdar, cddr, ..., caaaar, caaadr, …, cddddr
>
> (first '(a b c d e))              ; totéž, co (car '(a b c d e))
A
> (fourth '(a b c d e))             ; totéž, co (cadddr '(a b c d e))
D
> (fourth '(a b c))
NIL
> (nth 3 '(a b c d e))              ; pozor !!! indexování začíná nulou !!!
```

```
D
> (nthcdr  3  '(a  b  c  d  e))            ; pozor !!! indexování začíná nulou !!!
(D  E)
> (last  '(a  b  c  d  e))                 ; vytvoření seznamu z posledního prvku
(E)
> (rest  '(a  b  c  d  e  f))              ; totéž, co (cdr  '(a  b  c  d  e  f))
(B  C  D  E  F)
> (reverse  '(a  b  c  d  e))              ; reverze prvků seznamu
(E  D  C  B  A)
> (delete  'b  '(a  b  c  b  d  e))        ; odstranění všech výskytů prvku v seznamu
(A  C  D  E)
> (list  'where  'is  'Mary)               ; vytvoření seznamu z prvků
(WHERE  IS  MARY)
> (append  '(I)  '(am)  '(hungry))         ; vytvoření seznamu spojením seznamů
(I  AM  HUNGRY)
> (sort  '(1  3.3  6  2  –8.6  4  9)  '<)  ; vzestupné seřazení číselného seznamu
(–8.6  1  2  3.3  4  6  9)
> (sort  '(2  1  8  6  9  3  4)  '>)       ; sestupné seřazení číselného seznamu
(9  8  6  4  3  2  1)
> (member  '2  '(1  2  3  4))              ; je-li první argument prvkem druhého
                                           ; argumentu, kterým musí být seznam,
                                           ; vrací část tohoto seznamu, která začíná
                                           ; daným prvkem
(2  3  4)
> (member  '0  '(1  2  3  4))              ; jinak vrací NIL
NIL
>
```

x+y

I/O funkce (příklady použití):

Čtení zdrojového souboru (tj. souboru s uživatelem definovanými funkcemi)

a) Soubor je v aktuálním adresáři a má příponu lsp:

```
> (load 'name)          ; jméno souboru je libovolné, zde je použito jméno name
; loading "name.lsp"
T
>
```

b) V ostatních případech:

```
> (load 'full_path)     ; například  (load ‘d:/zboril/xlisp/opora/adt.lsp)
; loading "full_path"
T
>
```

Zápis do / čtení ze souboru během výpočtu

Zápis do souboru:

```
> (setq  myf  (open  'mf.txt  :direction  :output))     ; otevření souboru pro zápis
                                  ; myf je jméno filehandle, mf.txt je jméno souboru,
                                  ; obě jména mohou být libovolná
```

```
#<File-Stream: #420c0482
> (print  "Hallo"  myf)        ; zápis vyhodnoceného S-výrazu (řetězu "Hallo")
"Hallo"                        funkce současně vrací stejnou hodnotu (“Hallo”)
> (print  (+  5  6)  myf)      ; zápis vyhodnoceného S-výrazu (hodnoty 11)
11                             funkce současně vrací stejnou hodnotu (11)
> (close  myf)                 ; zavření souboru
NIL
>                              ; Aktuální obsah souboru mf.txt:
                               ; Hallo
                               ; 11
```

Čtení ze souboru:

```
>(setq  mf   (open  'mf.txt  :direction  :input))        ; otevření souboru pro čtení
                               ; mf je jméno filehandle, mf.txt je jméno souboru,
                               ; obě jména mohou být libovolná
#<File-Stream: #420c0482
> (read  mf)                   ; čtení prvního S-výrazu
Hallo
> (read  mf)                   ; čtení druhého S-výrazu
11
> (read  mf)                   ; (neúspěšný) pokus o čtení dalšího S-výrazu
NIL
(close mf)                     ; zavření souboru
NIL
>
```

6.4.3 Definice uživatelských funkcí

Definice uživatelských funkcí:

(defun jméno_funkce (formální parametry) (tělo_funkce))

Příklady definic:

```
> (defun  square  (x)  (*  x  x))              funkce vracející druhou mocninu
SQUARE

> (defun  father  (x)                  funkce vracející jméno otce zadané osoby
        (cond  ((equal  x  'john)      'jack)
               ((equal  x  'joe)       'fred)
               ((equal  x  'susan)     'fred)
               (T                      'I_do_not_ know.) ))
FATHER
```

Pro definici uživatelských funkcí je, stejně jako u jazyka PROLOG, typická rekurze:

```
> (defun factorial (n)                         funkce vracející faktoriál
        (cond  ((<  n  0)  (print  “Error – argument must be positive number!”))
               ((eq  n  1)  1)
               (T  (*  n  (factorial  (–  n  1)))) ) )
```

Příklady použití:

```
> (square 2.5)
6.25
> (father  'joe)                ; kdo je otcem Joe?
FRED
>(father  'mary)               ; kdo je otcem Mary?
I_DO_NOT_KNOW.
> (factorial 7)
5040
> (factorial -3)
Error – argument must be positive number!
>
```

Úkol: Definujte funkci pro určení indexu maximálního prvku lineárního číselného seznamu, funkci pro výběr druhého maximálního prvku lineárního číselného seznamu, funkci pro odstranění posledního prvku seznamu a funkci pro stanovení počtu prvků obecného seznamu.

6.5 ADS

Abstraktní datové struktury (ADS) - zásobník, fronta, množina:

V následujících funkcích pro vložení prvku je prvním argumentem vkládaný prvek E a druhým argumentem aktuální seznam L (zásobník, fronta, množina). Hodnotou funkce je nový stav příslušné datové struktury. Totéž platí pro výběr prvku z množiny. Funkce pro výběr prvku ze zásobníku/fronty (seznamu L) vypisují vybraný prvek a jejich hodnotou je seznam L bez vybraného prvku. U operací průniku, sjednocení a rozdílu dvou množin jsou tyto množiny argumenty funkce a hodnotou funkce je opět nový stav, tj. seznam reprezentující výsledek příslušné operace.

Test na prázdný seznam (stejné pro zásobník, frontu i množinu):

```
(defun  empty  (L) (null  L))
```

Zásobník (Stack)

```
(defun  push  (E  L)  (cons  E  L)  )                    vložení prvku

(defun  pop  (L)                                         výběr  prvku
        (cond  ((null  L)  (cond  ((print  "error – stack is empty!")  L)))
               ((print  (car  L))  (cdr  L))  )  )
```

Pozn.: Je-li seznam L prázdný, funkce *cond* testuje podmínku, kterou je výpis chybového hlášení. Hodnota tohoto S-výrazu je rozdílná od NIL, a proto se jako hodnota funkce *pop* vrátí opět prázdný seznam.

Fronta (Queue)

```
(defun  enqueue  (E  L)  (append  L  (cons  E  nil))  )      vložení prvku

(defun  dequeue  (L)                                         výběr  prvku
        (cond  ((null  L)  (cond  ((print  "error – queue is empty!")  L)))
               ((print  (car  L))  (cdr  L))  )  )
```

Fronta s prioritou (Priority Queue)

```
(defun insert (E L)                                   vložení prvku
      (cond ((null L) (cons E L))
            ((precedes E (car L)) (cons E L))
            ( T (cons (car L) (insert E (cdr L)))) ) )

(defun precedes (A B) (< A B) )                       platí pouze pro čísla!

(defun dequeue (L)                                    výběr prvku
      (cond ((null L) (cond ((print "error – queue is empty!") L)))
            ((print (car L)) (cdr L)) ) )
```

Množina (Set)

```
(defun add (E L)                                      vložení prvku
      (cond ((member E L) L)
            ( T (cons E L)) ) )

(defun delete (E L)                                   výběr prvku
      (cond ((null L) L)
            ((equal E (car L)) (cdr L))
            ( T (cons (car L) (delete E (cdr L)))) ) )

(defun intersection (L1 L2)                           průnik dvou množin
      (cond ((null L1) nil)
            ((member (car L1) L2)
                  (cons (car L1) (intersection (cdr L1) L2)))
            ( T (intersection (cdr L1) L2)) ) )

(defun union (L1 L2)                                  sjednocení dvou množin
      (cond ((null L1) L2)
            ((member (car L1) L2) (union (cdr L1) L2))
            (T (cons (car L1) (union (cdr L1) L2))) ) )

(defun difference (L1 L2)                             rozdíl dvou množin
      (cond ((null L1) nil)
            (((member (car L1) L2) (difference (cdr L1) L2))
            (T (cons (car L1) (difference (cdr L1) L2))) ) )

(defun subset (L1 L2)          testuje, zda první množina je podmnožinou druhé
      (cond ((null L1) T)
            ((member (car L1) L2) (subset (cdr L1) L2))
            ( T nil) ) )

(defun equal_set (L1 L2)                              testuje rovnost množin
      (and (subset L1 L2) (subset L2 L1)) )
```

Ukázka konverzace

Podobně, jako jsme v jazyku PROLOG definovali klauzuli *do*, můžeme v jazyku LISP definovat funkci *do*, pomocí které si ukážeme možné použití výše definovaných funkcí (jde o „program“, který se spustí zadáním funkce (*do nil*)):

```
(defun do (L)
        (cond ((listp L) (do_act L))                    ; L musí být seznam
              ( T "error - argument is not list!") ) )  ; jinak konec

(defun do_act (L) (action (read_input) L) )             ; zde je začátek rekurze

(defun read_input ( )                                   ; čtení příkazu
        (cond ((print
  "Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): ")
              (read)) ) )

(defun action (act L)                                   ; rozhodnutí o akci
        (cond ((equal act 'empty) (do_act (do_empty L)))
              ((equal act 'push) (do_act (do_push L)))
              ((equal act 'pop) (do_act (do_pop L)))
              ((equal act 'enqueue) (do_act (do_enqueue L)))
              ((equal act 'insert) (do_act (do_insert L)))
              ((equal act 'dequeue) (do_act (do_dequeue L)))
              ((equal act 'add) (do_act (do_add L)))
              ((equal act 'delete) (do_act (do_delete L)))
              ((equal act 'end) "by by")
              ( T (do_act L)) ) )
(defun do_empty (L)                                     ; akce pro příkaz empty
        (cond ((print (empty L)) (print L))
              ( T (print L)) ) )

(defun do_push (L)                                      ; akce pro příkaz push
        (cond ((print "Zadejte prvek: ") (print (push (read) L))) ) )

(defun do_pop (L) (print (pop L)) )                     ; akce pro příkaz pop

(defun do_enqueue (L)                                   ; akce pro příkaz enqueue
        (cond ((print "Zadejte prvek: ") (print (enqueue (read) L))) ) )

(defun do_insert (L)                                    ; akce pro příkaz insert
        (cond ((print "Zadejte prvek: ") (print (insert (read) L))) ) )

(defun do_dequeue (L) (print (dequeue L)) )             ; akce pro příkaz dequeue

(defun do_add (L)                                       ; akce pro příkaz add
        (cond ((print "Zadejte prvek: ") (print (add (read) L))) ) )

(defun do_delete (L)                                    ; akce pro příkaz delete
        (cond ((print "Zadejte prvek: ") (print (delete (read) L))) ) )
```

Ukázka konverzace:

```
> (do nil)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
empty
T
NIL
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
push
"Zadejte prvek: "
a
(A)
```

```
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
push
"Zadejte prvek: "
b
(B A)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
empty
NIL
(B A)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
pop
B
(A)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
enqueue
"Zadejte prvek: "
-4
(A -4)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
enqueue
"Zadejte prvek: "
5
(A -4 5)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
dequeue
A
(-4 5)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
insert
"Zadejte prvek: "
2
(-4 2 5)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
insert
"Zadejte prvek: "
3.6
(-4 2 3.6 5)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
add
"Zadejte prvek: "
kolo
(KOLO -4 2 3.6 5)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
delete
"Zadejte prvek: "
3.6
(KOLO -4 2 5)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
pop
KOLO
```

(-4 2 5)
"Zadejte prikaz (empty push pop enqueue insert dequeue add delete end): "
end
"by by"
> (exit)

6. 6 Základní prohledávací algoritmy

Základní prohledávací algoritmy

V této kapitole jsou uvedeny programy implementující základní prohledávací algoritmy BFS (pro úlohu „hlavolam 8“) a DFS (pro úlohu dvou džbánů).

6.6.1 BFS

x+y

BFS (se seznamem CLOSED) - úloha „hlavolam 8“:

Prvky seznamů *Open*, *Closed* a *Temp* jsou dvouprvkové seznamy – prvním prvkem je daný uzel a druhým prvkem jeho bezprostřední předchůdce.

Pozn.: Stejně jako u podobného příkladu v jazyku PROLOG je pozice kamene určena jeho pozicí v seznamu

1	2	3
4	5	6
7	8	9

, tzn., že například stav

1	4	2
7	8	3
	6	5

bude popsán seznamem takto: (1 4 2 7 8 3 o 6 5).

V následujících definicích funkcí jsou pro přehlednost podmínkové části funkcí *cond* zobrazeny modrým písmem a odpovídající hodnotové S-výrazy červeným písmem.

```
; část programu nezávislá na konkrétní úloze
(defun bfs (Start Goals) (path (list (cons Start nil)) '( ) Goals))

(defun path (Open Closed Goals)
        (cond   ( (null Open) 'No_solution_was_found )
                ( (memb (caar Open) Goals)
                        (write_path (caar Open) (cons (car Open) Closed)) )
                ( T (path (append (cdr Open) (get_children Open '( ) Closed))
                          (cons (car Open) Closed) Goals) ) ))

(defun memb (E L)
        (cond   ( (null L) nil )
                ( (equal E (car L)) T )
                ( T (memb E (cdr L)) ) ))

(defun get_children (Open Temp Closed)
        (cond   ( (move (caar Open) (append (cdr Open) Temp Closed))
                        (cons (list (move (caar Open)
                                          (append (cdr Open) Temp Closed))
                                    (caar Open))
                              (get_children Open
                                            (cons (list (move (caar Open)
                                                              (append (cdr Open) Temp Closed))
                                                        (caar Open)) Temp)
                                                  Closed)) )
                ( T  nil ) ))
```

```
(defun write_path (State Closed)
        (and (print 'The_path_is:) (write_state State Closed) 'by_by)  )

(defun write_state (State Closed)
        (cond   ( (eq (cadr (find State Closed)) nil)
                        (print (car (find State Closed))) )
                ( (write_state (cadr (find State Closed)) Closed)
                        (print (car (find State Closed))) ) ))

(defun find (State Closed)
        (cond   ( (equal State (caar Closed)) (car Closed) )
                ( T (find State (cdr Closed)) ) ))


% část závislá na konkrétní úloze, úloha dvou džbánů („hlavolam 8“):
(defun move (S L)
        (cond   ( (and  (> (sp_pos S) 3)                ; možný tah nahoru
                        (not_in (change (sp_pos S) (- (sp_pos S) 3) S) L))
                                (change (sp_pos S) (- (sp_pos S) 3) S) )
                ( (and  (/= 0 (mod (sp_pos S) 3))       ; možný tah doprava
                        (not_in (change (sp_pos S) (+ (sp_pos S) 1) S) L))
                                (change (sp_pos S) (+ (sp_pos S) 1) S) )
                ( (and  (< (sp_pos S) 7)                ; možný tah dolů
                        (not_in (change (sp_pos S) (+ (sp_pos S) 3) S) L))
                                (change (sp_pos S) (+ (sp_pos S) 3) S) )
                ( (and  (/= 1 (mod (sp_pos S) 3))       ; možný tah doleva
                        (not_in (change (sp_pos S) (- (sp_pos S) 1) S) L) )
                                (change (sp_pos S) (- (sp_pos S) 1) S) ) ))

(defun not_in (E L)
        (cond   ( (null L) T )
                ( (equal E (caar L)) nil )
                ( T (not_in E (cdr L)) ) ))

(defun mod (x y) (- x (* (/ x y) y)) )

(defun sp_pos (S)
        (cond   ( (eq 'o (car S)) 1 )
                ( T (+ 1 (sp_pos (cdr S))) ) ))

(defun change (sp_p p_p S) (insert 'o p_p (insert (piece p_p S) sp_p S)) )

(defun insert (x ix S)
        (cond   ( (eq ix 1) (cons x (cdr S)) )
                ( T (cons (car S) (insert x (- ix 1) (cdr S))) ) ))
(defun piece (p_p S)
        (cond   ( (eq p_p 1) (car S) )
                ( T (piece (- p_p 1) (cdr S)) ) ))
```

Program se spustí voláním funkce *bfs*, jejímž prvním argumentem bude počáteční stav a druhým argumentem seznam cílových stavů úlohy, např. pro jeden cílový stav takto:

```
> (bfs '(1 4 2 8 o 3 7 6 5) '((1 2 3 8 o 4 7 6 5)))
```

Hodnotou funkce *bfs* bude hodnota funkce *path*, která má tři argumenty: Seznam *Open*, do kterého je uložen počáteční uzel *(Start nil)*, prázdný seznam *Closed* (tj. Seznam '( )) a seznam cílových stavů *Goals*.

Hodnotou funkce *path* je buď výpis hlášení *No_solution_was_found* (v případě, že seznam *Open* je prázdný), nebo výpis řešení představovaný hodnotou funkce *write_path* (v případě, že první uzel v seznamu *Open* je uzlem cílovým), nebo hodnota funkce *path*, ve které je seznam *Open* dán spojením zbytku původního seznamu *Open* bez jeho prvního uzlu (hodnota funkce *cdr*) a hodnotou funkce *append* všech následníků prvního uzlu původního seznamu *Open* (hodnota funkce *get_children*), a ve které je k seznamu *Closed* přidán (hodnota funkce *cons*) první prvek původního seznamu *Open* (hodnota funkce *car*).

Funkce *memb* vrací hodnotu T, pokud je její první argument (seznam reprezentující uzel) prvkem seznamu daného druhým argumentem (seznamem cílových stavů). Jinak vrací tato funkce hodnotu nil.

Funkce *get_children* vrací jako svou hodnotu seznam všech bezprostředních následníků prvního uzlu ze seznamu *Open*, kteří nejsou ve zbytku seznamu *Open*, ani v seznamu *Closed*, a vrací hodnotu nil, pokud žádný takový následník neexistuje. K tomu využívá pomocný seznam *Temp*, do kterého postupně generované následníky ukládá (tento seznam je při prvním volání funkce *get_children* prázdný). Funkce *get_children* pracuje následovně: pokud existuje bezprostřední následník expandovaného uzlu (hodnota funkce *move* je různá od nil), který ještě není v seznamu daném dočasným spojením (funkce *append*) zbytku seznamu *Open* (funkce *cdr*) a seznamů *Temp* a *Closed*, tak tohoto následníka připojí (funkce *cons*) k ostatním bezprostředním následníkům vráceným funkcí *get_children* (rekurzivní volání), jejímž druhým argumentem je nyní seznam Temp doplněný o aktuálního bezprostředního následníka (funkce *cons*), ke kterému je připojen jeho předchůdce – expandovaný uzel (funkce *list*). Dokud existuje další následník, tak se funkce *get_children* stále zanořuje, až nakonec (následník již neexistuje) se jí přiřadí hodnota prázdného seznamu - nil. Při vynořování se pak připojují jednotliví následníci, až při konečném vynoření je její hodnotou seznam všech přípustných bezprostředních následníků.

Funkce *not_in* vrací hodnotu T, pokud uzel daný prvním argumentem není v seznamu uzlů daném druhým argumentem. V tomto seznamu jsou však uzly uvedeny ve dvouprvkových seznamech (druhými prvky jsou bezprostřední předchůdci těchto uzlů), proto je zde použita funkce *caar*.

Funkce *write_path* vypíše informaci *The_path_is:*, dále vypíše řešení/cestu jako hodnotu funkce *write_state* a vrátí hodnotu *by_by* (funkcí *and* je zde vynuceno vyhodnocení obou funkcí (*print* a *write_state*) podmínkové části).

Funkce *write_state* vychází z cílového stavu daného prvním argumentem a postupně hledá jeho předchůdce v seznamu *Closed* pomocí funkce *find*. Rekurzivně se zanořuje až narazí na kořenový uzel (jeho předchůdcem je hodnota nil). Pak vypíše hodnotu tohoto uzlu a při vynořování vypisuje postupně

všechny jeho relevantní následníky konče cílovým uzlem.

Funkce *find* hledá předchůdce uzlu daného prvním argumentem v seznamu uzlů daném druhým argumentem. Stejně jako u funkce *not_in* jsou uzly v tomto seznamu uvedeny ve dvouprvkových seznamech, a proto i zde je použita funkce *caar*.

Funkce *move* popisují možné přechody/tahy mezi jednotlivými stavy (uzlem a jeho bezprostředními následníky). První části každé podmínky (funkce *and*) testují, zda jsou možné (postupně) tahy nahoru, doprava dolů či doleva, druhé části podmínek „hlídají“, aby příslušný nový stav byl skutečně novým stavem, tj. aby to byl stav, který dosud není v seznamu daném druhým argumentem (spojení seznamů *Open*, *Temp* a *Closed* – viz definici funkce *get_children*). Pokud některý test uspěje, tak funkce *move* vrátí jako svou hodnotu příslušný nový stav.

Hodnotou funkce *mod* je zbytek (modulo) po celočíselném dělení.

Funkce *sp_pos* vrací pozici prázdného políčka (představovaného symbolem o), funkce *piece* vrací číslo kamene, který leží na zadané pozici.

Funkce *change* vrací nový stav. Jejími argumenty jsou původní a nová pozice prázdného políčka a vlastní tah provádí dvojím voláním funkce *insert*: nejprve se uloží číslo kamene, který leží na nové pozici prázdného políčka, na aktuální pozici prázdného políčka a poté se vloží symbol o na novou pozici prázdného políčka.

Funkce *insert* vloží symbol daný prvním argumentem (číslo kamene, nebo symbol o pro prázdné políčko) na pozici danou druhým argumentem v seznamu daném třetím argumentem.

Úkol: Proveďte nutné úpravy programu (definice funkcí) tak, aby řešil úlohu dvou džbánů metodou A* (s minimalizaci množství vody (buď vody ze zdroje, nebo vody, se kterou se manipuluje).

6.6.2 DFS

DFS - úloha dvou džbánů:

Program pro metodu DFS (bez CLOSED) se od předchozího programu pro metodu BFS zcela liší - zásadní změnou je především to, že každý stav je uložen v seznamu společně se všemi svými předchůdci.

```
; část programu nezávislá na konkrétní úloze
(defun dfs (Start Goals) (path (list (cons Start nil)) Goals))

(defun path (Open Goals)
        (cond   ( (null Open) 'No_solution_was_found )
                ( (member (caar Open) Goals) (write_path (car Open)) )
                ( T (path (append (get_children Open '( )) (cdr Open)) Goals) ) ))

(defun member (E L)
        (cond   ( (null L) nil )
                ( (equal E (car L)) T )
                ( T (member E (cdr L)) ) ))
```

```
(defun get_children (Open Temp)
        (cond   ( (move (caar Open) (cdar Open) (cdr Open) Temp)
                  (cons
                    (cons
                      (move (caar Open) (cdar Open) (cdr Open) Temp)
                      (car Open) )
                    (get_children Open
                      (cons
                        (cons
                          (move (caar Open) (cdar Open) (cdr Open) Temp)
                          (car Open) )
                        Temp  ) ) ) )
                ( T nil ) ) )

(defun write_path (Path)
        (cond   ( (and (print 'The_path_is:) (Print_Reverse_Path Path))
                  '(by_by) ) ))

(defun Print_Reverse_Path (Path)
        (cond   ( (null (cdr Path)) (print (car Path)) )
                ( (Print_Reverse_Path (cdr Path)) (print (car Path))) ))

; část programu závislá na konkrétní úloze (dva džbány s obsahy 4 a 3 litry
(defun move (S A RO TMP)            ; State Ancestors Rest_of_Open Temporary
        (cond   ( (and  (< (car S) 4) (not_in (cons 4 (cdr S)) A RO TMP))
                        (cons 4 (cdr S)) )
                ( (and  (< (cadr S) 3) (not_in (list (car S) 3) A RO TMP))
                        (list (car S) 3) )
                ( (and  (> (car S) 0) (not_in (cons 0 (cdr S)) A RO TMP))
                        (cons 0 (cdr S)) )
                ( (and  (> (cadr S) 0) (not_in (list (car S) 0) A RO TMP))
                        (list (car S) 0) )
                ( (and  (> (car S) 0) (< (cadr S) 3) (>= (- 3 (cadr S)) (car S))
                        (not_in (list 0 (+ (car S) (cadr S))) A RO TMP))
                        (list 0 (+ (car S) (cadr S))) )
                ( (and  (> (car S) 0) (< (cadr S) 3) (< (- 3 (cadr S)) (car S))
                        (not_in (list (+ (- (car S) 3) (cadr S)) 3) A RO TMP))
                        (list (+ (- (car S) 3) (cadr S)) 3) )
                ( (and  (> (cadr S) 0) (< (car S) 4) (>= (- 4 (car S)) (cadr S))
                        (not_in (list (+ (car S) (cadr S)) 0) A RO TMP))
                        (list (+ (car S) (cadr S)) 0) )
                ( (and (> (cadr S) 0) (< (car S) 4) (< (- 4 (car S)) (cadr S))
                        (not_in (list 4 (+ (- (cadr S) 4) (car S))) A RO TMP))
                        (list 4 (+ (- (cadr S) 4) (car S))) ) ))

(defun not_in (S A RO TMP)
        (cond( (member S A) nil )
              ( (mcaar S RO) nil )
              ( (mcaar S TMP) nil )
              ( T T ) ))
```

```
(defun mcaar (E L)
        (cond  ( (null L) nil )
               ( (equal E (caar L)) T )
               ( T (mcaar E (cdr L)) ) ))
```

Program se spustí voláním funkce d*fs*, jejímž prvním argumentem bude počáteční stav a druhým argumentem seznam cílových stavů úlohy, např. takto:

> (dfs '(0 0) '((2 0) (0 2)))

Hodnotou funkce *dfs* bude hodnota funkce *path*, která má také dva argumenty: Seznam *Open*, do kterého je uložen počáteční uzel *Start* (*Open* = ((*Start*))), a seznam cílových stavů *Goals*.

Hodnotou funkce *path* je podobně jako u metody BFS buď výpis hlášení *No_solution_was_found* (v případě, že seznam *Open* je prázdný), nebo výpis řešení představovaný hodnotou funkce *write_path* (v případě, že první uzel v seznamu *Open* je uzlem cílovým), nebo hodnota funkce *path*, ve které je nový seznam *Open* dán spojením (hodnotou funkce *append*) všech následníků prvního uzlu původního seznamu *Open* (hodnota funkce *get_children*) se zbytkem původního seznamu *Open* bez jeho prvního uzlu (hodnota funkce *cdr*).

Funkce *member* vrací hodnotu T, pokud je její první argument (seznam reprezentující uzel) prvkem seznamu daného druhým argumentem (seznamem cílových stavů). Jinak vrací tato funkce hodnotu nil.

Funkce *get_children* vrací jako svou hodnotu seznam všech bezprostředních následníků prvního uzlu ze seznamu *Open*. Jde o poměrně složitou funkci, kterou lze slovně popsat takto: Pokud existuje tah z prvního uzlu v seznamu *Open* zleva (funkce *caar*) takový, že nový uzel není předchůdcem expandovaného uzlu (funkce *cdar*) a současně není ve zbytku seznamu *Open* (funkce *cdr*), ani v dočasném seznamu *Temp* (ten je při volání funkce *get_children* z funkce *path* prázdný), tak funkce volá rekurzivně sama sebe - prvním argumentem při tomto volání je opět původní seznam *Open*, druhým argumentem je pak původní seznam *Temp*, ke kterému je zleva (toto je nepodstatné) připojen již zjištěný možný tah z předchozí podmínkové části funkce (funkce *cons*), doplněný o své předchůdce (druhá funkce *cons*). Neexistuje-li přípustný tah, tak funkce vrací prázdný seznam – k tomuto prázdnému seznamu se pak při „vynořování“ jednotlivé nové uzly připojují až výslednou vrácenou hodnotou je seznam všech bezprostředních následníků expandovaného uzlu.

Funkce *write_path* vypíše informaci *The_path_is:*, dále vypíše řešení/cestu jako hodnotu funkce *Print_Reverse_path* a vrátí hodnotu *by_by* (podobn2 jako u metody BFS).

Funkce *Print_Reverse_path* je jednoduchá, protože cílový stav/uzel je v jediném seznamu se všemi svými předchůdci – vypíše prvky seznamu zprava doleva, t.j. od počátečního k cílovému uzlu.

Funkce *move* jsou nyní nepatrně složitější, než u metody BFS. Nový uzel/stav *S* (State) totiž nesmí být ani předchůdcem *A* (Ancestors) expandovaného uzlu, ani nesmí být ve zbytku seznamu *Open* (*RO* - Rest_of_Open), ani nesmí být mezi již vygenerovanými následníky expandovaného uzlu TMP (Temporary). Splnění

těchto podmínek zajišťuje funkce *not_in*. Tato funkce obsahuje ve své definici funkce *member* (prvek seznamu, funkce je popsána výše) a *mcaar* (první prvek některého seznamu v seznamu – seznamy RO a TMP obsahují seznamy jednotlivých uzlů a jejich předchůdců, např. (((3 0) (0 3) (0 0)) ((4 0) (0 0))) !).

Úkol: Úkol: Definujte funkce *move* pro jiné velikosti džbánů. Pokuste se definovat tahy i pro některé jiné úlohy.

6.7 Shrnutí

Shrnutí

V kapitole byly vysvětleny principy jazyka LISP a bylo zde ukázáno, jak je možné v tomto jazyku implementovat prohledávací metody BFS a DFS. Z principů jazyka je důležité znát jeho syntax, postup interpretu při vyhodnocování S-výrazů, a také některé zabudované funkce. Je samozřejmě nutné umět definovat nové funkce, tj. psát jednoduché programy v tomto jazyku.

6.8 Kontrolní otázky

Kontrolní otázky

1. Jak je definován S-výraz?
2. Jak postupuje interpret jazyka LISP při vyhodnocování S-výrazů?
3. Jak se definují nové funkce?
4. Jak vypadá program v jazyku LISP?

7 REPREZENTACE ZNALOSTÍ

Cílem této kapitoly je seznámit studenty se základními schématy používanými pro reprezentaci znalostí v UI. Postupně zde budou popsána logická, síťová, strukturální a procedurální schémata a budou diskutována jejich silná a slabá místa.


5 hod

7.1 Úvodní informace

Úvodní informace

Znalosti jsou základem inteligence. Bez znalostí nelze dělat žádná rozhodnutí, ani žádné jiné „inteligentní“ činnosti.

Lidé používají k popisu znalostí především přirozené jazyky. Pro popis specifických znalostí pak používají i jiné „jazyky“, například chemické vzorce, noty apod. Pro strojové použití však žádný z těchto jazyků není vhodný – přestože není obtížné navrhnout a realizovat jejich vnitřní reprezentaci, je prakticky nemožné takto zapsané znalosti efektivně využívat.

Pod pojmem „reprezentace znalostí“ (*knowledge representation*) se v UI obvykle rozumí nejen vlastní reprezentace, ale i výběr relevantních znalostí.

Výběr relevantních znalostí je závislý na charakteru řešených problémů a neexistuje obecný návod, jak takové znalosti určit. Přesto je tento výběr velmi důležitý a závisí na něm nejen efektivnost, ale často i úspěšnost práce umělého systému.

Formální popisy znalostí (schémata reprezentace) by měly respektovat zejména požadavky na:

- snadné přidávání a modifikaci znalostí,
- sémantické sdružování příbuzných znalostí,
- hierarchické členění znalostí,
- snadnou a efektivní manipulaci s těmito znalostmi.

Dále by měly tyto popisy poskytovat i prostředky pro reprezentaci dědičností, výjimek, nejistot, nejednoznačností, příčinností a dočasností.

Schémata pro reprezentaci znalostí klasické UI se rozdělují do čtyř základních kategorií (první tři kategorie představují tzv. deklarativní schémata):

- logická schémata reprezentace,
- síťová schémata reprezentace,
- strukturální schémata reprezentace,
- procedurální schémata reprezentace.

7.2 Logická schémata

Logická schémata

Logická schémata reprezentace používají k reprezentaci znalostí různé logiky, nejčastěji predikátovou logiku prvního řádu. Tato logika (její principy byly popsány v kap. 4.3) je dominantní proto, že má velmi dobře definovanou sémantiku a má jasnou inferenční strategii.

Formule predikátové logiky umožňují poměrně snadnou a efektivní reprezentaci dědičností, výjimek a lze jimi postihnout i hierarchické členění znalostí. Například obecné předpoklady pro třídu ptáků dědí každý příslušník této třídy:

$$\forall x(\text{pták}(x) \Rightarrow \text{létá}(x))$$
$$\forall x(\text{pták}(x) \Rightarrow \text{má\_peří}(x))$$

Někdy však některý obecný předpoklad nemusí být pravdivý. Nelétá například pštros, pták se zlomeným křídlem, nebo pták, který je mechanickou hračkou. Takové výjimky pak lze ošetřit následovně:

$\forall x(pták(x) \wedge \neg výjimka\_pták(x) \Rightarrow létá(x))$
výjimka_pták(pštros)
$\forall x(pták(x) \wedge$ zlomené_křídlo$(x) \Rightarrow$ výjimka_pták$(x))$
$\forall x(pták(x) \wedge$ mechanická_hračka$(x) \Rightarrow$ výjimka_pták$(x))$
...
pták(skřivan)
pták(orel)
pták(pštros)

Predikátová logika 1. řádu má bohužel také řadu nevýhod, které její použití jako univerzálního schématu reprezentace znalostí výrazně omezují:

- Logické formule mohou být pravdivé i když jsou nesmyslné:

  je_rovno(plus(2, 2), 5) $\Rightarrow$ barva(myš,modrá)

- Úprava formulí může změnit jejich původní význam:

  zajíc(x) $\Rightarrow$ hnědý(x) ; je-li *x* zajícem, je *x* hnědé
  $\neg$zajíc(x) $\vee$ hnědý(x)
  hnědý(x) $\vee \neg$zajíc(x)
  $\neg$hnědý(x) $\Rightarrow \neg$zajíc(x) ; není-li *x* hnědé, není *x* zajícem

- Predikátová logika nepodporuje sémantické sdružování příbuzných znalostí, neboť informace o objektech a jejich vztazích jsou uložené ve zcela nezávislých klauzulích.

- Predikátová logika je logikou monotónní. Proto každá přidaná klauzule musí být pravdivá a odvozené klauzule musí být stále platné. Jinými slovy to znamená, že znalosti lze přidávat, ale nelze je modifikovat. Tato skutečnost činí problémy při formulaci výroků založených na důvěře, resp. na nejistých předpokladech a při formulaci výroků s omezenou časovou platností.

- Predikátová logika striktně rozlišuje mezi pravdou a nepravdou. V reálných aplikacích je však výhodné akceptovat i výroky typu „nejsem si zcela jist“, „s největší pravděpodobností ne“, „asi ano“ apod.

Poslední dva problémy se snaží řešit některé netradiční logiky, jejichž popis přesahuje rámec předmětu (nemonotónní logika, modální logika, temporální logika, vícehodnotové logiky, fuzzy logika, ap.).

7.3 Síťová schémata

Síťová schémata

Síťová schémata reprezentace jsou schémata grafická. Umožňují přirozenou, relativně jednoduchou a sémanticky sdruženou reprezentaci objektů (individuí) i abstraktních pojmů (tříd) a příslušných vztahů mezi nimi.

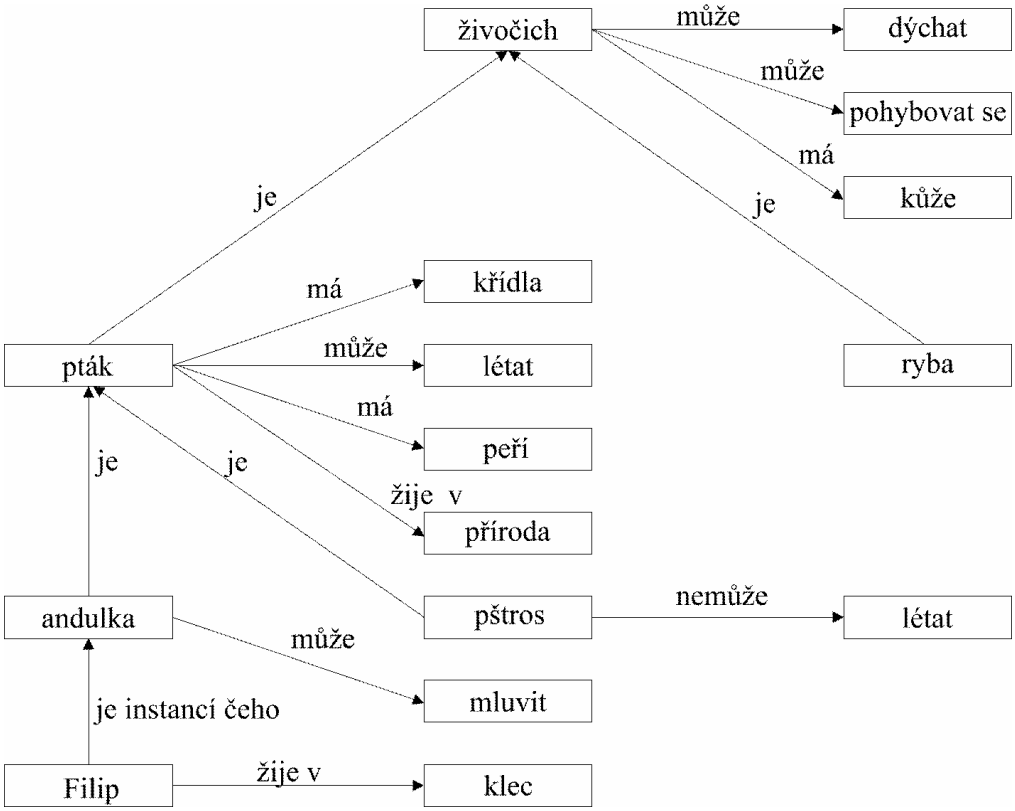
Každý objekt i pojem je v těchto schématech reprezentován jediným způsobem a na jediném místě a všechny objekty a pojmy, které s nějakým jiným objektem či pojmem souvisí, musí s ním být, třeba i zprostředkovaně, propojeny. Za představitele síťových schémat se považují sémantické sítě a pojmové grafy.

7.3.1 Sémantické sítě

Sémantické sítě

Sémantické sítě jsou orientované grafy s popsanými uzly i hranami. Uzly představují objekty nebo pojmy a hrany mezi uzly ukazují na jejich vzájemné vztahy. Příklad sémantické sítě je uveden na Obr. 7.1.

Základním vztahem v sémantických sítích je vztah "je". Používá se obvykle jak pro označení příslušnosti objektu o ke třídě T ($o \in T$ - přesnější označení tohoto vztahu je "je instancí“), tak i pro označení vztahu pojmu A k obecnějšímu pojmu B, resp. vztahu podtřídy A ke třídě B ($A \subset B$). Základním vztahem je uvedený vztah proto, že umožňuje dědění vlastností.



Obr. 7.1 Příklad sémantické sítě

Sémantické sítě umožňují snadné přidávání a modifikaci znalostí, sémantické sdružování příbuzných znalostí i jejich hierarchické členění. Poskytují prostředky pro reprezentaci dědičností a výjimek, problémy jsou opět s reprezentací nejistých a dočasných informací. Největším jejich problémem je však manipulace s uloženými znalostmi – není ani snadná, ani efektivní.

Snaha o zjednodušení manipulace vedla ke standardizaci relací v těchto sítích.

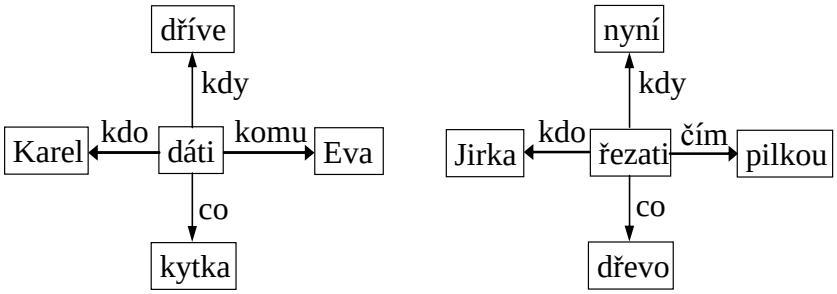
Nejjednoduššími sémantickými vztahy jsou vztahy vyplývající z konkrétních sloves, které přímo zahrnují aktéry, objekty, prostředky a čas (*case frames*). Například věty *Karel dal Evě kytku* a *Jirka řeže dřevo pilkou* lze reprezentovat sémantickými sítěmi podle Obr. 7.2.

Nejsložitějšími sémantickými vztahy se naopak vyznačují sítě teorie pojmové závislosti (*Conceptual dependency*). Jde o starý a dnes již nepoužívaný, nicméně zřejmě o nejkomplexnější přístup k reprezentaci znalostí sémantickými sítěmi.

Základním pojmem je zde akce, která může být reprezentována jednou nebo několika akcemi z následující množiny primitivních akcí:

ATRANS	abstraktní přenos (*abstract transfer*), např. dát
PTRANS	fyzický přenos (*physical transfer*), např. jít
MTRANS	mentální přenos (*mental transfer*), např. vyprávět
MBUILD	mentální tvorba (*mental building*), např. rozhodovat
MOVE	pohyb části těla (*move*), např. kopnout
GRASP	uchopení objektu někým (*grasp*), např. chňapnout
PROPEL	aplikace fyzické síly na objekt (*propel*), např. tlačit
SPEAK	produkování zvuku (*speak*), např. mluvit
ATTEND	zaměření pozornosti (*attend*), např. poslouchat
INGEST	přijímání potravy (*ingest*), např. jíst
EXPEL	vylučování (*expel*), např. plakat

Bližší popis relací založených na výše uvedených akcích přesahuje rámec této opory.



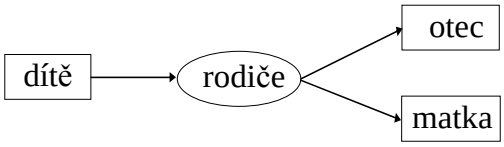
Obr. 7.2 Příklady základních vztahů (*case frames*)

7.3.2 Pojmové grafy

Pojmové grafy

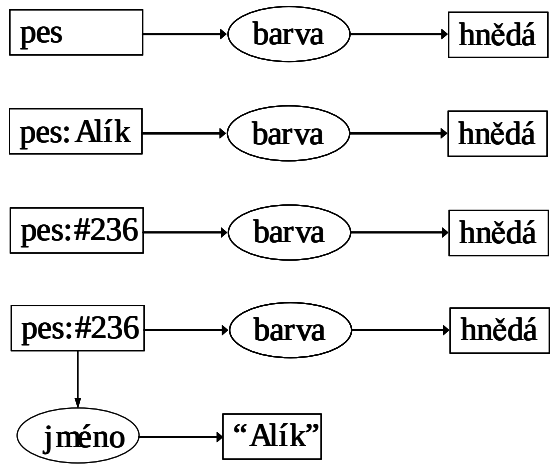
Pojmové grafy nemají, na rozdíl od sémantických sítí, ohodnocené hrany. Vzájemné vztahy mezi pojmy zde reprezentují speciální relační uzly a proto každá hrana musí spojovat uzel reprezentující pojem s uzlem reprezentujícím relaci. Na relace není přitom kladeno žádné omezení - mohou být unární, binární, atd. Každý pojmový graf pak reprezentuje libovolně složitý jednoduchý výrok a báze znalostí je tvořena množinou těchto grafů. Pro snadné rozlišení pojmových uzlů a relačních uzlů jsou první kresleny jako obdélníky a druhé jako ovály.

Příklad jednoduchého pojmového grafu – relace mezi třemi pojmy:



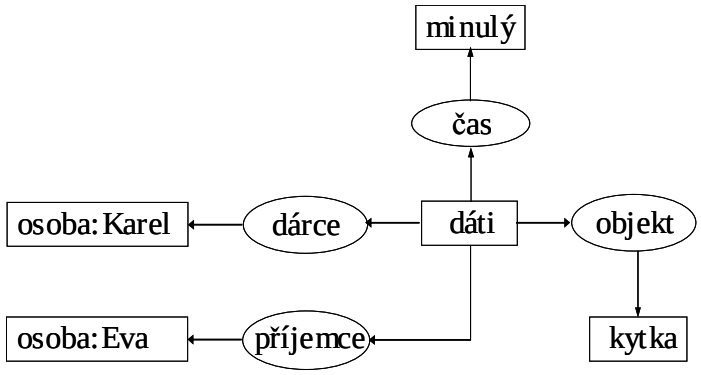
Obr. 7.3 Příklad jednoduchého pojmového grafu

Pojmové grafy umožňují rozlišovat mezi generickými a individuálními pojmy. Na Obr. 7.4 se první vztah týká obecného psa, druhý vztah nějakého psa se jménem Alík, třetí vztah konkrétního psa (označeného indexem 236) a poslední čtvrtý vztah konkrétního psa Alíka.



Obr. 7.4 Příklad na generické a individuální pojmy:

Na následujícím obrázku (Obr. 7.5) je pro porovnání s reprezentací pomocí sémantických sítí uveden pojmový graf věty *Karel dal Evě kytku* (ve smyslu, že nějaký Karel dal nějaké Evě nějakou kytku).

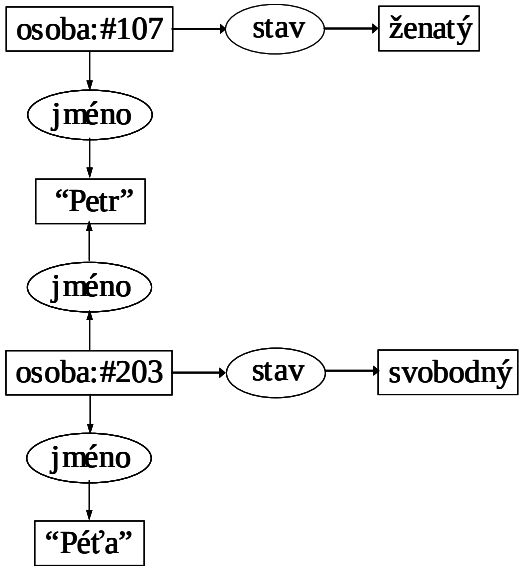


Obr. 7.5 Pojmový graf věty *Karel dal Evě kytku*

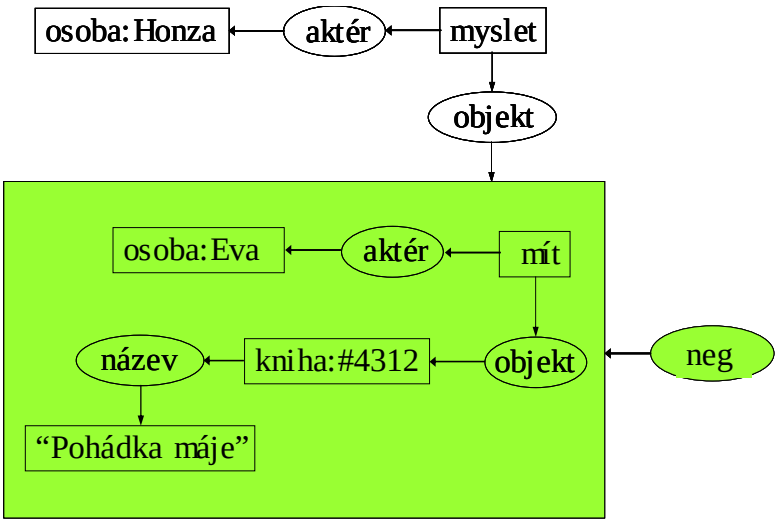
Na Obr. 7.6 a 7.7 jsou uvedeny pojmové grafy reprezentující složitější věty, resp. výroky: *Jeden Petr je ženatý, druhý, zvaný též Péťa, je svobodný* a *Honza si myslí, že Eva nemá knihu „Pohádka máje“* .

S pojmovými grafy lze provádět některé operace, které umožňují vytváření nových pojmových grafů:

- Restrikce vytvoří nový pojmový graf z původního grafu nahrazením vybraného pojmového uzlu uzlem reprezentujícím jeho specializaci (generický pojem může být nahrazen individuálním pojmem stejného typu (k typovému jménu se doplní jméno individua), typ může být nahrazen podtypem (jméno typu se přepíše jménem podtypu).
- Spojení dvou grafů lze provést, obsahují-li oba grafy identický pojmový uzel, např. uzel u_i. Spojení pak spočívá ve vytvoření nového grafu, ve kterém jsou k uzlu u_i připojeny všechny relační uzly z obou grafů původních.
- Zjednodušení je operace, která odstraňuje z pojmového grafu duplicitní relace; takové relace se mohou v pojmovém grafu objevit například po operaci spojení.



Obr. 7.6 Pojmový graf věty *Jeden Petr je ženatý, druhý, zvaný též Péťa, je svobodný*



Obr. 7.7 Pojmový graf reprezentující větu, kdy argumentem relace je výrok: *Honza si myslí, že Eva nemá knihu „Pohádka máje"*

Pojmové grafy mají stejnou vyjadřovací sílu jako predikátová logika 1. řádu. Operace s pojmovými grafy však neodpovídají pravidlům inference. Dva zásadní rozdíly jsou tyto:

- V predikátové logice lze odvodit výroky, které jsou sice pravdivé, ale jsou nesmyslné,
- V pojmových grafech lze vytvořit graf, který má smysluplný význam, nemusí však být pravdivý.

7.4 Strukturální schémata

Strukturální schémata

Strukturální schémata jsou určena pro reprezentace znalostí, které se často opakují. Obsahují významné lidské přístupy k myšlení, jako jsou očekávání, předpoklady, stereotypy, zvažování výjimek a neostrých hranic mezi třídami.

Strukturální schémata jsou jistým rozšířením síťových schémat, kdy jednotlivé uzly reprezentující objekty nebo pojmy jsou nahrazeny komplexnějšími strukturami, nazývanými *rámce*. Vyjadřovací síla, výhody a nevýhody těchto schémat jsou velmi podobné vyjadřovací síle, výhodám a nevýhodám síťových schémat, umožňují však efektivnější manipulaci s uloženými znalostmi.

Rámce obsahují položky, jejichž hodnotami mohou být číselná nebo symbolická data, ukazatele na jiné rámce, nebo odkazy na procedury, které s daty rámce manipulují. Ukazatele na jiné rámce a jména položek nahrazují hrany síťových schémat, a proto báze znalostí je tvořena pouze množinou rámců.

Typický rámec má následující strukturu:

Jméno rámce
- Ukazatele na nadřazené rámce
  - ako (a-kind-of)
  - instance (is-a)
- Významové položky
  - jméno položky
  - hodnota / implicitní (default) hodnota
  - restrikce (omezení hodnoty položky, např. číselným intervalem)
- Procedury
  - if-needed (je-li třeba)
  - if-added (je-li měněna hodnota významové položky)
  - if-removed (je-li zrušena hodnota významové položky)

Pro procedury *if-needed*, *if-added* a *if-removed* se běžně používá název démoni. Naznačuje se tak, že jde o "spící" procedury, které se aktivují pouze určitou, jménem naznačenou událostí.

Démon *if-needed* může být vyvolán například při požadavku na zjišťování vlastností děděných z nadřazených rámců, démon *if-added* může např. zjistit, zda zadávaná hodnota leží v povoleném intervalu a démon *if-removed* může např. testovat, zda se odstraněním položky neporuší bezespornost systému.

Příklad budování rámcové reprezentace znalostí:

Výchozí znalost: Existuje třída věcí, která se nazývají *auta*. Každé auto má čtyři kola, je poháněno motorem a jeho palivem je buď benzín nebo nafta. Výchozí rámec obsahuje své jméno, ukazatel na nadřazený rámec a tři významové položky se jménem položky, její hodnotou, resp. default hodnotou a restrikcí:

Jméno rámce:	auto	
Nadřazený rámec:	věc	
Kola	4	-
Pohon	motor	-
Palivo	?	{benzín, nafta}

Další znalost: V Itálii jsou vyráběna auta značky FIAT. Pro auta této značky se vytvoří nový rámec. Položky nadřazeného rámce (auto), jejichž hodnoty nejsou v rozporu s auty FIAT, lze dědit, a proto se v novém rámci nemusí vyskytovat:

Jméno rámce:	auto FIAT	
Nadřazený rámec:	auto	
Země výrobce:	Itálie	-

Další znalost: Jedním z typů značky FIAT je čtyřválcové benzínové auto PUNTO. Rámec pro toto auto pak může vypadat takto:

Jméno rámce:	FIAT PUNTO	
Nadřazený rámec:	auto FIAT	
Palivo	benzín	-
Počet válců	4	-

Další znalost: Podtypem aut FIAT PUNTO jsou auta PUNTO Sporting, která mají 16 ventilů, obsah 1.3 l a výkon 63 kW. Brzdový systém mají buď klasický nebo ABS. Pro tato auta se vytvoří další rámec.

Jméno rámce:	PUNTO-Sporting	
Nadřazený rámec:	FIAT PUNTO	
Počet ventilů	16	-
Obsah válců [l]	1.3	-
Výkon [kW]	63	-
Brzdový systém	?	{klasický,ABS}

Auto PUNTO Sporting vlastníka Xxxx má SPZ 1D0 1234 a má klasický brzdový systém:

Jméno rámce:	1D0 1234	
Nadřazený rámec:	PUNTO Sporting	
Vlastník	Xxxx	
Brzdový systém	klasický	-

Předpokládejme nyní možnou otázku na počet ventilů na válec. Pak lze první rámec doplnit o démona *if-needed*:

Jméno rámce:	auto	
Nadřazený rámec:	věc	
Kola	4	-
Pohon	motor	-
Palivo	?	{benzín, nafta}
Počet ventilů/válec	?	Zjisti počet ventilů, zjisti počet válců, spočítej podíl počet ventilů / počet válců

7.5 Procedurální schémata

Procedurální schémata

Jak již bylo zmíněno v úvodu, logická, síťová i strukturální schémata jsou deklarativní schémata, protože znalosti reprezentují především deklaracemi objektů / pojmů a jejich vzájemných vztahů.

Znalosti je však možné reprezentovat i zcela odlišným způsobem, a to jako množiny pravidel typu

if *P* then *A*

která se interpretují takto:

jestliže je splněna podmínka P pak je možné vykonat akci A

S pravidly jsou pak spojeny procedury umožňující vykonání jejich akčních částí, a proto se příslušná schémata reprezentace nazývají procedurální.

Procedurální schémata umožňují snadné přidávání a modifikaci znalostí i jejich hierarchické členění a především umožňují snadnou a efektivní manipulaci s těmito znalostmi. Při použití prostředků moderních logik umožňují i velmi efektivní práci s nejistotu informací, nejednoznačnou informací i s příčinností a dočasností. Nejsou však vhodným nástrojem pro sémantické sdružování příbuzných znalostí a dědičnost.

Obecným systémem, který pracuje s procedurálními schématy reprezentace znalostí (tj. s pravidly) je produkční systém. Tento systém má tři základní části:

- Bázi znalostí (pravidel)
- Bázi dat (pracovní paměť)
- Inferenční (odvozovací) mechanizmus

Produkční systém vychází z počátečního stavu báze dat a na tuto bázi aplikuje postupně pravidla z báze pravidel tak dlouho, dokud jde aplikovat nějaké pravidlo, nebo dokud báze dat neobsahuje požadovaná cílová data. V řadě kroků je však možné aplikovat současně více pravidel - aplikovatelná pravidla tvoří tzv. konfliktní množinu. Inferenční mechanizmus pak musí z této množiny vybrat k aplikaci pouze jediné pravidlo, například:

1. Podle čísla (priority) pravidla.
2. Podle časových známek objektů v podmínkové části.
3. Podle počtu objektů v podmínkové části.
4. Náhodně.

Pokud je zadaná u pravidel priorita, je přirozeně nutné tuto prioritu respektovat. V opačném případě a v případě, že konfliktní pravidla mají stejnou prioritu, lze kontrolovat časové známky objektů (jsou uloženy v bázi dat) v podmínkových částech pravidel – ke zpracování se vybere pravidlo s „nejmladšími“ objekty, což odpovídá lidským přístupům při řešení obecných problémů. Pokud ani toto nerozhodne o výběru pravidla, je dalším kritériem výběru počet objektů v podmínkových částech pravidel – vybere se pravidlo, které má největší počet objektů ve své podmínkové části, protože pravděpodobnost opětovného splnění podmínky u pravidel s více objekty je nižší, než pravděpodobnost opětovného splnění podmínky u pravidel s méně objekty. Není-li ani potom možné rozhodnout, vybere se jedno pravidlo náhodně.

x+y

Příklady práce produkčního systému (výběr pravidel podle priorit):

1. příklad (řazení):

Produkční pravidla:
1. ba → ab
2. ca → ac
3. cb → bc

Krok výpočtu	Báze dat	Konfliktní množina pravidel	Vybrané pravidlo
0	cbaca	{1, 2, 3}	1
1	cabca	{2}	2
2	acbac	{1, 3}	1
3	acabc	{2}	2
4	aacbc	{3}	3
5	aabcc	{ }	-

2. Příklad 2 (důkaz)

Produkční pravidla:
1. $p \wedge q \rightarrow goal$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow q$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $start \rightarrow v \wedge r \wedge q$

a) Výpočet řízen daty (*Data driven*):

Krok výpočtu	Báze dat	Konfliktní množina pravidel	Vybrané pravidlo
0	start	{6}	6
1	start,v,r,q	{6, 5}	5
2	start,v,r,q,s	{6, 5, 2}	2
3	start,v,r,q,s,p	{6, 5, 2, 1}	1
4	start,v,r,q,s,p,goal		

b) Výpočet řízen cílem (*Goal driven*):

Krok výpočtu	Báze dat	Konfliktní množina pravidel	Vybrané pravidlo
0	goal	{1}	1
1	goal,p,q	{1, 2, 3, 4}	2
2	goal,p,q,r,s	{1, 2, 3, 4, 5}	3
3	goal,p,q,r,s,w	{1, 2, 3, 4, 5}	4
4	goal,p,q,r,s,w,t,u	{1, 2, 3, 4, 5}	5
5	goal,p,q,r,s,w,t,u,v	{1, 2, 3, 4, 5, 6}	6
6	goal,p,q,r,s,w,t,u, start		

7.6 Shrnutí

Shrnutí

V kapitole byly vysvětleny principy reprezentace znalostí a byly zde popsány čtyři základní schémata: logické, síťové, strukturální a procedurální. Byly zde uvedeny příklady reprezentací pomocí jednotlivých schémat a diskutovány jejich silné i slabé stránky.

7.7 Kontrolní otázky

Kontrolní otázky

1. Jaký je princip logického schématu reprezentace znalostí?
2. Jaký je princip síťových schémat reprezentace znalostí?
3. Jaký je princip strukturálního schématu reprezentace znalostí?
4. Jaký je princip procedurálního schématu reprezentace znalostí?

8 STROJOVÉ UČENÍ

Cílem této kapitoly je seznámit studenty se základními přístupy ke strojovému učení. Postupně jsou zde popsány a vysvětleny principy vybraných metod učení s učitelem, učení bez učitele a posilovaného učení.

10 hod

8.1 Úvodní informace

Úvodní informace

Strojové učení (*machine learning*) je schopnost inteligentního systému měnit své znalosti tak, že příště bude vykonávat stejnou nebo podobnou úlohu účinněji (efektivněji). Strojové učení lze rozdělit do tří základních skupin:

- Učení s učitelem (*supervised learning*)
- Učení bez učitele (*unsupervised learning*)
- Posilované (motivované) učení (*reinforcement learning*)

Učení s učitelem spočívá v tom, že pro každý krok učení je známá požadovaná odezva a systém je tak okamžitě informován o aktuálním hodnocení poslední akce. Učení se provádí na tzv. trénovací množině příkladů *T*, kdy každý příklad je představován množinou vstupních hodnot (vstupním vektorem) a množinou správných/požadovaných výstupních hodnot (výstupním vektorem):

$$T = \left\{ \left(\vec{i}_1, \vec{d}_1\right), \left(\vec{i}_2, \vec{d}_2\right), \ldots, \left(\vec{i}_P, \vec{d}_P\right) \right\},$$

kde značí

$\vec{i}_i$ … *i*-tý vstupní vektor

$\vec{d}_i$ … *i*-tý výstupní vektor

Prvky vstupních i výstupních vektorů mohou nabývat číselných i symbolických hodnot.

V dalším textu budou popsány principy následujících metod učení s učitelem:

- Rozhodovací stromy (*decision trees*), kap. 8.2,
- Prohledávání prostoru verzí (*version-space search*), kap. 8.2,
- Rozpoznávání obrazů (*pattern recognition*), kap. 9.

Učení bez učitele spočívá v nalezení podobností ve vstupních vektorech trénovací množiny – žádná podpůrná informace není obvykle dostupná. Při učení se hledají shluky obrazů (*clustering*) představovaných mnohorozměrnými číselnými vstupními vektory trénovací množiny *T*:

$$T = \left\{ \vec{i}_1, \vec{i}_2, \ldots, \vec{i}_P \right\}$$

V kapitole 8.3 bude popsán princip jednoduché shlukovací metody, která hledá předepsaný počet shluků.

Posilované učení se od učení s učitelem odlišuje tím, že systém provádí akce, po jejichž provedení dostává „odměnu“ – ta může být pozitivní i negativní a navíc různě veliká. Podstatné je, že každá akce ovlivňuje i následující akce a systém musí maximalizovat výsledný součet všech odměn. V některých případech, například u deskových her, jsou běžné akce/tahy hodnoceny nulou a pouze

konečné tahy jsou hodnoceny odlišně (např. +1 za výhru, -1 za prohru) a systém musí sám zjistit, které jeho tahy byly dobré a které naopak špatné. Princip tohoto učení bude stručně popsán v kapitole 8.4.

Pozn.: Učení je přirozenou vlastností umělých neuronových sítí, jejichž popis je jednou z hlavních náplní magisterského předmětu SFC (SoftComputing).

8.2 Učení s učitelem

Učení s učitelem

V této kapitole budou naznačeny principy několika metod učení s učitelem: metody rozhodovacích stromů a tří metod prohledávání prostoru verzí. Všechny tyto metody pracují s vektory, které nabývají symbolických hodnot (případné číselné hodnoty jsou rovněž chápány jako symbolické) a jsou založené na předpokladu, že každá hypotéza, která vyhovuje dostatečně velké množině trénovacích příkladů, bude vyhovovat i dalším příkladům.

8.2.1 Rozhodovací stromy

Rozhodovací stromy

Rozhodovací stromy slouží ke klasifikaci objektů na základě hodnot jejich atributů / vlastností. Jsou vytvářeny ze známé množiny příkladů a jsou známou metodou používanou při tzv. dolování dat, resp. získávání znalostí z databází (*data mining, or knowledge-discovery in databases*).

Princip metody bude nejlépe patrný z jednoduchého praktického příkladu. Předpokládejme, že máme množinu čtrnácti příkladů daných jednotlivými řádky následující tabulky:

	Příjem	Dluh	Historie úvěrů	Ručení	Risk
1	< 15	vysoký	špatná	žádné	vysoký
2	15 – 35	vysoký	neznámá	žádné	vysoký
3	15 – 35	nízký	neznámá	žádné	přiměřený
4	< 15	nízký	neznámá	žádné	vysoký
5	> 35	nízký	neznámá	žádné	nízký
6	> 35	nízký	neznámá	adekvátní	nízký
7	< 15	nízký	špatná	žádné	vysoký
8	> 35	nízký	špatná	adekvátní	přiměřený
9	> 35	nízký	dobrá	žádné	nízký
10	> 35	vysoký	dobrá	adekvátní	nízký
11	< 15	vysoký	dobrá	žádné	vysoký
12	15 – 35	vysoký	dobrá	žádné	přiměřený
13	> 35	vysoký	dobrá	žádné	nízký
14	15 – 35	vysoký	špatná	žádné	vysoký

První čtyři atributy (sloupce 2 – 5) v každém řádku jsou podmínkové atributy a představují vstupní vektory, atribut v posledním sloupci je rozhodovací atribut a představuje jednoprvkový výstupní vektor. Atributy *Příjem*, *Historie úvěrů* a *Risk* jsou tříhodnotové, zbývající dva atributy (*Dluh* a *Ručení*) mají pouze dvě hodnoty.

Předpokládejme, že na základě těchto příkladů má bankovní úředník stanovit hodnotu rozhodovacích atributů pro nové klienty, zná-li hodnoty podmínkových atributů těchto klientů.

Zvolená úloha se zdá být na první pohled velmi jednoduchá. Je nutné si však uvědomit, že pro jednu kombinaci rozhodovacího atributu by měla úplná tabulka celkem 36 řádků (součin počtu hodnot jednotlivých podmínkových atributů $3 \cdot 2 \cdot 2 \cdot 3 = 36$) a že 3-hodnotový rozhodovací atribut tak vede k 3^{36} možným různým tabulkám ($3^{36} \cong 1.5 \cdot 10^{17}$)! V reálné praxi pak podobné tabulky mají desítky i stovky vícehodnotových atributů a statisíce až miliony řádků.

Základní algoritmus rozhodovacího stromu má dva vstupní parametry (množinu všech příkladů MP a množinu všech podmínkových atributů MA) a vrací úplný rozhodovací strom.

Algoritmus Decision tree

Algoritmus Decision_tree

1. Je-li množina příkladů MP prázdná, vrať listový uzel označený „hodnotou“ *Neúspěch*.
2. Mají-li všechny příklady množiny příkladů MP stejnou hodnotu rozhodovacího atributu, vrať listový uzel označený touto hodnotou, jinak pokračuj.
3. Je-li množina atributů MA prázdná, vrať listový uzel označený disjunkcí všech hodnot, které nabývají příklady v množině příkladů MP, jinak pokračuj.
4. Vyber atribut A_i, odstraň jej z množiny atributů MA a učiň jej kořenem aktuálního stromu (nechť MA_{-i} je množina atributů MA bez atributu A_i).
5. Pro každou hodnotu H_j vybraného atributu A_i:
   - Vytvoř novou větev stromu označenou hodnotou H_j a podmnožinou příkladů MP_{AiHj}, která obsahuje ty prvky původní množiny příkladů MP, jejichž atribut A_i má hodnotu H_j,
   - volej rekurzivně algoritmus Decision_tree s parametry MP_{AiHj} a MA_{-i},
   - připoj vrácený strom / list k této větvi.

I když se zdá algoritmus Decision_tree velmi jednoduchý, skrývá v sobě jeden zásadní problém, kterým je výběr atributu v bodu 4. Nevhodné výběry vedou k hlubokým a neefektivním vyhledávacím stromům (Obr. 8.1) a to přestože optimální strom může být zcela jednoduchý (Obr. 8.2).

Algoritmus ID3

Algoritmus ID3

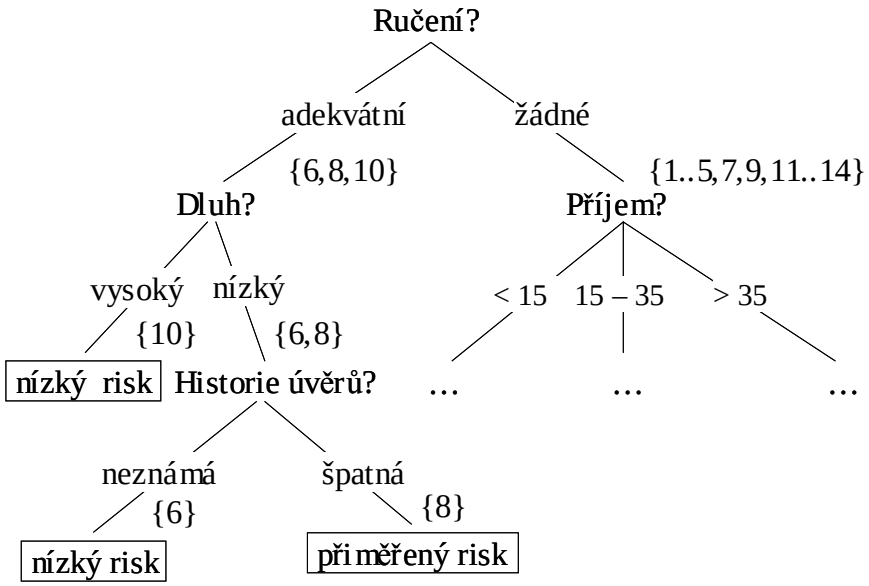
Nejznámější algoritmus, řešící současně problém výběru atributu v kroku 4, je algoritmus známý pod zkratkou ID3 (*Induction of Decision Trees*). Výběr atributu v tomto algoritmu je založen na maximalizaci informačního zisku.

Z teorie informace je známo, že entropie, nebo-li míra neuspořádanosti zprávy (množiny příkladů MP) s *n* možnými odpověďmi (tj. s *n* možnými hodnotami

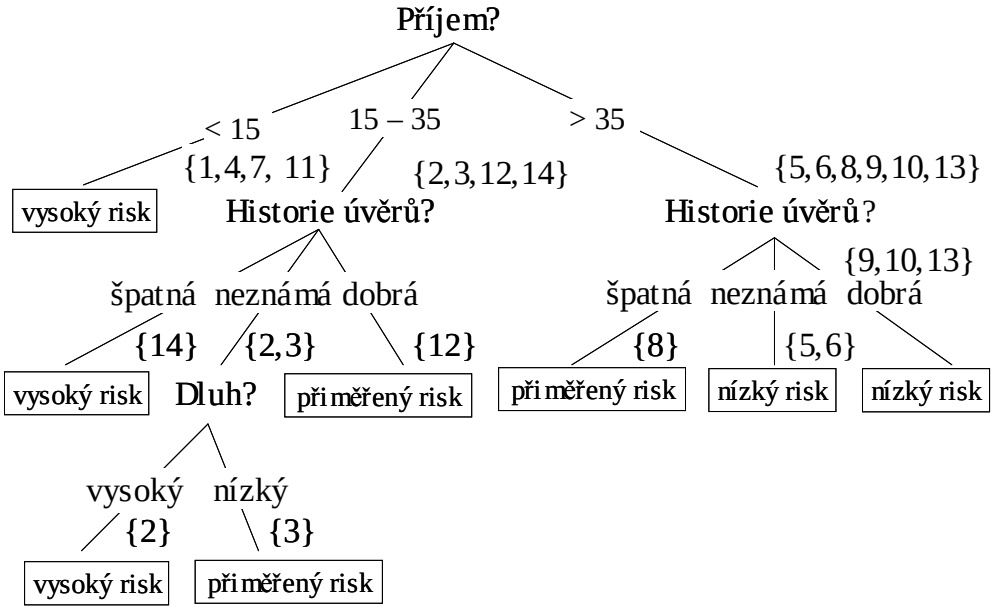
rozhodovacího atributu) $\{o_1, o_2, \ldots, o_n\}$, jejichž pravděpodobnosti výskytu jsou $p(o_i)$, je dána výrazem:

$$E(MP) = \sum_{i=1}^{n} - p(o_i)\log_2(p(o_i)) \qquad [bit]$$

Pozn.: Maximální hodnota entropie pro n možných odpovědí je dána výrazem $E_{max}(M) = \log_2 n$. Této hodnoty se dosáhne, jsou-li všechny odpovědi stejně pravděpodobné - pro dvě odpovědi je hodnota entropie 1 bit (tj. 1 bit je potřebný k odstranění neurčitosti zprávy), pro čtyři odpovědi je hodnota entropie 2 bity, apod. Pokud je pravděpodobnost některé odpovědi nulová, resp. jedničková, přispívá tato odpověď k entropii nulovou hodnotou ($0 \cdot \log_2 0 = 0$, $1 \cdot \log_2 1 = 0$).



Obr. 8.1 Příklad neúplného rozhodovacího stromu



Obr. 8.2 Úplný a optimální rozhodovací strom

Atribut *A*, který má *m* různých hodnot, rozděluje množinu příkladů MP do *m* podmnožin, z nichž každá obsahuje MP_i příkladů. Entropie, tj. očekávaná informace potřebná k dokončení stromu, bude-li jeho kořenem atribut A, je pak dána výrazem

$$E(A) = \sum_{i=1}^{m} \frac{MP_i}{MP} E\left(MP_i\right)$$

a odpovídající informační zisk vztahem

$$zisk\left(A\right) = E(MP) - E(A)$$

Vybraným atributem podle bodu 4 algoritmu je pak atribut, který přináší největší informační zisk.
Podobně se postupuje v každém nelistovém uzlu – při rekurzivním volání se atribut *A* z množiny atributů odstraní a za množinu příkladů MP se považuje podmnožina MP_i (viz bod 5 algoritmu).

Pro předchozí příklad (tabulku) platí:

MP	=	{1,2,…,14}
p(vysoký risk)	=	6/14
p(přiměřený risk)	=	3/14
p(nízký risk)	=	5/14

a entropie tabulky pak je

$$E\left(tabulka\right) = -\frac{6}{14}\log_2\left(\frac{6}{14}\right) - \frac{3}{14}\log_2\left(\frac{3}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 1.531$$

Informační zisk s respektováním odpovědí na atribut *Příjem* (tři různé hodnoty atributu <15, 15-35, >35) se vypočítá takto:

MP_1= {1,4,7,11}, MP_2 = {2,3,12,14}, MP_3 = {5,6,8,9,10,13}

$$E\left(prijem\right) = \frac{4}{14}\left(-\frac{4}{4}\log_2\left(\frac{4}{4}\right)\right) + \frac{4}{14}\left(-\frac{2}{4}\log_2\left(\frac{2}{4}\right) - \frac{2}{4}\log_2\left(\frac{2}{4}\right)\right) +$$
$$+\frac{6}{14}\left(-\frac{5}{6}\log_2\left(\frac{5}{6}\right) - \frac{1}{6}\log_2\left(\frac{1}{6}\right)\right) = 0.565$$

$$zisk\left(prijem\right) = E(tabulka) - E(prijem) = 0.966$$

Pozn.: Odpovědi (tj. hodnoty rozhodovacího atributu *Risk*) na všechny čtyři příklady podmnožiny MP1 jsou stejné (vysoký risk), v podmnožině MP2 se vyskytují dvě stejně pravděpodobné odpovědi (2 * vysoký risk a 2 * přiměřený risk) a v podmnožině MP3 se vyskytují dvě odpovědi s různou pravděpodobností (1 * přiměřený risk, 5 * nízký risk).

Informační zisk s respektováním odpovědí pro jiné atributy se vypočítají podobně:

$$zisk\left(dluh\right) = 0.063\ bitů$$
$$zisk\left(historie\ \ uveru\right) = 0.266\ bitů$$
$$zisk\left(ruceni\right) = 0.206\ bitů$$

Nejvyšší zisk přináší výběr atributu *Příjem*, proto se tento atribut vybere za kořen celého stromu.

Pro všechny čtyři příklady nové množiny příkladů MP = MP_1 jsou hodnoty rozhodovacího atributu stejné, proto algoritmus vrací podle bodu 2 listový uzel označený touto hodnotou (vysoký risk).

V nové množině příkladů MP = MP_2 = {2,3,12,14} se vyskytují dvakrát dvě různé odpovědi (přiměřený a vysoký risk) a nová množina atributů obsahuje atributy {*Dluh, Historie úvěrů, Ručení*}. Entropie této množiny příkladů je dána vztahem

$$E(MP = MP_2) = -\frac{2}{4}\log_2\left(\frac{2}{4}\right) - \frac{2}{4}\log_2\left(\frac{2}{4}\right) = -\log_2\left(\frac{2}{4}\right) = 1$$

Informační zisk s respektováním odpovědí na atribut *Dluh* (dvě různé hodnoty atributu: vysoký, nízký) se vypočítá takto:

MP_1= {2,12,14}, MP_2 = {3}

$$E(dluh) = \frac{1}{4}\left(-\frac{1}{1}\log_2\left(\frac{1}{1}\right)\right) + \frac{3}{4}\left(-\frac{1}{3}\log_2\left(\frac{1}{3}\right) - \frac{2}{3}\log_2\left(\frac{2}{3}\right)\right) = 0.689$$

$$zisk(dluh) = E(MP = MP2) - E(dluh) = 0.311$$

a podobně se vypočítá informační zisk s respektováním odpovědí pro jiné atributy:

$$zisk(historie\ \ uveru) = 0.5$$
$$zisk(ruceni) = 0$$

Nejvyšší zisk přináší výběr atributu *Historie úvěru*, proto se tento atribut vybere za počátek („kořen“) dalších větví.

Dvě větve MP_1 = {14} a MP_3 = {12} vedou přímo k listovým uzlům (vysoký a přiměřený risk), třetí obsahuje dva příklady, které mají k rozdílné hodnoty rozhodovacího atributu MP_2 ={2,3}.

V nové množině příkladů MP = MP_2 = {2,3} se vyskytují dvě různé odpovědi (vysoký a přiměřený). Nová množina atributů obsahuje atributy {*Dluh* a *Ručení*}. Entropie a informační zisky vypočítáme již známým postupem:

$$E(MP = MP_2) = -\frac{1}{2}\log_2\left(\frac{1}{2}\right) - \frac{1}{2}\log_2\left(\frac{1}{2}\right) = -\log_2\left(\frac{1}{2}\right) = 1$$
$$zisk(dluh) = 0.5$$
$$zisk(ruceni) = 0$$

Nejvyšší zisk přináší výběr atributu *Dluh*, a proto se tento atribut vybere za počátek („kořen“) posledních dvou větví. Ty vedou k listovým uzlům a tím je hlavní větev počínající uzlem *Příjem* = 15-35 dokončena.

Podobně se postupuje při budování poslední větve uzlu *Příjem* > 35:

MP = MP_3 = {5,6,8,9,10,13}

$$E(MP = MP_3) = -\frac{1}{6}\log_2\left(\frac{1}{6}\right) - \frac{5}{6}\log_2\left(\frac{5}{6}\right) = 0.65$$

$$zisk(dluh) = 0.109$$
$$zisk(historie\ \ uveru) = 0.65$$
$$zisk(ruceni) = 0.191$$

Výběr atributu Historie úvěru vede ke třem novým větvím, které jsou všechny zakončeny ohodnocenými listovými uzly – tím je proces budování stromu ukončen.

Pozn.: Poznamenejme, že tvorba rozhodovacího stromu spočívá v prohledávání stavového prostoru, jehož uzly jsou představovány jednotlivými atributy a větve hodnotami těchto atributů. Algoritmus ID3 je pak praktickou aplikací metody *Greedy search*.

8.2.2 Prohledávání prostoru verzí

Prohledávání prostoru verzí

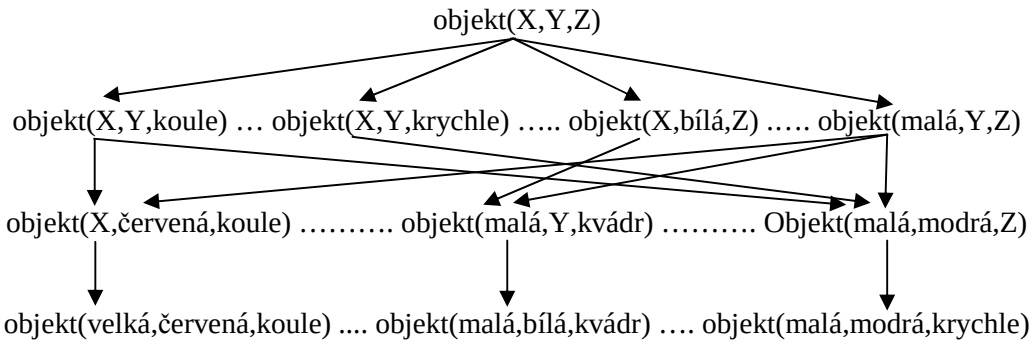
Prohledávání prostoru verzí představuje soubor metod učení významům pojmů, resp. hypotéz, na základě pozitivních a negativních příkladů (jde o aplikaci přístupů, které se používají při učení malých dětí). Při učení se hledá takový popis daného pojmu nebo hypotézy, který zahrnuje všechny pozitivní příklady a vylučuje všechny negativní příklady z trénovací množiny příkladů.

Princip metod bude zřejmý z jejich aplikací na jednoduchý příklad:

Předpokládejme obecný pojem *objekt*, u kterého jsme schopni určit velikost, barvu a tvar. Pro jednoduchost dále předpokládejme pouze dvě různé velikosti, tři různé barvy a tři různé tvary:

objekt(velikost,barva,tvar),
velikost = {velká, malá},
barva = {červená,bílá,modrá},
tvar = {koule,kvádr,krychle}.

Úplný prostor verzí pro všechny možné objekty výše uvedených vlastností je ukázán na Obr 8.3.



Obr. 8.3 Prostor verzí

Nejobecnějším pojmem je zde pojem objekt(X,Y,Z), méně obecnými (více specifikovanými) pojmy jsou zde pojmy se dvěma proměnnými (například objekt(X,bílá,Z)), ještě méně obecnými (ještě více specifikovanými) pojmy jsou zde pojmy s jednou proměnnou (například objekt(malá,Y,kvádr)) až konečně nejvíce specifickými pojmy jsou zde pojmy bez proměnných (například

objekt(velká,červená,koule)). Poznamenejme, že z opačného směru pohledu jsou pojmy s jednou proměnnou méně specifikovanými (obecnějšími) pojmy a pojmy se dvěma proměnnými ještě méně specifikovanými (ještě obecnějšími) pojmy.

Při prohledávání prostoru verzí se používají dvě operace: zobecňování (nahrazení konstanty proměnnou) a specializace (nahrazení proměnné konstantou).

Předpokládejme nyní, že umělý systém se bude učit, co znamená pojem *míč*. Nechť trénovací příklady jsou tyto:

Příklad	kladný/záporný
1. objekt(malá,červená,koule)	kladný
2. objekt(malá,modrá,kvádr)	záporný
3. objekt(velká,červená,koule)	kladný
4. objekt(velká,červená,krychle)	záporný
5. objekt(malá,modrá,koule)	kladný
6. objekt(malá,bílá,kvádr)	záporný

Existují tři přístupy/metody učení prohledáváním prostoru verzí: od specifického k obecnějšímu pojmu (*specific to general search*), od obecného k více specifickému pojmu (*general to specific search*) a spojení obou předchozích přístupů (*candidate elimination*).

Specific to general search

Algoritmus Specific_to_general_search:

- Vytvoř dvě prázdné množiny *S* a *N*.
- Ulož do množiny *S* první kladný příklad.
- Pro každý další příklad *p* z trénovací množiny:
  - Je-li *p* kladným příkladem, pak pro každý pojem $s \in S$:
    - Jestliže pojem *s* nelze unifikovat s příkladem *p*, pak jej nahraď jeho nejvíce specifickým zobecněním, které lze unifikovat s příkladem *p*,
    - odstraň z *S* všechny pojmy *s*, které jsou více obecné, než jiné pojmy v *S*,
    - odstraň z *S* všechny pojmy *s*, které lze unifikovat s některým pojmem v *N*.
  - Je-li p záporným příkladem, pak:
    - Odstraň z *S* všechny pojmy *s*, které lze unifikovat s příkladem *p*,
    - přidej příklad p do *N*.

Ukázka práce algoritmu Specific_to_general_search na příkladu:

	S = {}, N = {}
1. p = objekt(malá,červená,koule)	S = {objekt(malá,červená,koule)}
2. p = objekt(malá,modrá,kvádr)	N = {objekt (malá,modrá,kvádr)}
3. p = objekt(velká,červená,koule)	S = {objekt(X,červená,koule)}
4. p = objekt(velká,červená,krychle)	N = {objekt(malá,modrá,kvádr), objekt(velká,červená,krychle)}
5. p = objekt(malá,modrá,koule)	S = {objekt (X,Y,koule)}
6. p = objekt(malá,bílá,kvádr)	N = {objekt(malá,modrá,kvádr), objekt(velká,červená,krychle), objekt(malá,bílá,kvádr)}

Výsledek: nejvíce specifické zobecnění je **objekt(X,Y,koule)**.

General to specific

Algoritmus General_to_specific_search:

- Vytvoř dvě prázdné množiny *G* a *P* a ulož do *G* nejobecnější pojem.
- Pro každý další příklad *p* z trénovací množiny:
  - Je-li *p* záporným příkladem, pak pro každý pojem $g \in G$
    - Jestliže pojem *g* lze unifikovat s příkladem *p*, pak jej nahraď jeho nejobecnější specializací, kterou nelze unifikovat s příkladem *p*,
    - odstraň z *G* všechny pojmy *g*, které jsou více specializované, než jiné pojmy v *G*,
    - odstraň z *G* všechny pojmy *g*, které nelze unifikovat s některým pojmem v *P*.
  - Je-li *p* kladným příkladem, pak:
    - Odstraň z *G* všechny pojmy *g*, které nelze unifikovat s příkladem *p*,
    - přidej příklad *p* do *P*.

Ukázka práce algoritmu General_to_specific_search na příkladu:

	G = { objekt(X,Y,Z)}, P = {}
1. p = objekt(malá,červená,koule)	P = { objekt(malá,červená,koule)}
2. p = objekt(malá,modrá,kvádr)	G = { objekt(velká,Y,Z), objekt(X,červená,Z), objekt(X,bílá,Z), objekt(X,Y,koule), objekt(X,Y,krychle)}
	⇒ G = { objekt(X,červená,Z), objekt(X,Y,koule)}
3. p = objekt(velká,červená,koule)	P = { objekt(malá,červená,koule), objekt(velká,červená,koule)}
4. p = objekt(velká,červená,krychle)	G = { objekt(malá,červená,Z), objekt(X,červená,kvádr), objekt(X,červená,koule), objekt(X,Y,koule)}
	⇒ G = { objekt(X,Y,koule)}
5. p = objekt(malá,modrá,koule)	P = { objekt(malá,červená,koule), objekt(velká,červená,koule), objekt(malá,modrá,koule)}
6. p = objekt(malá,bílá,kvádr)	G = { objekt(X,Y,koule)}

Výsledek: nejvíce obecná specializace je **objekt(X,Y,koule)**

Candidate eliminations

Algoritmus Candidate_eliminations

- Vytvoř dvě prázdné množiny *G* a *S* a ulož do *G* nejobecnější pojem.
- Ulož do množiny *S* první kladný příklad.
- Pro každý další příklad *p* z trénovací množiny:
  - Je-li *p* kladným příkladem:
    - Odstraň z *G* všechny pojmy *g*, které nelze unifikovat s příkladem *p*,
    - pro každý pojem $s \in S$:
      - Jestliže pojem *s* nelze unifikovat s příkladem *p*, pak jej nahraď jeho nejvíce specifickým zobecněním, které lze unifikovat s příkladem *p*,
      - odstraň z *S* všechny pojmy *s*, které jsou více obecné, než jiné

pojmy v *S*,
- odstraň z *S* všechny pojmy *s*, které nejsou více specifické, než některé pojmy v *G*.

- Je-li *p* záporným příkladem:
  - Odstraň z *S* všechny pojmy *s*, které lze unifikovat s příkladem *p*,
  - pro každý pojem $g \in G$:
    - Jestliže pojem *g* lze unifikovat s příkladem *p*, pak jej nahraď jeho nejvíce zobecněnou specializací, kterou nelze unifikovat s příkladem *p*,
    - odstraň z *G* všechny pojmy *g*, které jsou více specifické než jiné pojmy v *G*,
    - odstraň z *G* všechny pojmy *g*, které nejsou více obecné než některé pojmy v *S*.
- Jestliže *G* = *S* a obě množiny přitom obsahují jediný pojem, pak výsledkem učení je právě tento pojem.

Ukázka práce algoritmu Candidate_elimination na příkladu:

G = {objekt(X,Y,Z)}, S = {}

1. p = objekt(malá,červená,koule) S = {objekt(malá,červená,koule)}
2. p = objekt(malá,modrá,kvádr) G = {objekt(velká,Y,Z),
   objekt(X,červená,Z),
   objekt(X,bílá,Z),
   objekt(X,Y,koule),
   objekt(X,Y,krychle)}
   ⇒ G = {objekt(X,červená,Z),
   objekt(X,Y,koule)}
3. p = objekt(velká,červená,koule) S = {objekt(X,červená,koule)}
4. p = objekt(velká,červená,krychle) G = {objekt(malá,červená,Z),
   objekt(X,červená,kvádr),
   objekt(X,červená,koule),
   objekt(X,Y,koule)}
   ⇒ G = {objekt(X,Y,koule)}
5. p = objekt(malá,modrá,koule) S = {objekt(X,Y,koule)}
6. p = objekt(malá,bílá,kvádr) G = {objekt(X,Y,koule)}

G = S a obě množiny obsahují jediný pojem: **objekt(X,Y,koule)**

Pozn.: Všimněte si, že výsledný popis je u všech algoritmů stejný. Tento popis akceptuje všechny kladné příklady a současně vylučuje všechny záporné příklady trénovací množiny!

8.3 Učení bez učitele

Učení bez učitele

Učení bez učitele spočívá v hledání podobností mezi příklady trénovací množiny a v zařazování (klasifikaci) příkladů s podobnými charakteristikami do skupin (shluků, tříd). Umělý systém přitom nedostává žádnou informaci o správnosti klasifikace a tudíž ani o průběhu svého učení – jedinou informací může být počet shluků, do kterých má příklady z trénovací množiny klasifikovat.

Příklady trénovací množiny jsou prakticky vždy představované číselnými vektory příznaků klasifikovaných dějů či objektů. Metody učení bez učitele jsou pak založeny na předpokladu, že tyto vektory, resp. jejich koncové body, tvoří

v příslušném (*n*-rozměrném) „obrazovém“ prostoru shluky.

Výše zmíněné shluky koncových bodů vektorů však mohou představovat velmi rozmanité *n*-rozměrné útvary (například v několika souřadnicích může jít o zcela kompaktní shluky, zatímco v jiných souřadnicích mohou být tyto body i značně rozptýleny) a žádná metoda nemůže proto být zcela univerzální.

V dalším textu bude popsán princip nejznámější a nejpoužívanější metody, která klasifikuje příklady (číselné vektory) trénovací množiny příkladů do předem daného počtu *k* shluků, metody/algoritmu s názvem *k–means clustering*.

Algoritmus zařadí vektor do toho shluku k jehož středu/těžišti má nejkratší vzdálenost a učí se následujícím postupem: Nejprve náhodně vybere *k* vektorů z trénovací množiny a považuje je za středy shluků. Dále přiřadí všechny vektory trénovací množiny k příslušným shlukům. Poté přepočítá středy shluků, znovu přiřadí všechny vektory trénovací množiny k příslušným shluků, atd. Učení končí, až všechny vektory zařazuje do stále stejných shluků.

Pozn.: Je zřejmé, že na výsledek učení má vliv použitá metrika – Euklidovská, Hammingova, ap.

K–means clustering

Algoritmus k–means clustering:

1. Inicializuj *k* prototypů (náhodně vybraných, ale různých vektorů z trénovací množiny: $\vec{w}_j = \vec{x}_p, \quad j \in \langle 1, k \rangle, \;\; p \in \langle 1, P \rangle$).
2. Každý vektor $\vec{x}_p$ z trénovací množiny přiřaď do shluku C_j $j \in \langle 1, k \rangle$, jehož prototyp $\vec{w}_j$ má od vektoru $\vec{x}_p$ nejmenší vzdálenost:

$$\left|\vec{x}_p - \vec{w}_j\right| \le \left|\vec{x}_p - \vec{w}_i\right| \qquad i, j \in \langle 1, k \rangle$$

3. Pro každý shluk C_j $j \in \langle 1, k \rangle$ přepočítej prototyp $\vec{w}_j$ tak, aby byl těžištěm koncových bodů všech vektorů, které jsou k tomuto shluku právě přiřazené (nechť n_j je počet těchto vektorů):

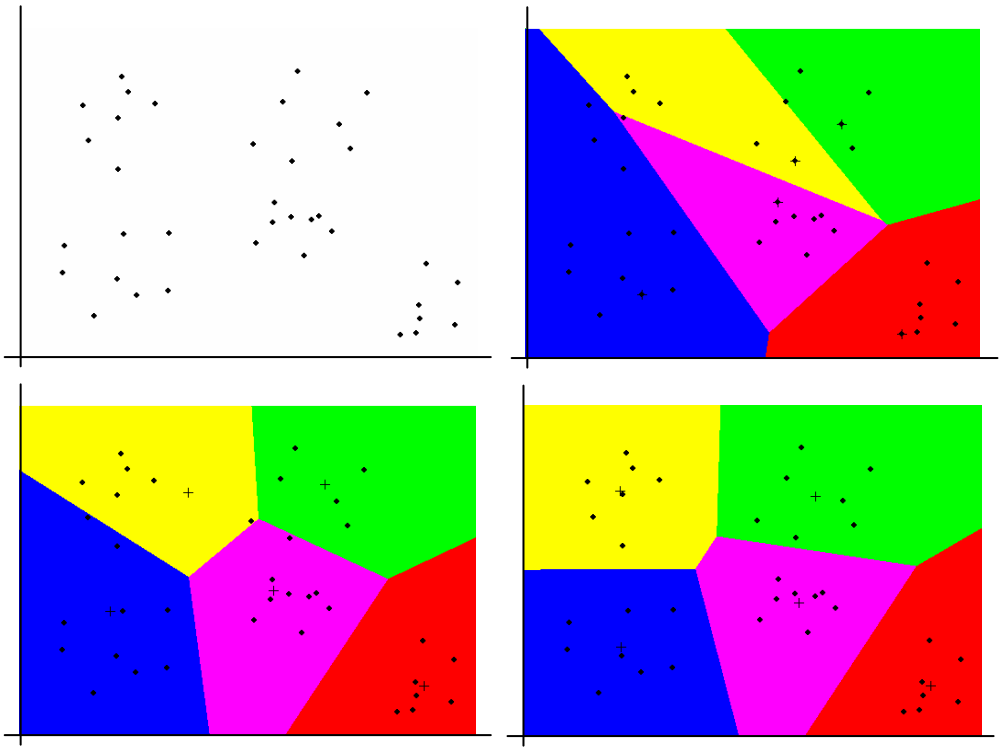
$$\vec{w}_j = \frac{\sum_{\vec{x}_i \in C_j} \vec{x}_i}{n_j}$$

4. Vypočítej „chybu“ aktuálního stavu shlukování (součet „chyb“ všech shluků, které jsou dány součty čtverců vzdáleností všech vektorů jednotlivých shluků od středů těchto shluků):

$$E = \sum_{j=1}^{k} \sum_{\vec{x}_i \in C_j} \left|\vec{x}_i - \vec{w}_j\right|^2$$

5. Pokud „chyba“ *E* klesla, nebo pokud byl některý vektor přiřazen k jinému shluku, vrať se na bod 2.

Na obrázku 8.4 je pro názornost ukázána trénovací množina dvourozměrných číselných vektorů (vlevo nahoře), počáteční náhodná volba pěti středů a následné rozdělení vektorů do shluků (vpravo nahoře), stav během učení (vlevo dole) a výsledek po ukončení učení (vpravo dole).

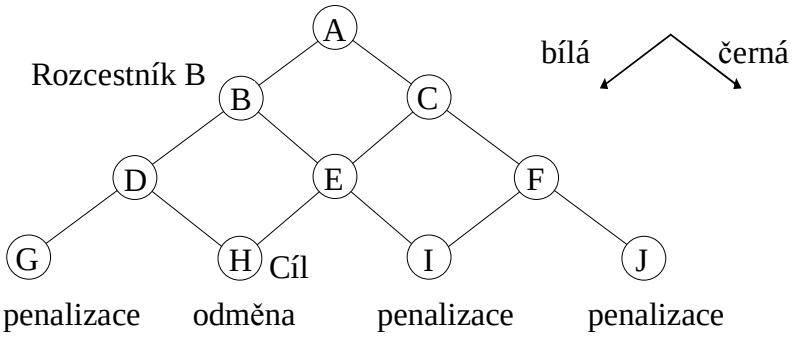


Obr. 8.4 Příklad učení algoritmem *k – means clustering*

8.4 Posilované učení

Posilované učení

Princip posilovaného učení bude zřejmý z následujícího příkladu: Uvažujme jednoduchý graf (Obr. 8.5), ve kterém z každého uzlu vedou maximálně dvě cesty a hledejme (optimální) cestu z počátečního uzlu *A* do cílového uzlu *H*.



Obr. 8.5 Graf pro naznačení principu posilovaného učení

Každý uzel (rozcestník) obsahuje schránku s černými a bílými kameny, které se využívají k určení směru cesty. Na začátku učení obsahuje schránka každého rozcestníku stejný počet černých a bílých kamenů ($N_{bílé} = N_{černé}$).

V průběhu učení jsou opakovaně prováděny „procházky“ z počátečního místa A, přičemž pravděpodobnosti pohybu na každém rozcestníku jsou dány výrazy:

$$p_{vlevo} = \frac{N_{bílé}}{N_{bílé} + N_{černé}}$$

$$p_{vpravo} = \frac{N_{černé}}{N_{bílé} + N_{černé}}$$

Na konci každé procházky se vyhodnotí cesta takto:

- Cesta vedla do H ⇒ odměna (přidání kamenů odpovídajících barev do schránek všech rozcestníků na cestě příslušné procházky)
- Cesta vedla do G, I, J ⇒ penalizace (odebrání kamenů odpovídajících barev ze schránek všech rozcestníků na cestě příslušné procházky)

Pravděpodobnost výběru optimální cesty do H se tak postupně zvyšuje, pravděpodobnost všech ostatních cest se naopak snižuje. Po naučení vybírá umělý systém cestu pouze porovnáním počtu kamenů – je-li ve schránce rozcestníku více bílých kamenů, pokračuje levou cestou a naopak.

V reálných případech je situace komplikovanější. Například při hledání cesty bludištěm může existovat z každého místa více cest a systém se může i vracet do míst (uzlů), kterými již dříve procházel. Dále si naznačíme principy tří metod, které se používají pro učení v těchto případech: pasivní metody *ADP learning* (*adaptive dynamic programming*), pasivní metody *TD learning* (*temporal difference*) a aktivní metody *Q learning*.

Předpokládejme pro jednoduchost desku s 3x3 poli, která má dva cílové stavy: Stav (3,3) s ohodnocením +1 a stav (2,3) s ohodnocením -1 (Obr. 8.6). Dále předpokládejme, že na desce nejsou žádné překážky, a že možné tahy jsou pouze na sousední políčka ve svislém nebo vodorovném směru.

	1	2	3
1			
2			
3		–1	+1

Obr. 8.6. Příklad desky

8.4.1 Metoda ADP learning

Metoda ADP learning

Metoda ADP learning ohodnocuje necílová políčka tak, aby optimální cesta z libovolného políčka do cílového políčka (3,3) vedla přes políčka s postupně se zvyšujícím ohodnocením.

Nejprve se provede řada náhodných procházek z daného výchozího místa (například z políčka (1,1)), které končí v některém cílovém políčku ((3,2) nebo (3,3)). Předpokládejme, že zvolená strategie pohybu spočívá v náhodném výběru směru, daném rovnoměrným rozložením pravděpodobnosti.

Dále pro jednoduchost předpokládejme pouze čtyři náhodné procházky:

1. $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3)$
2. $(1,1) \rightarrow (1,2) \rightarrow (2,2) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3)$
3. $(1,1) \rightarrow (2,1) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3)$
4. $(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2)$

Ohodnocení necílových políček se provede řešením Bellmanovy rovnice, která představuje soustavu lineárních algebraických rovnic:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U^{\pi}(s')$$

V rovnici značí:

$U^{\pi}(s)$	ohodnocení (*utility*) stavu *s* při použití strategie pohybu π.
R(s)	odměnu (*reward*) za dosažení stavu *s*.
$T(s,\pi(s),s')$	ohodnocení přechodu (*transition*) ze stavu *s* do stavu *s'* při použití strategie π. Stav s‘ je bezprostředním následníkem stavu *s*!
γ	(*discount*) faktor – určuje vliv ohodnocení následujících stavů *s'* na ohodnocení stavu *s*.

K ohodnocení přechodu $T(s,\pi(s),s')$ se může použít odhad maximální pravděpodobnosti ML (*maximum likelihood*) tohoto přechodu, který se určí jednoduše jako podíl počtu přechodů vykonaných během náhodných procházek ze stavu s do stavu s' z celkového počtu přechodů ze stavu s vykonaných během těchto procházek. Například ze stavu *s* = (1,2) byl během výše uvedených čtyř procházek vykonán přechod do stavu *s'* = (1,3) třikrát, do stavu *s'* = (2,2) jednou a do stavu *s'* = (1,1) ani jednou. Tak $T((1,2),\pi(1,2),(1,3)) = 0.75$, $T((1,2),\pi(1,2),(2,2)) = 0.25$ a $T((1,2),\pi(1,2),(1,1)) = 0$.

Pro jednoduchost dále předpokládejme, že odměna za dosažení libovolného nekoncového stavu je nulová (obvykle se však dosažení necílových políček ohodnocuje malou zápornou hodnotou, a tím se systém motivuje k hledání nejkratší cesty.).

Bellmanovy rovnice jsou nyní následující (pro jednoduchost vynecháváme u symbolů *U* symbol pro strategii π):

$$\begin{aligned}
U(1,1) &= 0 + \gamma(0.60 \cdot U(1,2) + 0.40 \cdot U(2,1)) \\
U(1,2) &= 0 + \gamma(0.75 \cdot U(1,3) + 0.25 \cdot U(2,2)) \\
U(1,3) &= 0 + \gamma(1.00 \cdot U(2,3)) \\
U(2,1) &= 0 + \gamma(0.50 \cdot U(1,1) + 0.50 \cdot U(3,1)) \\
U(2,2) &= 0 + \gamma(1.00 \cdot U(1,2)) \\
U(2,3) &= 0 + \gamma(1.00 \cdot U(3,3)) \\
U(3,1) &= 0 + \gamma(1.00 \cdot U(3,2)) \\
U(3,2) &= -1 + \gamma(0) \\
U(3,3) &= +1 + \gamma(0)
\end{aligned}$$

Řešením tohoto systému algebraických rovnic, např. pro faktor $\gamma = 0.9$, je ohodnocení stavů ukázané na následujícím obrázku:

	1	2	3
1	0.27	0.69	0.81
2	–0.28	0.62	0.9
3	–0.9	–1	+1

Obr. 8.7. Ohodnocení stavů desky metodou ADP learning

Pozn..: Samostaným problémem zůstává výběr optimální strategie π. Popis možných řešení tohoto problému přesahuje rámec předmětu.

8.4.2 Metoda TD learning

Metoda TD learning

Metoda TD learning je podobná metodě ADP learning, ohodnocení stavů však počítá, resp. přepočítává po každé náhodné procházce. K tomu používá vztah

$$U^{\pi}(s) = U^{\pi}(s) + \alpha\left(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)\right)$$

Koeficient α představuje koeficient učení a jeho hodnota může klesat s počtem průchodů stavem *s*.

Pro výše uvedený příklad bude přepočítávání ohodnocení metodou TD learning následující (předpokládejme, že $\gamma = 0.9$, $\alpha = 1/n(s)$, kde n(s) je počet průchodů stavem *s*):

- Nejprve se všechna ohodnocení U(i,j) vynulují.
- Po první náhodné procházce se změní pouze ohodnocení stavu (3,3):
  U(3,3) = 0 + 1(1 + 0.9 (0) – 0) = 1
- Po druhé náhodné procházce se změní ohodnocení stavu (2,3):
  U(2,3) = 0 + 1/2(0 + 0.9(1) – 0) = 0.45
- Po třetí náhodné procházce se změní ohodnocení stavů (1,3) a (2,3):
  U(1,3) = 0 + 1/3(0 + 0.9(0.45) – 0) = 0.135
  U(2,3) = 0.45 + 1/3(0 + 0.9(1) – 0.45) = 0.6
- Po čtvrté náhodné procházce se změní pouze ohodnocení stavu (3,2):
  U(3,2) = 0 + 1(–1 + 0.9 (0) – 0) = –1

	1	2	3
1	0	0	0.14
2	0	0	0.6
3	0	–1	+1

Obr. 8.8. Ohodnocení stavů desky metodou TD learning

Pozn.: Zdůrazněme, že ohodnocení každého stavu se přepočítává vždy v okamžiku, kdy systém při náhodné procházce tohoto stavu dosáhne.

8.4.3 Metoda Q learning

Metoda Q learning

Metoda Q learning je podobná metodě TD learning, místo hodnocení stavů však hodnotí akce v těchto stavech. K jejich hodnocení používá vztah:

$$Q(s,a) = Q(s,a) + \alpha\left(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right),$$

kde značí Q(s,a) označuje ohodnocení akce *a* provedené ve stavu *s*. Význam ostatních symbolů je stejný jako výše.

Pro výše uvedený příklad budou k označení akcí a (pro pohyby nahoru, doprava, dolů a doleva) použity symboly U,R,D,L (Up, Right, Down, Left).

Přepočítávání ohodnocení metodou Q learning je velmi podobné metodě TD learning. Pro možnost srovnání předpokládejme, že systém si vybral stejné procházky a stejné koeficienty α a γ:

- Nejprve se všechna ohodnocení Q(s,a) vynulují.
- Po první náhodné procházce se změní ohodnocení všech akcí stavu (3,3):
  Q((3,3),U) = Q((3,3),R) = Q((3,3),D) = Q((3,3),L) =
  = 0 + 1(1 + 0.9 (0) – 0) = 1
- Po druhé náhodné procházce se změní ohodnocení akce Q((2,3),D):
  Q((2,3),D) = 0 + 1/2(0 + 0.9(1) – 0) = 0.45
- Po třetí náhodné procházce se změní ohodnocení akcí Q((1,3),D) a Q((2,3),D):
  Q((1,3),D) = 0 + 1/3(0 + 0.9(0.45) – 0) = 0.135
  Q((2,3),D) = 0.45 + 1/3(0 + 0.9(1) – 0.45) = 0.6
- Po čtvrté náhodné procházce se změní ohodnocení všech akcí stavu (3,2):
  Q((3,2),U) = Q((3,2),R) = Q((3,2),D) = Q((3,2),L) =
  = 0 + 1(–1 + 0.9 (0) – 0) = –1

Přes výraznou podobnost se metody TD learning a Q learning významně liší. Pasivní metody posilovaného učení a tedy i metoda TD learning předpokládají předem danou strategii pohybu π - obvykle náhodnou, ale s jiným, než výše uvažovaným rovnoměrným rozložením pravděpodobnosti (některé směry pohybu bývají preferované). Aktivní přístupy k učení a tedy i metoda Q learning naopak předpokládají, že umělý systém sám v každém stavu rozhodne, kterou následnou akci (směr pohybu) zvolí.

8.5 Shrnutí

Shrnutí

V kapitole byly vysvětleny základní principy strojového učení - postupně zde byly popsány a vysvětleny základní principy vybraných metod učení s učitelem (rozhodovací stromy a prohledávání prostoru verzí), učení bez učitele (algoritmus K-means clustering) a posilovaného učení (metody ADP, TD a Q learning).

7.7 Kontrolní otázky

Kontrolní otázky

5. Jaké znáte skupiny metod strojového učení?
6. Jaký je princip metod učení s učitelem?
7. Jaký je princip metod učení bez učitele?
8. Jaký je princip metod posilovaného učení?

9 ROZPOZNÁVÁNÍ

10 hod

Cílem této kapitoly je seznámit studenty se základními principy rozpoznávání obrazů. Jsou zde popsány a vysvětleny principy statistických metod (pro oddělitelné i neoddělitelné shluky obrazů reprezentovaných vektory číselných příznaků) a metod strukturálního rozpoznávání (obrazů reprezentovaných řetězy nebo grafy kvalitativních příznaků).

9.1 Úvodní informace

Úvodní informace

Rozpoznávání (*pattern recognition*) je relativně samostatná část strojového učení, jejímž cílem je naučit se správně klasifikovat objekty nebo jevy/signály na základě obrazů/vzorů (*patterns*) jejich příznaků. Tyto příznaky mohou být buď kvantitativní/číselné nebo kvalitativní/strukturální. V prvním případě se využívá statistických informací o příznacích obrazů obsažených v množině trénovacích dat a jde o tzv. statistické příznakové rozpoznávání, ve druhém případě se využívá informací o vztazích mezi příznaky obrazů rozpoznávaných struktur a jde o tzv. syntaktické nebo strukturální rozpoznávání.

Pozn.: Zásadním předpokladem úspěšného rozpoznávání je výběr relevantních příznaků. Tento výběr je přirozeně problémově závislý a záleží výhradně na zkušenosti uživatele, který příznaky popisující objekty nebo jevy vybírá.

9.2 Statistické rozpoznávání

Statistické rozpoznávání

Předpokládejme, že rozpoznávané objekty nebo jevy jsou popsané svými obrazy, kterými jsou n-rozměrné číselné vektory (symbol $^{T}\vec{x}$ označuje transponovaný vektor (pro shodu s výpočty naznačenými dále))

$$\vec{x}_i = {}^{T}[x_{i1}, x_{i2}, \ldots, x_{in}]$$

Dále předpokládejme, že máme k dispozici trénovací množinu dvojic

$$T = \{(\vec{x}_1, c_i), (\vec{x}_2, c_j), \ldots, (\vec{x}_P, c_k)\}$$

kde první prvek každé dvojice představuje obraz (vektor příznaků) a druhý prvek označuje jeho třídu (číslo – skalár) z předem dané množiny R tříd:

$$C = \{c_1, c_2, \ldots, c_R\}$$

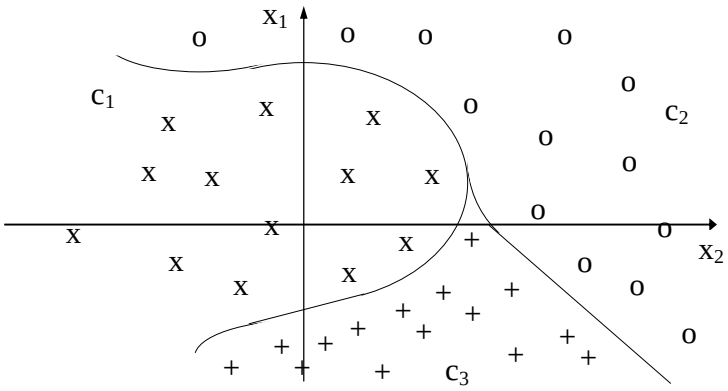
Metody rozpoznávání vycházejí z předpokladu, že obrazy objektů nebo jevů stejných tříd tvoří v n-rozměrném obrazovém prostoru shluky. Tyto shluky mohou být od sebe zřetelně oddělitelné (separovatelné), nebo se mohou prolínat; pak jsou neoddělitelné (neseparovatelné).

Umělé systémy, které se na trénovací množině naučí obrazy rozpoznávat a poté klasifikují nové obrazy, se nazývají klasifikátory.

Rozpoznávání obrazů reprezentovaných oddělitelnými shluky

Příklad tří oddělitelných shluků ve dvourozměrném obrazovém prostoru je uveden na následujícím obrázku (Obr. 9.1).

Učení klasifikátoru pak spočívá v nalezení křivek (obecně *n*-rozměrných ploch), které obrazy jednotlivých tříd od sebe oddělují. Vychází se přitom z tzv. diskriminačních funkcí, které jsou definovány takto:



Obr. 9.1 Oddělitelné shluky ve dvourozměrném obrazovém prostoru

$$g_r(\vec{x}) > g_s(\vec{x}) \quad r \neq s, \quad c_{\vec{x}} = r$$

Výše uvedený vztah říká, že pro daný vektor $\vec{x}$ musí vracet největší hodnotu ta diskriminační funkce, která reprezentuje třídu do níž vektor $\vec{x}$ patří. Je zřejmé, že rovnice rozdělujících křivek (*n*-rozměrných nadploch) jsou pak dány vztahy:

$$g_r(\vec{x}) = g_s(\vec{x}) \quad r \neq s$$

Činnost klasifikátoru při vlastní klasifikaci je tedy zcela jednoduchá: Pro daný vstupní obraz/vektor vypočítá hodnoty všech *R* diskriminačních funkcí a klasifikuje příslušný obraz/vektor do třídy, která přísluší funkci s největší výstupní hodnotou.

Často se používá klasifikace do dvou tříd (dichotonomie). Pak je možné použít jedinou diskriminační funkci, která představuje rozdíl výše definovaných dvou diskriminačních funkcí a klasifikovat pouze podle znaménka hodnoty této funkce:

$$g(\vec{x}) = g_1(\vec{x}) - g_2(\vec{x})$$
$$g(\vec{x}) > 0 \qquad pro\ c_{\vec{x}} = 1$$
$$g(\vec{x}) < 0 \qquad pro\ c_{\vec{x}} = 2$$

Problémem zůstává učení klasifikátoru, které spočívá v nalezení diskriminačních funkcí.

Existuje rozsáhlá teorie, která se zabývá problematikou hledání diskriminačních funkcí. V této opoře si ukážeme pouze algoritmus učení lineárního klasifikátoru pro dichotomii, který hledá koeficienty lineární diskriminační funkce

$$g(\vec{x}) = q_0 + q_1 x_1 + q_2 x_2 + \ldots + q_n x_n = q_0 + \sum_{i=1}^{n} q_i x_i$$

Učení lineárního klasifikátoru pro dichotomii

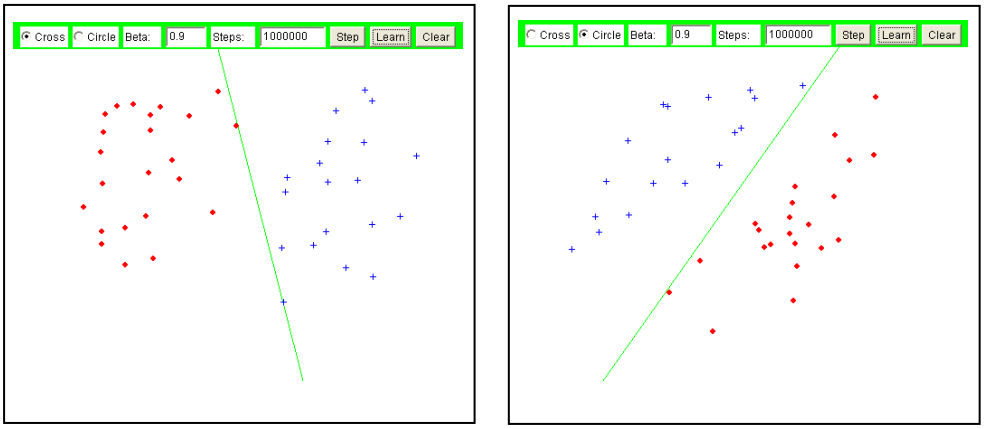
Nechť $T = \{(\vec{x}_1, c_1), (\vec{x}_2, c_2), ..., (\vec{x}_P, c_P)\}$ je trénovací množina, $\vec{x}_i = {}^T[x_{i0}, x_{i1}, \ldots, x_{in}]$, $x_{i0} = 1$, $c_i \in$ <-1,1> , $\vec{q}$ je vektor koeficientů diskriminační funkce (vektor vah) $\vec{q} = [q_0, q_1, \ldots, q_n]$ a β je konstanta:

1. Vynuluj vektor vah $\vec{q} = \vec{0}$
2. Nastav indikátor změny *modif* = **false**
3. Pro každý vektor $\vec{x}_i$ z trénovací množiny (i = 1...P), který není správně klasifikován ($sign(g(\vec{x}_i)) \neq c_i$), uprav vektor vah podle vztahu

$$\vec{q} = \vec{q} + \beta \frac{c_i{}^T \vec{x}_i}{\|\vec{x}_i\|}, \qquad \|\vec{x}_i\| = \sqrt{x_0^2 + x_1^2 + \ldots + x_n^2}$$

   a nastav indikátor změny *modif* = **true**;
4. Pokud došlo k úpravě vektoru vah (*modif* = = **true**) vrať se na bod 2, jinak je učení klasifikátoru ukončeno.

Dva příklady rozdělení dvourozměrného obrazového prostoru klasifikátorem pro dichotonomii jsou ukázány na Obr. 9.2.



Obr. 9.2 Příklady rozdělení dvourozměrného obrazového prostoru klasifikátorem pro dichotonomii

Pozn.: R klasifikátorů pro dichotonomii naučených na klasifikaci „patří/nepatří´do třídy r, r ∈ <1,R>, nahradí jeden lineární klasifikátor klasifikující do R tříd.

Nelze-li obrazy různých tříd oddělit od sebe lineárními nadplochami, lze tento problém řešit některým z následujících postupů:

- Použitím několika lineárních diskriminačních funkcí pro každou třídu (vede k obtížnému učení):

$$g_r(\vec{x}) = \max_{h=1\ldots H}\left(g_r^h(\vec{x})\right)$$

- Použitím obecných diskriminačních funkcí (v praxi obtížně proveditelné):

$$g_r(\vec{x}) = a_r + b_r \ln(x_1) + c_r tg(d_r x_2) + \ldots$$

- Použitím nelineární transformace, která převede původní *n*-rozměrný obrazový prostor na *m*-rozměrný obrazový prostor, ve kterém již lze obrazy separovat lineárními nadrovinami – pak lze použít klasifikátor s lineárními diskriminačními funkcemi (podobně pracují neuronové sítě).
- Použitím dále popsaného klasifikátoru pro neoddělitelné třídy obrazů

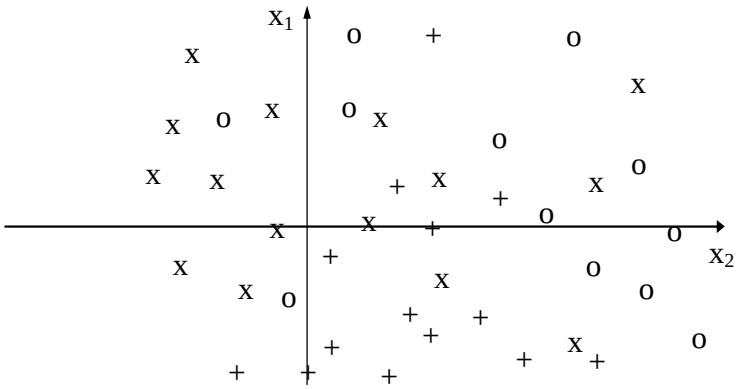
Rozpoznávání obrazů reprezentovaných neoddělitelnými shluky

Příklad tří neoddělitelných shluků ve dvourozměrném obrazovém prostoru je uveden na Obr. 9.3.

V případě neoddělitelných shluků již nelze tvrdit, že obraz $\vec{x}$ patří do třídy *r*, ale lze konstatovat, že obraz $\vec{x}$ patří do třídy *r* s pravděpodobností $p(r/\vec{x})$.

Při odvozování učícího algoritmu vycházejme z definice ztrátové matice, jejímiž prvky jsou ztráty uživatele, když klasifikuje obraz třídy *s* do třídy *r*:

$$\lambda = \begin{bmatrix} \lambda(1,1) & \lambda(1,2) & \cdots & \lambda(1,R) \\ \lambda(2,1) & \lambda(2,2) & \cdots & \lambda(2,R) \\ \vdots & \vdots & \vdots & \vdots \\ \lambda(R,1) & \lambda(R,2) & \cdots & \lambda(R,R) \end{bmatrix}$$



Obr. 9.3 Neoddělitelné shluky ve dvourozměrném obrazovém prostoru

Ztráta při správné klasifikaci je zřejmě nulová ($\lambda(i,i) = 0$), pro jednoduchost dále předpokládejme, že ztráta při jakékoliv chybné klasifikaci je $\lambda(i,j) = 1$ *i # j*. Pak ztrátová matice má následující tvar:

$$\lambda = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & \cdots & 0 \end{bmatrix}$$

Dále nechť $p(\vec{x})$ je pravděpodobnost výskytu obrazu $\vec{x}$ a nechť $p(i/\vec{x})$ je podmíněná pravděpodobnost skutečnosti, že obraz $\vec{x}$ patří do třídy *i*.

Příspěvek ztráty $\lambda(r,s)$ k celkové střední ztrátě je dán součinem této ztráty s apriorní pravděpodobností výskytu obrazu $\vec{x}$ a podmíněné pravděpodobnosti, že tento obraz je obrazem objektu/jevu třídy *s*; na součin obou pravděpodobností se pak aplikuje Bayesovo pravidlo :

$$\lambda(r,s)p(s/\vec{x})p(\vec{x}) \equiv \lambda(r,s)p(\vec{x}/s)p(s)$$

Celková střední ztráta chybné klasifikace do třídy *r* je pak dána součtem:

$$L_{\vec{x}}(r) = \sum_{s=1}^{R} \lambda(r,s) p(\vec{x}/s) p(s),$$

který je možné s respektováním ztrátové matice upravit na tvar

$$\begin{aligned} L_{\vec{x}}(r) &= \sum_{s=1}^{R} \lambda(r,s) p(\vec{x}/s) p(s) = \\ &\sum_{s=1}^{R} p(\vec{x}/s) p(s) - p(\vec{x}/r) p(r) = \\ &= p(\vec{x}) - p(\vec{x}/r) p(r) \end{aligned}$$

Cílem klasifikátoru je přirozeně tuto ztrátu minimalizovat (pravděpodobnost $p(\vec{x})$ je dána, minimalizovat ji nelze, a proto ji lze vypustit):

$$\min(L_{\vec{x}}(r)) = \max(p(\vec{x}/r) p(r)) = \max(p(r/\vec{x}) p(\vec{x}))$$

Předchozími úvahami jsme dospěli k funkci, jejíž hodnota má být při klasifikaci libovolného obrazu $\vec{x}$ maximalizována - nejmenší ztrátu proto utrpí uživatel, klasifikuje-li obraz do třídy *r*, pro níž je hodnota součinu $p(\vec{x}/r) p(r)$ maximální, a proto funkce daná tímto součinem je hledanou diskriminační funkcí

$$g_r(\vec{x}) = p(\vec{x}/r) p(r)$$

Z praktických důvodů, které budou zřejmé níže, je vhodné za diskriminační funkci použít přirozený logaritmus výše uvedeného součinu pravděpodobností (pro a > b platí, že ln(a) > ln(b))

$$g_r(\vec{x}) = \ln(p(\vec{x}/r)) + \ln(p(r))$$

Předpokládejme dále, že rozložení obrazů jednotlivých tříd v obrazovém prostoru se řídí n-rozměrným normálním rozložením hustoty pravděpodobnosti

$$p(\vec{x}/r) = \frac{1}{(2\pi)^{\frac{n}{2}} \sqrt{|\overline{\sigma}_r|}} e^{-\frac{1}{2}\,{}^{T}(\vec{x}-\vec{\mu}_r)\overline{\sigma}_r^{-1}(\vec{x}-\vec{\mu}_r)}$$

Pak diskriminační funkce jsou dány vztahy:

$$\begin{aligned} g_r(\vec{x}) &= \ln(p(\vec{x}/r)) + \ln(p(r)) = \\ &= -\frac{1}{2}\,{}^{T}(\vec{x}-\vec{\mu}_r)\,\overline{\sigma}_r^{-1}(\vec{x}-\vec{\mu}_r) - \frac{1}{2}\ln(|\overline{\sigma}_r|) + \ln(p(r)) \end{aligned}$$

Pozn.: Logaritmus konstanty $1/(2\pi)^{\frac{n}{2}}$ je možné vypustit, protože je stejný pro všechny obrazy všech tříd a na porovnání proto nemá vliv.

Učení klasifikátoru metodou odhadu (pro n-rozměrné normální rozložení):

Nechť $T = \{(\vec{x}_1, c_1), (\vec{x}_2, c_2), ..., (\vec{x}_P, c_P)\}$ je trénovací množina, $\vec{x}_i = {}^{T}[x_{i1}, x_{i2}, \ldots, x_{in}]$ $c_i \in$ <1,R>, $\vec{\mu}_r = {}^{T}[u_1, u_2 \ldots, u_n]$ jsou vektory středních hodnot, $\overline{\sigma}_r$ jsou disperzní matice, $\vec{\mu} = {}^{T}[u_1, u_2 \ldots, u_n]$, $\vec{v} = {}^{T}[v_1, v_2 \ldots, v_n]$, $\vec{w} = {}^{T}[w_1, w_2 \ldots, w_n]$ jsou pomocné vektory, $p(r)$ je pravděpodobnost výskytu obrazů třídy *r*. Ostatní symboly jsou jednorozměrné pomocné proměnné.

Pro všechny třídy *r* nastav: $\vec{\mu}_r = \vec{0}; \quad \overline{\sigma}_r = \overline{0}; \quad P_r = 0;$

1. Pro všechny obrazy $\vec{x}_i$ trénovací množiny *T* počítej:
   - $\vec{\mu} = \vec{\mu}_{c_i}; \quad \vec{\mu}_{c_i} = \left(P_{c_i}\vec{\mu}_{c_i} + \vec{x}_i\right) / \left(P_{c_i} + 1\right)$
   - Jestliže $P_{c_i} > 0$ počítej:

     $\vec{v} = \left(\vec{x}_i - \vec{\mu}_{c_i}\right)$

     $\vec{w} = \left(\vec{\mu} - \vec{\mu}_{c_i}\right)$

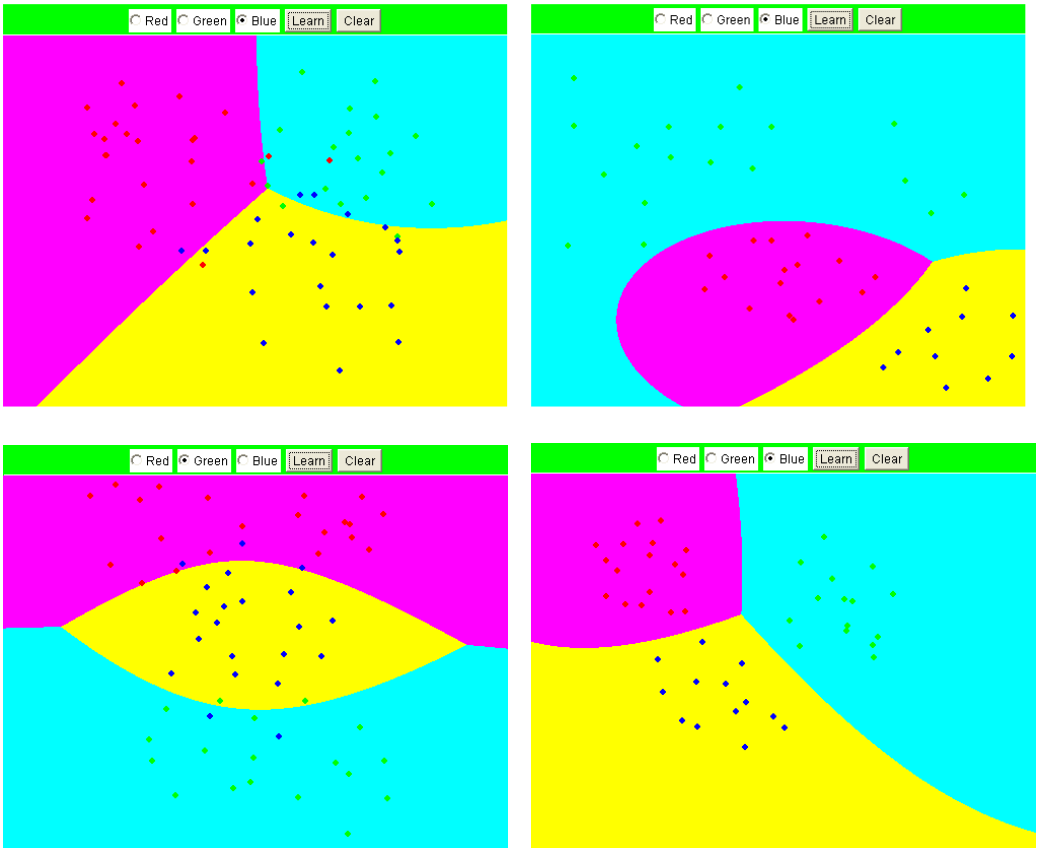
     $\overline{\sigma}_{c_i} = \left(\left(P_{c_i} - 1\right)\overline{\sigma}_{c_i} + \vec{v}^{\,T}\vec{v} + P_{c_i}\vec{w}^{\,T}\vec{w}\right) / P_{c_i}$

     $P_{c_i} = P_{c_i} + 1$
2. Pro všechny třídy *r* počítej:

   $p(r) = P_r / P$

   $g_r(\vec{x}) = -\frac{1}{2}{}^{T}(\vec{x} - \vec{\mu}_r)\,\overline{\sigma}_r^{-1}(\vec{x} - \vec{\mu}_r) - \frac{1}{2}\ln\left(\left|\overline{\sigma}_r\right|\right) + \ln(p(r))$

Na Obr. 9.4 je ukázáno rozdělení dvourozměrného obrazového prostoru do třech podprostorů klasifikátorem nastaveným výše uvedenou metodou odhadu. Povšimněte si, že klasifikátor pracuje uspokojivě i v případech, kdy normální rozložení obrazů není dodrženo.



Obr. 9.4 Příklady rozdělení dvourozměrného obrazového prostoru klasifikátorem učeným metodou odhadu

9.2 Syntaktické a strukturální rozpoznávání

Syntaktické a strukturální rozpoznávání

Syntaktické a strukturální rozpoznávání je založeno na předpokladu, že každý rozpoznávaný objekt nebo jev lze rozložit na jednoduché části, která se nazývají primitiva. Objekt nebo jev je pak reprezentován obrazem, který vyjadřuje vztahy mezi primitivy tohoto objektu nebo jevu.

První problémem je výběr optimální množiny primitiv. Malá množina má malou vyjadřovací sílu, velká množina může být nezvládnutelná při učení klasifikátoru.

Druhým problémem je výběr úrovně/komplexnosti popisu (řetěz, strom, graf). Řetězový popis je nejjednodušší a učení klasifikátoru je relativně snadné. Grafový popis je univerzální, učení klasifikátorů je však výrazně složitější.

Strukturální rozpoznávání je v literatuře chápáno dvěma různými způsoby:

- Jako vytvoření obrazu, tj. vytvoření popisu struktury objektu nebo jevu pomocí primitiv.
- Jako rozpoznávání porovnáváním obrazu se vzory/prototypy uloženými v databázi (*template matching*).

Dále se omezíme na popis syntaktického rozpoznávání obrazů představovaných řetězy. Popis ostatních problémů (syntaktické rozpoznávání stromových a grafových obrazů a *template matching*) přesahuje rámec předmětu.

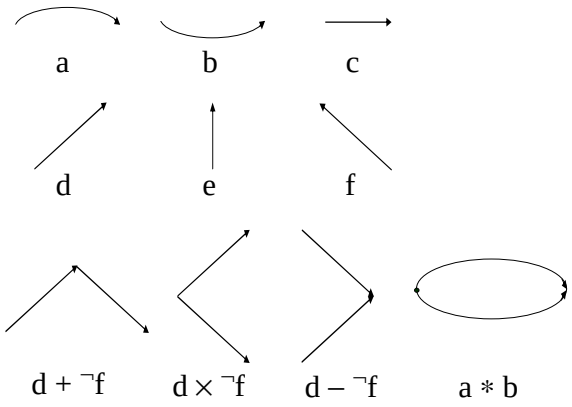
Syntaktické rozpoznávání klasifikuje obrazy (obecně do R tříd) pomocí syntaktické analýzy. Obrazy, tj. řetězové popisy objektů nebo jevů, jsou přitom chápány jako věty, primitiva jako množina terminálních symbolů. Pokud pak gramatika jazyka pro obrazy třídy r přijme všechny věty, které reprezentují obrazy třídy r, a odmítne všechny věty, které reprezentují obrazy jiných tříd, lze klasifikaci převést na problém určení gramatiky, která jako jediná ze všech R gramatik přijímá rozpoznávaný obraz/větu.

Se syntaktickým rozpoznáváním je spojen problém výběru typu gramatiky. Dále se pro jednoduchost omezíme pouze na deterministickou regulární gramatiku.

9.3.1 PDL

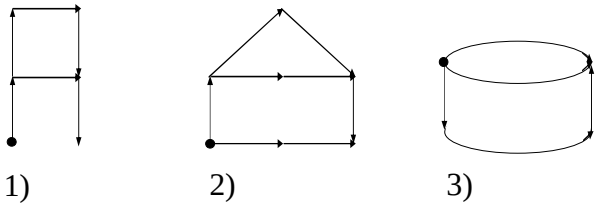
PDL

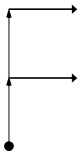
Na výběr primitiv a operátorů vyjadřující jejich vzájemné vztahy není kladeno žádné omezení. Příkladem možného přístupu k vytváření řetězového popisu je jazyk pro popis obrázků PDL (*picture description language*).



Obr. 9.5 Příklady primitiv a jejich vzájemných vztahů v PDL

PDL předpokládá pět operátorů pro označení vzájemných vztahů (+, ¬, ×, –, ∗), a to nejen vztahů mezi primitivy, ale i vztahů mezi složitějšími strukturami. Význam těchto operátorů je zřejmý z Obr. 9.5.





1) e + ((e + c + ¬e) ∗ c) + ¬e
2) (e + ((d + ¬f) ∗ (c+c)) + ¬e) ∗ ∗ (c +c)
3) a ∗ b ∗ (¬e + b + e)
4) e + (c × (e + c))

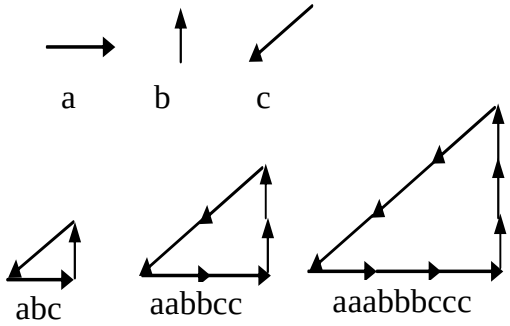
Obr. 9.6 Příklady popisů jednoduchých struktur prostředky PDL

Na Obr. 9.6 jsou ukázány čtyři příklady popisů jednoduchých struktur prostředky PDL. I tyto jinak velmi jednoduché popisy jsou však složité pro praktickou ukázku syntaktického rozpoznání. Uvažujme proto dva jiné velmi jednoduché příklady:

x+y

Příklad 1. Syntaktické rozpoznání trojúhelníků

Předpokládejme tři primitiva popsaná symboly a, b, c, z nichž jsou složeny tři různě velké trojúhelníky (Obr. 9.7).



Obr. 9.7 Příklady tří podobných trojúhelníků a jejich obrazů

Uvažujme regulární gramatiku $G(V_N,V_T,P,S)$ kde symbol V_N značí množinu neterminálních symbolů $\{Y_1,Y_2,Y_{21},Y_3,Y_{32},Y_{31},Z_1,Z_2,Z_3\}$, symbol V_T množinu terminálních symbolů {a,b,c}, symbol P množinu přepisovacích pravidel a symbol S startovací symbol.

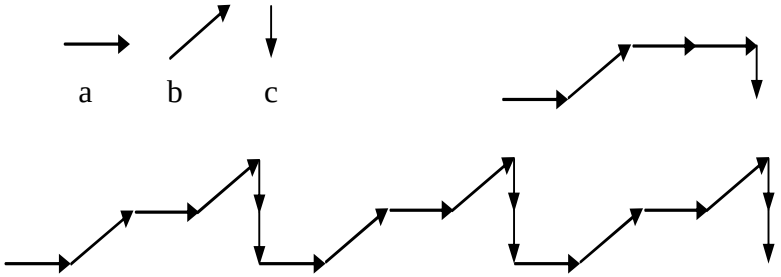
Věty/obrazy všech tří trojúhelníků přijímá a současně obrazy veškerých jiných objektů odmítá gramatika regulárního jazyka $L = \{a^n b^n c^n \mid n = 1, 2, 3\}$, která má následující přepisovací pravidla:

$S \rightarrow aY_1$	$Y_1 \rightarrow bZ_1$	$Z_1 \rightarrow c$
$Y_1 \rightarrow aY_2$	$Y_2 \rightarrow bY_{21}$	
	$Y_{21} \rightarrow bZ_2$	$Z_2 \rightarrow cZ_1$
$Y_2 \rightarrow aY_3$	$Y_3 \rightarrow bY_{32}$	
	$Y_{32} \rightarrow bY_{31}$	
	$Y_{31} \rightarrow bZ_3$	$Z_3 \rightarrow cZ_2$

x+y

Příklad 2. Syntaktické rozpoznání/kontrola správného signálu

Předpokládejme tři primitiva popsaná symboly a, b, c, která popisují signál ukázaný na Obr. 9.8.



Obr. 9.7 Příklad signálu

Předpokládejme, že správný signál (na obrázku dole) se vyznačuje opakující se posloupností $(ababcc)^n$, a že ve sledovaném časovém úseku je přípustná jediná odchylka, ukázaná na obrázku vpravo nahoře, která je popsána obrazem *abaac*.

Uvažujme opět regulární gramatiku $G(V_N,V_T,P,S)$. Symbol V_N nyní značí množinu neterminálních symbolů $\{S_1,P,P_1,Q,Q_1,R,R_1,U,U_1,V,V_1,X,Y\}$, symbol V_T množinu terminálních symbolů {a,b,c}, symbol P množinu přepisovacích pravidel a symbol S startovací symbol.

Přepisovací pravidla:

!

$S \rightarrow aP$		$S_1 \rightarrow aP_1$
$P \rightarrow bQ$		$P_1 \rightarrow bQ_1$
$Q \rightarrow aR$	$Q \rightarrow aX$	$Q_1 \rightarrow aR_1$
$R \rightarrow bU$	$X \rightarrow aY$	$R_1 \rightarrow bU_1$
$U \rightarrow cV$	$Y \rightarrow cS_1$	$U_1 \rightarrow cV_1$
$V \rightarrow cS$		$V_1 \rightarrow cS_1$

9.3.2 Inference gramatik

Inference gramatik

Učení syntaktického klasifikátoru spočívá v nalezení přepisovacích pravidel (tzv. inference gramatiky).

Uvažujme gramatiku $G(V_N,V_T,P,S)$, kde symbol V_N značí množinu neterminálních symbolů, symbol V_T množinu terminálních symbolů, symbol P množinu přepisovacích pravidel a symbol S ($S \in V_N$) startovací symbol. Abeceda V je dána sjednocením neterminálních a neterminálních symbolů ($V = V_T \cup V_N$). Symbol V^* označuje množinu všech slov nad abecedou V^*. Jazyk L je podmnožinou této množiny ($L \subseteq V^*$), pro jazyk L(G) generovaný

gramatikou G musí platit $L(G) = \{x \mid x \in V^*, S \rightarrow x\}$. Trénovací množina $T = \{S^+, S^-\}$ je dána množinou pozitivních příkladů $S^+ = \{x_i \mid x_i \in L(G)\}$ a množinou negativních příkladů $S^- = \{x_i \mid x_i \notin L(G)\}$. Inferenční procedura pak může být popsána takto:

0. Nechť G(0) je počáteční odhad gramatiky, k = 0.
1. i = 1, modif = false,
2. Vyber z trénovací množiny slovo x_i.
3. Jestliže $x_i \in S^+$ a G(k) negeneruje x_i tak:
   modifikuj $G(k) \rightarrow G(k+1)$,
   modif = true,
   k = k + 1.
4. Jestliže $x_i \in S^-$ a G(k) generuje x_i tak:
   modifikuj $G(k) \rightarrow G(k+1)$,
   modif = true,
   k = k + 1.
5. i = i + 1.
6. Není-li trénovací množina T prázdná vrať se na bod 2.
7. Byla-li gramatika modifikována (modif = = true) vrať se na bod 1.

Problémem zůstává modifikace gramatiky, podrobnější popis řešení tohoto problému však přesahuje rámec předmětu.

9.4 Shrnutí

Shrnutí

V závěrečné kapitole byly vysvětleny základní principy rozpoznávání obrazů. Byly zde popsány přístupy ke statistickému rozpoznávání obrazů (klasifikátor pro dichotonomii a klasifikátor nastavovaný metodou odhadu) i přístupy k syntaktickému a strukturálnímu rozpoznávání, včetně stručného popisu PDL.

9.5 Kontrolní otázky

Kontrolní otázky

1. Do jakých skupin se dělí metody rozpoznávání obrazů?
2. Co jsou diskriminační funkce?
3. Jak se nastavuje klasifikátor pro dichotonomii?
4. Jaký je princip klasifikátory nastavovaného metodou odhadu.
5. Jaký je princip syntaktického rozpoznávání?
6. Jaký je princip strukturálního rozpoznávání?