VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ALGORITMY PRO UMĚLOU INTELIGENCI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE JAN PETRŽELKA
AUTHOR

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ALGORITMY PRO UMĚLOU INTELIGENCI

AI ALGORITHMS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE JAN PETRŽELKA

AUTHOR

VEDOUCÍ PRÁCE Ing. VLADIMÍR JANOUŠEK, Ph.D.

SUPERVISOR

BRNO 2007

Abstrakt

Tato diplomová práce se zabývá algoritmy používanými v oblasti umělé inteligence, konkrétně se jedná o algoritmy popsané v knize Artificial Inteligence: A Modern Approach autorů Russela a Norviga a jejich implementaci v jazyce Squeak Smalltalk. Je kladen důraz na objektově orientovaný přístup, který vyplývá z podstaty jazyka Smalltalk. Zdrojem jsou kromě popisů algoritmů v pseudokódu přímo v knize také existující implementace v jazycích Lisp, Python a Java. Tato práce se věnuje algoritmům pro práci s inteligentními agenty a prostředími pro simulaci těchto agentů, prohledávání stavového prostoru, hraní her, plánování, logice, pravděpodobnosti a učení.

Klíčová slova

umělá inteligence, algoritmy, smalltalk, squeak, AIMA, prohledávání, hledání, plánování, hraní her, logika, učení, pravděpodobnost, neurčitost, agent, prostředí, OOP

Abstract

This master's thesis describes artificial intelligence algorithms based on the book Artificial Inteligence: A Modern Approach by S. Russel and P. Norvig and implementation of the algorithms in the Squeak Smalltalk programming language with object oriented approach. Algorithms are based on pseudocode in the book and existing implementations in Lisp, Python and Java language. Main concepts are intelligent agents, agent simulation environments, state space search, game playing, planning, uncertainty and learning.

Keywords

artificial intelligence, algorithms, smalltalk, squeak, AIMA, search, planning, game playing, logic, learning, probability, uncertainty, agent, environment, OOP

Citace

Jan Petrželka: Algoritmy pro umělou inteligenci, diplomová práce, Brno, FIT VUT v Brně, 2007

Algoritmy pro umělou inteligenci

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Vladimíra Janouška. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Jan Petrželka
21. května 2007

Poděkování

Rád bych na tomto místě poděkoval panu Ing. Vladimíru Janouškovi za odbornou pomoc poskytnutou při řešení problémů spojených s touto prací a především za seznámení s tak výjmečným programovacím jazykem, jakým bezesporu Squeak Smalltalk je.

© Jan Petrželka, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**

2 Agenti a prostředí **6**
2.1 Co je to umělá inteligence? . . . 6
2.2 Inteligentní agent . . . 6
2.3 Prostředí . . . 6
2.3.1 Prostředí plně nebo částečně pozorovatelné . . . 7
2.3.2 Prostředí deterministické nebo stochastické . . . 7
2.3.3 Prostředí epizodické nebo sekvenční . . . 7
2.3.4 Prostředí statické nebo dynamické . . . 7
2.3.5 Prostředí diskrétní nebo spojité . . . 7
2.3.6 Prostředí s jedním nebo více agenty . . . 8
2.4 Implementace agentů a jejich prostředí . . . 8
2.4.1 Třída Environment . . . 8
2.4.2 Třída GridEnvironment . . . 9
2.4.3 Třída VacuumWorld . . . 9
2.4.4 Třída WumpusWorld . . . 9
2.4.5 Třída Agent . . . 9
2.4.6 Třída AskUserAgent . . . 10
2.4.7 Třídy RandomVacuumAgent a RandomWumpusAgent . . . 10
2.4.8 Třídy ReactiveVacuumAgent a ReactiveWumpusAgent . . . 10
2.4.9 Třída Object . . . 10
2.4.10 Třída ObstacleObject . . . 10
2.4.11 Třída DeadlyObject . . . 10
2.4.12 Třída AgentBodyObject . . . 10
2.4.13 Třída ObjectSpec . . . 11

3 Řešení problémů **12**
3.1 Řešící agent . . . 12
3.2 Hledání řešení . . . 13
3.3 Neinformované metody hledání . . . 13
3.3.1 Breadth-first search . . . 13
3.3.2 Uniform-cost search . . . 14
3.3.3 Depth-first search . . . 14
3.3.4 Iterative deepening depth-first search . . . 14
3.4 Informované metody hledání . . . 14
3.4.1 Best-first search . . . 15
3.4.2 Greedy best-first search . . . 15

3.4.3 A* search . . . 15
3.5 Hraní her . . . 15
3.5.1 Minimax . . . 16
3.5.2 Alpha-beta pruning . . . 16
3.6 Implementace . . . 16
3.6.1 Třída Problem . . . 16
3.6.2 Třída Node . . . 17
3.6.3 Třída Game . . . 17
3.6.4 Třída GameState . . . 17
3.6.5 Prostředí . . . 17
3.7 Příklady . . . 18
3.7.1 Misionáři a kanibalové . . . 18
3.7.2 Piškvorky . . . 18
3.7.3 Dáma . . . 18

4 Logika **19**
4.1 Úvod do logiky . . . 19
4.2 Výroková logika . . . 20
4.2.1 Jazyk výrokové logiky . . . 20
4.2.2 Odvozování . . . 20
4.3 Predikátová logika . . . 21
4.3.1 Jazyk predikátové logiky . . . 21
4.3.2 Odvozování v predikátové logice . . . 21
4.4 Implementace . . . 22
4.4.1 Třídy a algoritmy výrokové logiky . . . 22
4.4.2 Třídy a algoritmy predikátové logiky . . . 22

5 Plánování **24**
5.1 Jazyk plánovacích problémů . . . 24
5.2 Řešení plánovacích problémů . . . 25
5.3 Implementace . . . 25
5.3.1 Objekty . . . 25
5.3.2 Prostředí . . . 26
5.3.3 Agent . . . 26

6 Neurčitost a pravděpodobnost **27**
6.1 Neurčité znalosti . . . 27
6.2 Bayesovské sítě . . . 28
6.3 Statistické metody . . . 29
6.4 Implementace . . . 29
6.4.1 Třída EnumerateJointAsk . . . 29
6.4.2 Třída BayesNetNode . . . 30
6.4.3 Třída BayesNet . . . 30

7 Učení **32**
7.1 Princip učení . . . 32
7.2 Rozhodovací stromy . . . 33
7.3 Další metody učení . . . 34
7.4 Implementace . . . 35

7.4.1 Třída DataSet 35
7.4.2 Třída DecisionTreeLearner 35
7.4.3 Třída AdaBoostLearner 35
7.4.4 Třída DecisionListLearner 36

8 Závěr **37**

Kapitola 1

Úvod

Objektem zájmu této práce je kniha Artificial Inteligence: A Modern Approach (Second Edition) autorů Stuarta Russela a Petera Norviga [1]. Tato kniha je primárně určena jako učebnice pro studium umělé inteligence.

Jelikož umělá inteligence je velmi široký pojem, popisuje i tato kniha mnoho pohledů na tuto problematiku: logiku, pravděpodobnost, vnímání, logické myšlení, učení a mnoho jiných. Všechny tyto oblasti spojuje jeden pojem – inteligentní agent. Umělá inteligence je zde definována jako studium agentů, kteří přijímají vjemy z prostředí a na jejich základě konají akce. Každý takový agent obsahuje funkci, která přiřazuje ke vjemu posloupnost akcí, přičemž existuje mnoho způsobů jak tuto funkci reprezentovat. Pro lepší názornost obsahuje kniha řadu algoritmů v pseudokódu, které představují řešení vybraných problémů z této oblasti. Aby bylo možné použít tyto algoritmy v praxi, existují v současné době jejich implementace ve třech jazycích, v Lispu, Pythonu a Javě.

Cílem této práce je vytvořit podobnou implementaci v jazyce Squeak Smalltalk a popsat tyto algoritmy jak po teoretické stránce, tak i po stránce implementační. Smalltalk, jakožto čistě objektově orientovaný jazyk, poskytuje, díky zapouzdření, vysokou úroveň abstrakce a to přispívá k přehlednosti výsledného kódu. Ten má být určen především pro výukové účely, kde přehlednost a čistota kódu je jedním ze stěžejních faktorů. Squeak je open source prostředí založené na Smalltalku-80, které se svým grafickým uživatelským rozhraním Morphic je vhodné pro výuku, navíc pro tento jazyk zatím neexistuje žádná významná knihovna algoritmů pro umělou inteligenci.

Pro porozumnění kódu se předpokládají základní znalosti v oblasti informatiky (algoritmy, datové struktury) a jazyka Smalltalk (např. seriál Pavla Křivánka na serveru root.cz [2] je dobrým úvodem do tohoto jazyka).

Tato práce navazuje na můj ročníkový projekt, ve kterém jsem začal s implementací dále popsaných algoritmů, konkrétně z kapitol 2, 3 a 5, a v rámci semestrálního projektu jsem přidal kód ze zbývajících kapitol. Souhrn implementovaného kódu pro Smalltalk a jeho umístění v knize je sepsán v příloze A.

Následující kapitoly obsahují vždy teoretický popis dané oblasti umělé inteligence a principy řešení problémů s nimi spojených, sloužící jako úvod do problematiky a pro lepší porozumění vytvořeného kódu, a dále konkrétní popis implementace vytvořených tříd a metod ve Squeaku.

Kapitola 2 seznamuje se stěžejní částí kódu, inteligentními agenty

Kapitola 3 popisuje řešení problémů a algoritmy pro hledání ve stavovém prostoru

Kapitola 4 představuje základy výrokové a predikátové logiky

Kapitola 5 se zabývá plánováním

Kapitola 6 vysvětluje prinicipy neurčitosti a pravděpodobnosti při řešení problémů

Kapitola 7 je zaměřena na učení

Kapitola 8 shrnuje a vyhodnocuje dosažené výsledky

Kapitola 2

Agenti a prostředí

2.1 Co je to umělá inteligence?

Existuje několik definic toho pojmu, které se odlišují především pohledem na toto téma. První pohled rozlišuje zda jsou předmětem zájmu myšlenkové procesy a uvažování nebo naopak zda jde především o chování subjektu. Z druhé strany lze posuzovat míru lidskosti či naopak míru inteligence. Systém je inteligentní, pokud za daných okolností udělá „tu správnou věc“.

Výše zmíněná kniha se věnuje přístupu inteligentního chování, protože inteligentní přístup je více obecný a především více vědecký než přístup odvíjející se z myšlenkových pochodů či lidského chování.

2.2 Inteligentní agent

Jak již bylo zmíněno v úvodu, agent je entita, která přijímá vjemy ze svého prostředí pomocí senzorů a provádí akce v tomto prostředí. Matematicky vzato je chování každého agenta popsáno funkcí, která každému vjemu přiřadí nějakou akci.

Inteligentní agent je nový pojem specifikující subjekt, který se chová tak, aby dosáhl nejlepšího výsledku. Abychom mohli určit, který výsledek je ten nejlepší, musíme ohodnotit jeho činnost. Ohodnocení je vždy závislé na dané situaci, nicméně základní pravidlo je, že by mělo být určeno podle toho, čeho chceme v prostředí dosáhnout, a ne podle toho, jak si myslíme, že by se agent měl chovat. Jako příklad je možno uvést prostředí z knihy AIMA VacuumWorld, kde agent-vysavač uklízí smetí na podlaze. Pokud by tento agent byl ohodnocen za každé uklizení nějakého množství smetí, nejlepší skóre by získal tak, že vše uklidí, vysype zpět na podlahu a uklízí znovu. Mnohem lepší způsob ohodnocení by tedy byl za čistou podlahu.

Inteligentní agent by měl pro každou posloupnost vjemů zvolit takovou akci, která podle předpokladu maximalizuje jeho ohodnocení v souladu s těmito vjemy a jakýmikoli dalšími znalostmi, které agent má.

2.3 Prostředí

Nejprve bychom měli definovat pojem *prostředí úlohy*. To bychom mohli nazvat „problémem“, ke kterému je inteligentní agent „řešením“. Prostředí úlohy je množina skládající se z ohodnocení, prostředí, agentových senzorů a jeho zařízení konajících akce. V návrhu agenta je

nutné jako první krok specifikovat prostředí úlohy co nejpřesněji.

Protože v oblasti umělé inteligence je mnoho různých možností specifikace prostředí úlohy, je vhodné jej klasifikovat podle několika faktorů. To nám umožní přesnější a jednodušší návrh agenta.

2.3.1 Prostředí plně nebo částěčně pozorovatelné

Plně pozorovatelné prostředí je takové, kdy senzory agenta mají přístup k úplnému stavu prostředí v každém okamžiku. Takové prostředí je výhodné, protože agent nepotřebuje interně udržovat informace o aktuálním stavu okolí. Prostředí může být částečně pozorovatelné například protože některé údaje senzory neposkytují, jako v případě VacuumWorld, kdy agent-vysavač má informaci o smetí pouze na místě, kde se právě vyskytuje, a neví, jestli je smetí i jinde.

2.3.2 Prostředí deterministické nebo stochastické

Pokud následující stav prostředí je přesně určen stavem současným a akcí konanou agentem, je toto prostředí deterministické, v opačném případě stochastické. V podstatě bychom mohli říct, že agent se v deterministickém a plně pozorovatelném prostředí nemusí zabývat pravděpodobností. Pokud je prostředí pozorovatelné částečně, může se jevit jako stochastické, zejména pokud je složité a není možné uchovat informace o nepozorovaných objektech. Proto je lepší o tomto rozdělení přemýšlet z pohledu agenta. Pokud je prostředí deterministické kromě akcí ostatních agentů, nazývá se strategické.

2.3.3 Prostředí epizodické nebo sekvenční

V epizodickém prostředí jsou agentovy zkušenosti rozděleny do atomických epizod, v každé agent vnímá a následně provede jednu akci. Podstatné je, že další epizoda nijak nezávisí na té předchozí. Typický příklad je agent zkoumající vadné výrobky na montážní lince, kde každé jeho rozhodnutí nemá vliv na ostatní výrobky. V sekvenčním prostředí, na druhou stranu, aktuální rozhodnutí může mít vliv na všechna další rozhodnutí. Epizodická prostředí jsou mnohem jednodušší, protože agent nemusí přemýšlet dopředu.

2.3.4 Prostředí statické nebo dynamické

Pokud se prostředí může změnit zatímco agent uvažuje, je pro agenta dynamické, v opačném případě statické. Se statickými prostředími se jednodušeji pracuje, protože agent nemusí sledovat okolní svět ve chvíli, kdy uvažuje nad akcí, ani si nemusí dělat starosti s plynutím času. Dynamická prostředí neustále po agentovi vyžadují nějakou akci a pokud se ještě nerozhodl, nedělá nic. Pokud se s plynutím času nemění prostředí ale agentovo skóre, je takové prostředí semidynamické, příkladem takového prostředí jsou šachy.

2.3.5 Prostředí diskrétní nebo spojité

Rozdělení na diskrétní nebo spojité může být použito na stav prostředí, způsob jak je vnímán čas nebo na agentovy vjemy a akce. Například šachová partie má konečný počet diskrétních stavů, stejně jako vjemů a akcí, narozdíl od řízení auta, kde rychlost i pozice jsou spojité veličiny. Vstup z digitální kamery je přesně vzato diskrétní, ale typicky je vnímán spojitě jako sled proměnlivých intenzit a pozic.

2.3.6 Prostředí s jedním nebo více agenty

Zdá se, že je velmi jednoduché klasifikovat prostředí jako single- nebo multi-agentní, nicméně je potřeba určit, zda ostatní objekty v daném prostředí je nutné považovat za agenty, nebo stochasticky chovající-se objekty. V prostředí s agentem A a objektem B můžeme říct, že B je agent, pokud by se jeho chování dalo popsat jako maximalizace ohodnocení, jehož hodnota závisí na chování agenta A. V partii šachu, například, se entita B snaží zvýšit svoje ohodnocení, a to, podle pravidel hry, snižuje ohodnocení agenta A. Takové prostředí je konkurenční multiagentní. Naopak při řízení vozidla, vyhýbání se srážkám vzájemně skóre zvyšuje, proto je to částečně kooperativní multiagentní prostředí. Částečně je to i konkurenční prostředí neboť např. na jednom parkovacím místě může stát pouze jedno auto. V multiagentních prostředích vyvstávají problémy s designem agentů, například při komunikaci mezi nimi.

2.4 Implementace agentů a jejich prostředí

Většina kódu je postavena na principu agentů, kteří konají akce na základě vjemů přijatých z prostředí, a prostředí, které reflektuje tyto akce změnami stavu. Všechny další algoritmy jsou odvozeny nebo připojeny k tomuto základnímu principu. Ten tudíž přímo vede k objektově orientované implementaci, kde lze s úspěchem využít abstrakci a především dědičnost. Pokud je například potřeba použít vyhledávací algoritmus, je vytvořeno obecné prostředí, jeho počáteční stav je nastaven na počáteční stav ze kterého se vyhledává, do tohoto prostředí je umístěn obecný agent a program tohoto agenta volá zvolený vyhledávací algoritmus. Agent poté provádí akce, postupně podle vygenerované posloupnosti, vedoucí k cílovému stavu. Ve Smalltalku lze toto provést na úrovni již existujících tříd pouze změnou jejich instancí, nicméně čistší cesta je vytvoření nové třídy pro prostředí, které je již pevně svázáno s daným problémem, tzn. potomka třídy obecné prostředí a vytvoření potomka třídy obecný agent, který je přímo uspůsoben k vyhledávání ve stavovém prostoru. A přesně tímto způsobem je vytvořen kód tohoto projektu. Zde jsou podrobně popsány třídy tvořící základy kódu projektu. Vzhledem k tomu, že Squeak nemá jmenné prostory, jsou jména všech tříd tohoto projektu doplněna prefixem „AIMA“, aby nedošlo k možnému konfliktu s již existujícími třídami systému. Tento prefix zde není uveden neboť zbytečně vede ke ztrátě přehlednosti. Toto platí i pro následující kapitoly.

2.4.1 Třída Environment

V hierarchii tříd reprezentujících prostředí je nejvýše *Environment*. Tato třída poskytuje mechanizmy pro běh prostředí, obsluhuje své agenty, předává jim vjemy, spouští jejich programy a provádí jejich akce. Rovněž zaznamenává skóre jednotlivých agentů. Každá instance této třídy v sobě uchovává kolekci agentů *agents*, kteří v prostředí působí, čítač kroků *steps*, maximální počet kroků *maxSteps*, po dosažení tohoto počtu kroků je prostředí ukončeno, a svůj aktuální stav *state*. Základní metoda této třídy je *run*, která spustí proces obsloužení jednotlivých agentů dokud není dosaženo maximálního počtu kroků *maxSteps* nebo ukončení simulace testem *isTermination*. V každém kroku metody *run* je každému agentovi předán jeho vjem pomocí metody *getPercept* a následně od něj vyžádána akce, ta je ve formě symbolu. Množinu všech možných akcí v daném prostředí vrací metoda *legalActions*. Potomci této třídy pak implementují jednotlivé akce jako metody (zprávy) a pomocí Smalltalkovské metody *perform: aSymbol* je tato akce provedena. Akce všech agentů

se provádějí až po vyhodnocení vjemů všech agentů v metodě *executeAgentActions*, ta je volána z *update*, kde se rovněž provádějí změny prostředí pro každý krok. Následně je všem agentům vyhodnoceno skóre pomocí metody *performanceMeasure*. Tím končí cyklus kroků a pokud není kladně vyhodnocena metoda *isTermination*, pokračuje cyklus dalším krokem.

2.4.2 Třída GridEnvironment

GridEnvironment je potomek obecného prostředí a reprezentuje prostředí s prostorem, který lze adresovat pomocí souřadnic x,y. Obecné prostředí vůbec pojem prostoru nezavádí. Na každé pozici v prostoru se může vyskytovat několik objektů, ty jsou podrobněji popsány níže. V množině *legalActions* této třídy jsou akce *turn*, *forward*, *grab* a *release*, které umožňují agentovi otočit se, jít dopředu a zvednout nebo pustit objekt vyskytující se na stejném místě jako agent, resp. v jeho kontejneru. Vzhledem k tomu, že se zde používá prostor, je třeba také vytvořit metody pro pohyb v něm, a to *gridAt:*, která vrací kolekci objektů na dané pozici, *gridAt:put:*, jež umístí objekt na danou pozici, a *gridAt:remove:*, která daný objekt ze zvolené pozice odebere. Dále je potřeba metoda, která hledá objekt na dané pozici *findObjectIf:at:* a také na sousedních pozicích *findNeighborIf:at:*. Jejich spojením je pak metoda *findObjectOrNeighborIf:at:*. Při vytvoření instance této třídy je nastavena velikost prostředí, souřadnice počátečního bodu a vytvořeny objekty pomocí třídy *ObjectSpec*, která je popsána níže.

2.4.3 Třída VacuumWorld

VacuumWorld je ukázkové prostředí pro agenta-vysavače, který se pohybuje v prostoru a vysává smetí, jeho skóre počítá metoda *performanceMeasure* v závislosti na počtu sesbíraných objektů - prachových částic. Při inicializaci je náhodně vytvořen prach na všech místech s pravděpodobností 25%. Mezi povolené akce patří kromě *turn* a *forward* také *suck*, která vysaje prach na pozici agenta, a *shutOff*, která agenta ukončí.

2.4.4 Třída WumpusWorld

WumpusWorld je rovněž ukázkové prostředí, tentokrát však jde o jeskyni obsahující kromě agenta také zlato, jámy a příšeru zvanou Wumpus. Agent musí na základě vjemů dosáhnout největšího skóre, v tomto případě to znamená najít zlato, nespadnout do jámy a nenechat se sežrat. V předefinované metodě *getPercept* je agentovi vrácen vjem na základě jeho okolí, pokud je v jeho sousedství Wumpus, obdrží symbol *#stench*, pokud je vedle jámy, dostane *#breeze* v případě, že na stejné pozici jako on je zlato, dostane *#glitter*, když narazí do zdi, je mu předáno *#bump* a pokud některý objekt v jeskyni vydá nějaký zvuk, dostane se mu *#sound*. Agentovi je předána kolekce všech symbolů, které jsou v danou chvíli pravdivé, a na jejich základě zvolí nějakou akci.

2.4.5 Třída Agent

Jako základní třída pro agenty slouží *Agent*. Obsahuje instanční proměnnou *program*, do které se ukládá blok s agentovým programem. Jeho nepovinným parametrem je kolekce vjemů a vrací symbol akce. Rovněž je možné přímo přetížit metodu *runProgram*, která tento blok vyhodnocuje, v případných potomcích se složitějšími programy. Dále každá instance uchovává v proměnných svoje skóre, jméno, aktuální vjem, akci, která je naplněna programem a své tělo, což je jeden z objektů v *GridEnvironment* a potomcích.

2.4.6 Třída AskUserAgent

AskUserAgent je potomek třídy *Agent*, který místo provedení svého programu vždy vyvolá dotaz na uživatele, zobrazí aktuální vjem a čeká na vložení akce, kterou má provést.

2.4.7 Třídy RandomVacuumAgent a RandomWumpusAgent

Tito potomci třídy *Agent* slouží k testovacím účelům pro *VacuumWorld* resp. *WumpusWorld*. Jejich program pouze náhodně vybírá akci z množiny všech možných akcí pro dané prostředí.

2.4.8 Třídy ReactiveVacuumAgent a ReactiveWumpusAgent

Tyto třídy agentů slouží k ukázce práce agenta ve *VacuumWorld* resp. *WumpusWorld*. Jejich program je v podstatě mapovací funkce, která na základě vjemů zvolí vhodnou akci za účelem získání největšího možného počtu bodů. Hlavní úkol agenta *ReactiveVacuumAgent* je vysávat smetí, čili pokud na něj narazí, vysaje jej. Pokud narazí do stěny, otočí se. V ostatních případech náhodně zvolí, jestli půjde dopředu nebo se otočí. Program agenta *ReactiveWumpusAgent* je trochu složitější, a to zejména z důvodu plánu akcí, který si agent interně uchovává. Pokud například narazí do stěny, otočí se dvakrát doprava a jde kupředu, čili se pohybuje od zdi pryč. Plán je řešen jako kolekce akcí, a pokud program hledá vhodnou akci a plán není prázdný, vrátí první akci z fronty. Pokud najde zlato, sebere jej. Vzhledem k tomu, že v prostředí *WumpusWorld* vydá zvuk pouze umírající Wumpus, pamatuje si agent také, jestli je Wumpus ještě naživu.

2.4.9 Třída Object

Object je základní třídou pro výskyt jakéhokoli objektu v prostoru prostředí. Každý objekt má definovaný tvar, barvu, velikost, kontejner pro další objekty, směr jakým je otočen a několik dalších atributů. Jsou nad ním definovány metody pro umístění v prostoru. Obsahuje také metody *isObstacle*, *isGrabable* a *isDeadly*, které vracejí pouze *true* nebo *false*, a slouží k jednoduché definici, co je objekt zač, zda je překážkou, jestli se dá zvednout, či je-li pro agenta smrtící. V následujících potomcích třídy jsou tyto metody jednoduše předefinovány.

2.4.10 Třída ObstacleObject

ObstacleObject je třída reprezentující překážku, konkrétně to může být např. zeď. Agent, který se pokusí přemístit na pozici s překážkou, obdrží vjem náraz (*#bump*).

2.4.11 Třída DeadlyObject

DeadlyObject je např. jáma nebo Wumpus a pokud se agent ocitne na pozici společně s objektem této třídy, je ukončen.

2.4.12 Třída AgentBodyObject

Protože i agent se musí vyskytovat na nějaké pozici v prostoru, je mu vytvořeno tělo, které je instancí této třídy. Toto tělo je při inicializaci přiděleno agentovi do instanční proměnné *body*.

2.4.13 Třída ObjectSpec

Poněvadž je potřeba pro každé prostředí s prostorem vytvářet určitou sadu objektů, je vhodné vytvořit třídu, která schopna tuto operaci nějakým způsobem usnadnit. Proto je třída *GridEnvironment* vytvořena tak, aby při inicializaci obdržela kolekci objektů typu *ObjectSpec* a tyto specifikace postupně aplikovala na sama sebe. Třída *ObjectSpec* očekává vložení typu objektu, který má reprezentovat pomocí metody *what*. Jako nepovinné údaje je možno vložit počet, pozici či pravděpodobnost vytvoření objektu pomocí metod *count*, *at* a *probability*. Pro zadání pozice obsahuje i speciální metody, které dokáží vytvořit objekt na všech pozicích prostoru, po okrajích, či náhodně: *atAll*, *atEdge* a *atFree*. Je tedy poměrně snadné vytvořit specifikaci pro zeď po okrajích prostoru, nebo prach všude s 25 procentní pravděpodobností. Při inicializaci prostředí je všem instancím třídy *ObjectSpec* zaslána zpráva *applyTo* s parametrem prostředí a ta pomocí metody *makeObject* a dříve zadaných specifikací vytvoří daný objekt. Třída zná pouze symbol tohoto objektu, samotné vytvoření instance zajistí třídní metoda daného prostředí, která je zavolána pomocí *perform*. Pokud je specifikováno, že objekt není lokální (metodou *isLocal*), čili nepatří-li přímo k danému prostředí, pak není volána třídní metoda prostředí, ale symbol je přímo nalezen v globálním slovníku *Smalltalk* a je vytvořena instance této třídy.

Kapitola 3

Řešení problémů

V předchozí kapitole byl ukázán inteligentní agent, který jednal na principu přímého přiřazení akce k určitému stavu. Takoví agenti nefungují příliš dobře v rozsáhlých prostředích, kde takové přiřazení akcí ke vjemům by bylo příliš složité a robustní. Na druhou stranu, agenti, kteří jsou zaměření na cíl místo konkrétního vjemu, mohou uspět díky zvažování budoucích možných akcí a vhodnosti výsledku těchto akcí.

3.1 Řešící agent

Jeden druh takového agenta se nazývá *řešící agent* a ten je předmětem zájmu této kapitoly. Řešící agenti se rozhodují co provedou na základě nalezení posloupnosti akcí, které vedou k žádaným stavům. Prvním krokem při tvorbě takového agenta je správná formulace cíle, který pomáhá agentovi hledat správnou cestu, vyloučí ostatní možnosti a tím značně zúží množinu stavů, do kterých agent může prostředí dostat. Za cíl můžeme považovat množinu stavů, kde každý stav splňuje podmínku, že problém je vyřešen. Protože agentův úkol je nalézt posloupnost akcí, které vedou k cílovému stavu, je nutné rozhodnout, s jakými akcemi a stavy může agent pracovat. To je důležité pro formulaci problému, zvolit určitou úroveň abstrakce, aby počet kroků k dosažení cíle nebyl příliš velký a bylo možné nalézt řešení. Proces hledání posloupnosti akcí vedoucích k cíli není přímý, existuje mnoho stavů, kde agent neví jak si vede, a proto je jeho úkolem vyzkoušet různé možné akce, které z daného stavu vedou, až do chvíle, kdy se dostane do stavu známého. Potom přehodnotí nalezené cesty a vybere tu nejlepší. Vyhledávací algoritmus obdrží problém na vstupu a jako výstup vrací řešení ve formě posloupnosti akcí. Jakmile je řešení nalezeno, nastává fáze provedení těchto akcí. Po formulaci cíle a řešeného problému, agent tedy zavolá vyhledávací algortimus, a poté provádí akce podle navráceného řešení, většinou to znamená, že provede první akci z posloupnosti a odstraní ji. Takto pokračuje dokud jsou ve frontě další akce. To samozřejmě znamená, že prostředí musí splňovat určitá kritéria. Především musí být statické, poněvadž agent při hledání cesty ani při provádění akcí již nebere ohled na prostředí. Jelikož se předpokládá, že počáteční stav je znám, prostředí musí být pozorovatelné. Při hledání je nutné vyčíslit všechny stavy, prostředí tedy musí být diskrétní. Nejdůležitějším faktorem je, že prostředí musí být deterministické, protože agent při provádění akcí ignoruje vjemy, nemůže tedy zpracovávat nepředvídané události.

Správná definice problému se skládá ze čtyř částí. První je počáteční stav, tj. stav prostředí, ve kterém agent začíná řešit problém. Další je popis možných agentových akcí, který je nejčastěji vyjádřen funkcí následovníka. Tato funkce vrací pro každý stav množinu

dvojic akce a následovníka, kde každá akce je jedna z přípustných akcí v tomto stavu a následovník je stav, do kterého se prostředí dostane po provedení akce. Počáteční stav a funkce následovníka dohromady definují stavový prostor problému, množinu všech stavů dosažitelných z počátečního stavu. Třetí část definice problému je cílový test, který určí, zda daný stav je cílovým stavem. Někdy je cílových stavů více a tento test jednoduše určí, jestli se prostředí nachází v jednom z nich. Ne vždy je cíl vyjádřen jako vyjmenovaná množina stavů, například v šachové partii nastane cílový stav vždy když král je ohrožen a nemá kam se schovat. Poslední částí definice je funkce určující cenu cesty. Řešící agent má obvykle funkci ceny založenou na jeho vlastním ohodnocení. Můžeme předpokládat, že cena cesty je součtem cen jednotlivých kroků, a funkci ceny kroku můžeme definovat jako $c(x,a,y)$, která vrací cenu provedení akce a ze stavu x do stavu y. Tyto části definice můžeme uložit do jedné datove struktury a předat je jako parametr algoritmu řešícímu daný problém. Výsledkem je pak cesta z počátečního do cílového stavu. Kvalita řešení je měřena pomocí funkce ceny cesty a řešení s nejnižší cenou je řešením optimálním.

3.2 Hledání řešení

Jakmile máme problém formulován, potřebujeme jej vyřešit. To je provedeno prohledáním stavového prostoru. Následující způsoby hledání využívají vyhledávací strom nebo vyhledávací graf, pokud je možné dosáhnout stejného stavu několika cestami. Kořenem tohoto stromu je uzel odpovídající počátečnímu stavu. Prvním krokem je kontrola, zda nejde o cílový stav, protože je nutné ošetřit i takové případy. V naprosté většině případů tomu tak není, a proto se pokračuje rozšířením aktuálního stavu aplikováním funkce následníka na tento stav a tím vygenerováním nové množiny stavů. Tímto jsme získali několik cest a musíme zvolit, kterou cestou se budeme nadále ubírat. Tato volba závisí na metodě hledání, jednotlivé způsoby budou popsány dále. Takto by se dal popsat obecný algoritmus pro prohledávání stavového prostoru. Jednotlivé uzly vyhledávacího stromu uchovávají tyto informace: stav, jeden z množiny stavového prostoru problému, rodičovský uzel, ze kterého byl tento uzel vygenerován, akce, která byla aplikována na rodiče a vedla k vytvoření tohoto uzlu, cena cesty z počátečního stavu až do tohoto uzlu a hloubka zanoření, tj. počet kroků z počátečního stavu. Výsledkem algoritmu pro řešení problémů je buď řešení nebo chyba, že řešení nebylo nalezeno, některé algoritmy mohou také skončit v nekonečné smyčce a vůbec nevrátit výsledek. Hodnocení algoritmu lze rozdělit do čtyř kritérií: úplnost, čili jestli algoritmus najde řešení pokud nějaké existuje, optimálnost, jestli najde nejlepší řešení, časová složitost a paměťová náročnost.

3.3 Neinformované metody hledání

Neinformované metody, nebo také slepé metody hledání, nemají o stavech žádné další informace, kromě zadání problému. Jediné co vědí o každém stavu je to, zda je či není cílový.

3.3.1 Breadth-first search

Breadth-first search (BFS), neboli slepé prohledávání do šířky, je jednoduchá metoda, pracující na principu expandování stromu postupně po úrovních, čili všechny uzly v dané hloubce jsou expandovány dříve, než uzly další úrovně. Je zřejmé, že tato metoda je úplná, optimální pouze v případě, že s narůstající hloubkou se nesnižuje cena kroku (například při

stejných cenách kroků), avšak je dost náročná na pamět neboť musí uchovávat množinu uzlů, ketré je třeba expandovat.

3.3.2 Uniform-cost search

Uniform-cost search (UCS) je narozdíl od předchozí metody optimální vždy, protože neexpanduje uzly v pořadí podle jejich hloubky, nýbrž podle nejnižší ceny cesty. Z toho vyplývá, že případě stejné ceny kroku u všech kroků je tato metoda identická s předchozí metodou. V případě, že by nějaký krok měl nulovou cenu, skončil by tento algoritmus v nekonečné smyčče. Nevýhoda této metody je, že často nejdříve expanduje mnoho uzlů s malými cenami před prozkoumáním ostatních cest s větší cenou, které mohou vést k řešení, což se odráží ve velké paměťové náročnosti.

3.3.3 Depth-first search

Depth-first search (DFS), neboli slepé prohledávání do hloubky, vždy expanduje nejhlouběji postavený uzel tak dlouho, dokud takový uzel nemá žádného následníka. Potom se metoda vrátí zpět ke druhému nejhlubšímu uzlu. Z toho vyplývají nevýhody této metody, může expandovat příliš hluboko, nebo i nekonečně hluboko, i když řešení se nalézá mnohem výše ve stromu v jiné větvi. Metoda tedy není úplná ani optimální. Má však velmi skromné paměťové nároky. Její varianta *Depth-limited search (DLS)* je založena na stejném pricipu, akorát obshauje omezení pro expanzi jen do určité hloubky. To znamená, že pokud je řešení níže, vůbec jej nenalezne. Obvykle se však dá určit kolik kroků je maximálně potřeba, a zamezí se tím možnost nekonečného expandování některé větve vyhledávacího stromu.

3.3.4 Iterative deepening depth-first search

Iterative deepening depth-first search (IDS) je metoda postupného zanořování založena na slepém prohledávání do hloubky. Pracuje na principu postupného zvyšování limitu zanoření a užití *DLS*, kombinuje výhody slepého prohledávání do hloubky i do šířky, není příliš paměťově náročná, je úplná a za stejných podmínek jako *BFS* je i optimální. Vzhledem k tomu, že stále postupně volá *DLS* pro stále větší limit hloubky zanoření, zdá se, že není příliš efektivní z důvodu neustálého opakování expanze vyšších uzlů, pravda je ovšem taková, že časová složitost *IDS* je menší než *BFS*, je tedy mnohem rychlejší. Obecně platí, že tato metoda je pro úlohy s velkým stavovým prostorem a neznámou hloubkou řešení ze všech neinformovaných metod nejlepší volbou.

Všechny tyto metody mohou při hledání narazit na významnou komplikaci - plýtvání času nalézáním již známých stavů. V některých problémech tato možnost nikdy nenastane, ale v řadě případů jsou akce vratné, jako například při hledání trasy. Stavové stromy jsou v těchto případech nekonečné, ale pokud budeme ignorovat již jednou expandované stavy, můžeme je změnšit na konečnou velikost. To ovšem znamená uchovat všechny již navštívené stavy v paměti, abychom je mohli porovnat s aktuálním stavem prostředí.

3.4 Informované metody hledání

Narozdíl od slepých metod prohledávání stavového prostoru, informované metody používají znalosti, které nejsou v definici problému, a mohou nalézt řešení mnohem efektivněji.

3.4.1 Best-first search

Best-first search (BestFS) je obecná metoda, která vybírá uzel pro expanzi na základě evaluační funkce *f(n)*. Obvykle je vybrán uzel s nejnižší hodnotou, protože evaluační funkce vyjadřuje vzdálenost od cíle. Ačkoli je v názvu slovo *best*, nevybírá vždy ten nejlepší uzel, pokud by to tak bylo, nebylo by to hledání, ale přímá cesta k cíli. Ovšem ne vždy je evaluační funkce přesná, a tak je vždy vybrán ten uzel, o kterém se předpokládá, že je nejlepší. Základem metody je heuristická funkce *h(n)*, která vrací onu předpokládanou vzdálenost od cíle. Pokud je *n* cílový stav, pak *h(n) = 0*.

3.4.2 Greedy best-first search

První variantou této metody je metoda *greedy best-first search*. Ta expanduje uzly pouze na základě jejich vzdálenosti od cíle, ohodnocuje tedy uzly pouze pomocí heuristické funkce, čili *f(n) = h(n)*. Ačkoli tato metoda ve většině případů používá pouze minimální velikost vyhledávacího stromu, není optimální ani úplná.

3.4.3 A* search

Nejznámější variantou *BestFS* metodou je *A* search*, která ohodnocuje uzly kombinací *g(n)*, ceny cesty k uzlu, s *h(n)*, předpokládané ceny cesty z uzlu do cíle. Znamená to, že *f(n)= g(n) + h(n)* a evaluační funkce vrací předpokládanou cenu řešení procházejícího uzlem *n*. Pokud heuristická funkce nikdy nepřecení cestu k cíli, nalezne tato metoda vždy optimální řešení. Její časová složitost není příliš velká, ale klade velké nároky na paměť neboť si uchovává všechny uzly, nehodí se proto pro řešení problémů příliš velkých rozměrů.

3.5 Hraní her

V minulé kapitole byla zmíněna multiagentní prostředí, ve kterých každý agent musí zvážit akce ostatních agentů a jak tyto akce ovlivní jeho hodnocení. V oblasti hraní her se převážně jedná o deterministické, plně pozorovatelné, konkurenční prostředí, kde se dva agenti střídají ve svých akcích, a výsledné ohodnocení jednoho je opačné k hodnocení druhého agenta. Například, pokud první hráč vyhraje partii šachu, druhý nutně prohrál. Hru jako takovou lze formálně definovat jako druh problému s následujícími prvky: počáteční stav, který udává stav hrací desky a který hráč je na tahu; funkci následníka, která vrací seznam párů $[tah, stav]$, kde každý pár vyjadřuje jeden z možných tahů a výsledný stav; test na konec hry a hodnotící funkce, která vrací číselné vyjádření koncového stavu. V šachu například tato funkce pro výhru, prohru nebo remízu vrací hodnoty +1, -1 nebo 0. V klasickém problému založeném na prohledávání stavového prostoru by optimálním řešením byla posloupnost tahů vedoucích k cílovému stavu, nicméně při hře má k tomu co říct i druhý hráč. Hráč A tedy musí najít strategii, která určuje jeho tah z počátečního stavu, pak tahy ze všech stavů, které mohly vzniknout tahem hráče B, dále tahy ze všech stavů, které mohly vzniknout tahem B na každý možný předchozí tah, a tak dále. Jinými slovy, optimální strategie vede k výsledkům ne horším než jakákoli jiná strategie, při které hráč hraje proti dokonalému protivníkovi.

3.5.1 Minimax

Metoda *minimax* vybírá nejlepší tah podle herního stromu, vytvořeného výše zmíněným hledáním všech možných tahů hráče i soupeře. Listy tohoto stromu jsou cílové stavy a každý je ohodnocen hodnotící funkcí. Potom je každý uzel stromu ohodnocen podle pravidla, že pokud se jedná o uzel hráče A, je mu přiřazena nejvyšší hodnota z ohodnocení následných uzlů, v případě uzlu soupeře B je to hodnota minimální, poněvadž platí, že čím vyšší ohodnocení, tím líp pro A, a naopak, čím nižší, tím líp je na tom B. Tím se ohodnocení dostane až ke kořenu stromu a podle něj hráč rozhodne, který tah provede. Jelikož je třeba vytvořit strom pro celou hru, je pro jiné než ty nejjednodušší hry obrovský a tato metoda prakticky nepoužitelná, nicméně slouží jako analytický základ pro použitelnější algoritmy.

3.5.2 Alpha-beta pruning

Metoda *alfa-beta řezu* dokáže eliminovat některé větve stromu a je tedy podstatně efektivnější než předchozí algoritmus. Vychází přímo z něj, nicméně větve, které nemohou ovlivnit ohodnocení uzlu, jsou z výpočtu vypuštěny. Pokud ve stromu existuje uzel n takový, že hráč může táhnout na tento uzel, a zároveň existuje lepší volba m buď u předka uzlu n nebo kdekoli výše ve stromu, pak uzlu n nebude při hře nikdy dosáhnuto. Metoda získala jméno podle dvou proměnných, *alfa* a *beta*, které uchovávájí hodnoty doposud nalezených nejlepších hodnot, *alfa* obsahuje pro hráče A nejvyšší hodnotu a *beta* nejlepší pro hráče B, čili nejnižší. Metoda postupně upravuje hodnoty alfa a beta jak prochází stromem a odřeže zbývající větve uzlu pokud zjistí, že ohodnocení aktuálního uzlu je horší, než aktuální alfa resp. beta pro hráče A resp. B. Efektivita metody je značně závislá na pořadí prohledávání jednotlivých uzlů. Pokud jsou nejdříve vyhodnoceny cesty, které se zdají být dobrým tahem, může tato metoda ve srovnání s *minimaxem* za stejný čas dosáhnout až dvojnásobné hloubky stromu.

3.6 Implementace

3.6.1 Třída Problem

Základní třídou pro řešení problémů pomocí prohledávání stavového prostoru je *Problem*. Obsahuje instanční proměnné *initialState*, ve které je uložen počáteční stav, a *goal*, ve které je uložen stav cílový. Problém je vyřešen, pokud metoda *goalTest* vrací *true*, což je při shodnosti předávaného a cílového stavu. Pokud nelze přesně definovat cílový stav je možné tuto metodu přetížit v potomku třídy. Třída sama o sobě je v podstatě abstraktní, protože nemá definovanou metodu *successors*, která vrací následné možné stavy ve dvojici s akcí. Tuto metodu si musí každý konkrétní problém, potomek této třídy, předefinovat. Řešení problému se spouští voláním metody *solve*. Tato třída rovněž obsahuje prohledávací algoritmy. Ty jsou implementovány pomocí kolekce uzlů, instancí třídy *Node*, ke které je přistupováno jako ke frontě, a každý algoritmus předepisuje v jakém pořadí je k uzlům ve frontě přistupováno. Třída zná následující algoritmy.

Slepé prohledávání do šířky provádí metoda *breadthFirstSearch*, která používá frontu typu FIFO. Stejný algoritmus, který ale navíc eliminuje návraty do předchozího stavu, je napsán pod názvem *noReturnsBreadthFirstSearch*. Test na opakování předchozího stavu zajišťuje třída *Node* metodou *isReturning*. Pro vynechání nejen opakujících se stavů, ale úplně všech duplicitních stavů ve vyhledávacím stromu slouží metoda *noDuplicatesBreadth-FirstSearch*. Slepé prohledávání do hloubky zajišťuje metoda *depthFirstSearch*, ta využívá

fronty s uzly jako zásobníku - LIFO. Jelikož se může snadno zacyklit, existuje její varianta s eliminací všech cyklů, která podobně jako v předchozím případě využívá funkčnosti třídy *Node*, ale metody *isLooping*, která kontroluje nejen předchozí na stav, ale všechny předchozí stavy a tím bezpečně detekuje jakýkoli cyklus. Tato metoda má název *noCyclesDepthFirstSearch*. Jako základ informovaných algoritmů vyhledávání je metoda *bestFirstSearch*, která přijímá jako parametr blok s evaluační funkcí. Ačkoli *uniformCostSearch* patří mezi neinformované algorimy, využívá *bestFirstSearch* tak, že jí předává funkci vracející cenu kroku. Metoda *greedySearch* vybírá uzly na základě předpokládané ceny cesty k cíli, předává tedy *bestFirstSearch* jako parametr heuristickou funkci. Algoritmus *aStarSearch* je na bázi *bestFirstSearch*, kde evaluační funkce sčítá aktuální cenu cesty a předpokládanou cenu do cíle.

3.6.2 Třída Node

Jednotlivé uzly jsou základními prvky pro uchování stavového prostoru a jeho procházení. Každý uzel představuje stav a má metodu *expand*, která vygeneruje uzly, ke kterým lze dojít z jeho stavu. Uchovává rovněž cenu aktuální cesty, přepokládanou cenu do cíle, aktuální hloubku zanoření a odkazy na následující uzly. Instance této třídy tvoří vyhledávací strom.

3.6.3 Třída Game

Game je třída určená pro popis a nalezení řešení při hraní her. Podobně jako pro problém, pro konkrétní hru je nutné vytvořit potomka a nadefinovat metody *makeMove*, *legalMoves* a *isGameOver*. První z nich určuje jak se určitým tahem změní stav hry, druhá vrací kolekci povolených tahů pro daný stav a třetí testuje zda existuje vítěz nebo je remíza. S touto třídou je pevně svázána třída *GameState* určující stav hry. Sama třída *Game* obsahuje implementaci několika algoritmů pro hraní her. Následují názvy jejich metod.

Metoda *minimaxDecision* je základní algoritmus, který prohledává postupně všechny stavy dokud nedojde k cíli a podle těchto informací vybere nejlepší tah. Ačkoli dojde vždy k nejlepšímu výsledku, vzhledem k jeho neefektivnosti je pro netriviální hry nepoužitelný. Metoda *minimaxCutoffDecision* prohledává stavový prostor podobně jako *minimax* ale pouze do n úrovní, poté na koncové stavy aplikuje evaluační funkci a na základě těchto informací rozhodne, jak táhnout. Podobně jako omezený minimax prohledává metoda *alphaBetaDecision* jen do určené úrovně a vrátí také stejný výsledek, ale prohledá méně uzlů, je tudíž efektivnější. Náhodný výběr z množiny možných tahů provádí metoda *pickRandomMove*. Aby mohl do hry zasahovat i uživatel, existuje metoda *askGameUser*, která vyzve uživatele, aby vybral jeden z množiny možných tahů.

3.6.4 Třída GameState

Třída *GameState* je určena pro uložení aktuálního stavu hry. Ve svých instančních proměnných uchovává hrací desku, seznam hráčů, jejich skóre a tah, který tento stav vytvořil. Seznam hráčů obsahuje na prvním místě vždy aktuálního hráče, tudíž postupně rotuje doleva jak se hráči střídají.

3.6.5 Prostředí

ProblemSolvingEnvironment je třída určená k řešení problémů. Každý problém má metodu *asEnvironment*, která vytvoří instanci tohoto prostředí. *ProblemSolvingAgent* je určen pro práci v prostředí pro řešení problémů, vlastní proměnnou *algorithm* a při inicializaci

prostředí je jeho program definován tak, aby nejprve nalezl řešení daného problému a poté vykonával krok po kroku až do cílového stavu. Třída *GameEnvironment* je vytvořena jako prostředí pro hraní her. Každá hra má metodu *asEnvironment*, která vytvoří instanci tohoto prostředí. *GameAgent* je určen pro hraní her, vlastní proměnnou *algorithm* a přetíženou metodu *runProgram*, která je upravena pro komunikaci s hrou a přímé volání algoritmu pro výpočet tahu podle agenta. *HumanGameAgent* a *RandomGameAgent* jsou třídy odvozené od *GameAgent*, používající algoritmus pro výběr tahu podle volby uživatele resp. náhodné volby.

3.7 Příklady

3.7.1 Misionáři a kanibalové

Pro ukázku řešení problémů byla zvolena klasická úloha, ve které je nutné přepravit z jedno břehu na druhý 3 misionáře a 3 kanibaly, přičemž loďka unese 2 osoby a na žádném břehu nesmí kanibalové nikdy přečíslit misionáře. Pro tento problém byla vytvořena třída *CannibalProblem*, která je potomkem třídy *Problem*. Pro jednodušší reprezentaci stavů byla vytvořena třída *CannibalState*, která popisuje počet jednotlivých osob na obou březích.

3.7.2 Piškvorky

Jako ukázka hraní her byly zvoleny piškvorky (*Tic-Tac-Toe*). Třída *TTTGame* definuje potřebné funkce pro zjištění možných tahů, vykonání tahu a testu na konec hry, rovněž evaluační funkci pro algoritmus alfa-beta. *AlphaBetaTTTAgent* využívá tohoto algoritmu pro výběr tahu.

3.7.3 Dáma

Další ukázkou hry využívající tento kód je Dáma (*Checkers*). Hra využívá alfa-beta řezu a je schopná hledat až 10 půltahů dopředu bez výrazného zpomalení hry. Každý stav šachovnice je ohodnocen podle počtu a polohy bílých a černých kamenů a dam, neboť herní strom není dopočítán až do konce hry a je tedy nezbytné listy omezeného stromu nějak ohodnotit. Hra obsahuje poměrně vydařené grafické prostředí, inspirované šachy, které jsou ve Squeaku standardně. Základní funkčnost zajišťuje třída *Checkers*, hra se spouští provedením výrazu *Checkers new*.

Kapitola 4

Logika

Tato kapitola se zabývá reprezentací znalostí a logickým myšlením, což jsou nezbytné součásti umělé inteligence. Agent, který využívá znalostí, je schopen dosáhnout svého cíle i v komplexních prostředích. Řešící agenti sice znají výsledek svých akcí a díky němu hledají řešení, ovšem takové znalosti jsou velmi specifické. Šachový program dokáže spočítat možné tahy krále, nicméně nemá ani ponětí o tom, že žádná figurka nemůže být na dvou místech zároveň. Oproti tomu *logický agent* může těžit ze svých znalostí mnohem obecněji. Znalosti také hrají velkou roli v částečně pozorovatelných prostředích, kde je agent může kombinovat s aktuálními vjemy a odvodit tak některé skryté aspekty současného stavu okolí před volbou své akce.

4.1 Úvod do logiky

Základní součástí logického agenta je báze znalostí, což je množina vět, které jsou vyjádřeny v jazyce pro reprezentaci znalostí a obsahují nějaké tvrzení o okolním světě. Jelikož potřebujeme způsob, jak do báze znalostí přidávat věty a jak z ní zjišťovat, co je známo, existují metody *tell* a *ask*. Obě tyto metody mohou využít odvození pro vytvoření nových vět z již existujících v bázi znalostí. Logický agent musí při odvozování dodržet pravidlo, že jakákoli odpověď na otázku musí logicky vyplývat z toho, co bylo do báze znalostí vloženo dříve. Kdykoli agent odvodí závěr z dostupných informací, je zaručeno, že tento závěr je správný, pokud jsou správné dostupné infomace. To je stěžejní vlastnost logického myšlení. Obecný program logického agenta je založen na následujícím principu. Jako každý jiný agent přijímá vjem na vstupu a jeho výstupem je akce. Vjem je nejprve uložen do báze znalostí metodou *tell*. Následně je proveden dotaz do báze (*ask*) a po vyhodnocení dotazu je navrácena akce. Tato akce je před provedením uložena do báze znalostí, jelikož je nutné zaznamenat, že tato akce skutečně byla provedena. Z uvedeného postupu vyplývá, že je možné vytvořit agenta pouze vložením vět do báze znalostí, což při vhodném reprezentačním jazyce může velmi zjednodušit návrh systému. Takový přístup se nazývá deklarativní. Oproti tomu procedurální přístup přímo ukládá požadované chování agenta jako kód programu, což může podstatně zlepšit efektivitu systému. V současné době je zřejmé, že návrh úspěšného agenta musí kombinovat oba tyto přístupy.

Jak již bylo řečeno, báze znalostí je množina vět. Každá věta musí být ve správném tvaru, který je určen syntaxí reprezentačního jazyka. Logika také definuje sématiku jazyka, tj. význam každé věty. Podle definice, sémantika jazyka definuje pravdivost každé věty pro každé možné prostředí. Například v aritmetice je věta $x + y = 4$ pravdivá v prostředí, kde

x je 2 a y je 2, ale není pravdivá v prostředí s $x = 1$ a $y = 1$. To znamená, že každá věta v logice musí být pro jakékoli prostředí pravdivá či nepravdivá. Logické myšlení agenta je založeno na logických důsledcích. Z věty A logicky vyplývá věta B, což znamená, že pro každý model (prostředí), ve kterém je A pravdivá, je B rovněž pravdivá. Nejjednodušším algoritmem pro odvozování je kontrola modelů, což je vlastně vyjádření všech možných modelů a porovnání, zda věta A je pravdivá ve všech modelech, ve kterých je pravdivá báze znalostí. Odvozovací algoritmus je bezesporný (*sound*), jestliže odvozené věty jsou vždy logickým důsledkem již známé věty, a úplný (*complete*), pokud dokáže odvodit každou logicky vyplývající větu z již známých vět v bázi znalostí.

4.2 Výroková logika

4.2.1 Jazyk výrokové logiky

Výroková logika je velmi jednoduchá, poslouží tedy dobře k vysvětlení základních prvků této oblasti. Jako první bude popsán jazyk výrokové logiky.

Syntaxe tohoto jazyka definuje správný tvar vět - formulí výrokové logiky. Nejmenší element jazyka je atomická formule (atom), což je výrokový symbol, který je buď pravdivý nebo nepravdivý. Existují dva symboly se zvláštním významem, vždy pravdivý - *True* a vždy nepravdivý - *False*. Formule se skládají z atomických formulí a logických spojek (negace, konjunkce, disjunkce, implikace a ekvivalence).

Sémantika jazyka definuje pravidla pro určení pravdivosti formule pro daný model, tj. nastaví hodnotu pravdivosti pro každý výrokový symbol. Hodnota prvotních formulí je dána modelem, jednoduché formule jsou vyhodnoceny podle pravdivostní tabulky a komplexní formule jsou pomocí rekurze rozloženy na jednoduché a vyhodnoceny podle předchozích pravidel.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

Tabulka 4.1: Pravdivostní tabulka

Báze znalostí obsahuje formule, které určují pravdivost báze pro daný model. Jelikož je nutné, aby všechny formule byly splněny, můžeme bázi brát jako jedinou formuli, která je konjunkcí formulí v bázi obsažených.

4.2.2 Odvozování

Cílem odvození je zjistit, zda daná formule logicky vyplývá z báze znalostí. Nejjednodušší algoritmus vychází přímo z principu logického důsledku, vyčíslení všech možných modelů a následná kontrola, jestli daná formule je platná v každém modelu, ve kterém je platná báze znalostí. Ve výrokové logice lze model vyjádřit jako přiřazení hodnoty pravda nebo nepravda pro každý výrokový symbol vyskytující se v bázi znalostí. Algoritmus pro kontrolu modelů tedy rekurzivně vyčíslí všechna možná přiřazení hodnot k proměnným a pro každou variantu zkontroluje platnost formule a báze znalostí. Tento algoritmus je bezesporný, jelikož je

založen přímo na definici logického důsledku, a úplný, protože vždy skončí, modelů je vždy konečný počet.

Výroková logika je pro mnoho úloh použitelná, nícméně není optimální pro prostředí neomezené velikosti, protože postrádá výrazové schopnosti pro vyjádření času, prostoru a vztahů mezi objekty. V takových případech je vhodné využít logiku prvního řádu (logiku predikátovou).

4.3 Predikátová logika

4.3.1 Jazyk predikátové logiky

Výrazové prostředky výrokové logiky, jak již bylo řečeno, nejsou bohaté natolik, aby s jejich pomocí bylo možno vyjádřit znalosti o komplexních prostředích. Jazyk predikátové logiky nám však umožňuje reprezentaci faktů ve srozumitelnější a mnohdy přesnější podobě. Narozdíl od modelu ve výrokové logice, který je množinou pravdivostních hodnot každého výrokového symbolu, obsahuje model predikátové logiky objekty. Tyto objekty mohou mít mezi sebou různé vztahy, některé z těchto vztahů jsou funkcemi, takové, kde existuje pouze jedna „hodnota“ pro daný „vstup“. Tyto relace mezi objekty jsou vlastně jen množiny n-tic objektů, které mají mezi sebou nějaký vztah. Základní složkou jazyka jsou symboly reprezentující objekty, vztahy a funkce. Tyto symboly jsou tedy rozděleny na individuové konstanty, predikátové a funkční symboly. Další součástí jazyka jsou termy, každý term je logický výraz pro určitý objekt. Každá individuová konstanta je tedy term, nicméně ne vždy máme zvláštní symbol pro každý objekt, a proto je term možné vyjádřit funkčním symbolem následovaným množinou termů v závorce, což jsou argumenty dané funkce. Narozdíl od procedurálních programovacích jazyků, fukční symbol neznamená žádné volání podprogramu, který předává návratovou hodnotu, je to pouze komplikovaný název pro objekt a umožňuje nám odvozovat fakta aniž bychom nějakou funkci definovali. Termy, které odkazují na objekty, a predikátové symboly, které značí relace, tvoří atomické formule, vyjadřující fakta. Atomická formule je pravdivá v daném modelu, pokud relace, ke které se vztahuje predikátový symbol, obsahuje objekty, ke kterým se vztahují argumenty tohoto symbolu. Stejně jako ve výrokové logice, složitější formule lze vytvořit pomocí logických spojek (negace, konjunkce, disjunkce, implikace a ekvivalence). Abychom mohli vyjadřovat vlastnosti nejen jednotlivých objektů, ale i kolekcí objektů, potřebujeme kvantifikátory. Univerzální kvantifikátor $\forall x$ uplatňuje daný výrok pro každý objekt v bázi znalostí nahrazením proměnné x ve formuli za jednotlivé objekty. Umožní tak vyjádřit obecná fakta o prostředí, jako např. $\forall x(medved(x) \wedge bily(x)) \Rightarrow ledni_medved(x)$. Podobně tak lze pomocí existenčního kvantifikátoru $\exists x$ učinit tvrzení o nějakém objektu aniž bychom jej jmenovali. Toto tvrzení je pak pravdivé, pokud v daném modelu existuje alespoň jeden objekt, po jehož přiřazení do proměnné x je formule kvantifikátoru platná.

4.3.2 Odvozování v predikátové logice

První možností, jak vyhodnotit výraz predikátové logiky, je převést jej na výraz výrokové logiky. K tomu je zapotřebí pravidel pro odstranění kvantifikátorů a predikátových symbolů. Univerzální kvantifikátor můžeme odstranit tak, že z dané formule substitucí volného termu za proměnnou získáme množinu formulí logicky vyplývajích z kvanfitifikované formule. Existenční kvantifikátor odstraníme nahrazením proměnné za symbol, který se doposud v bázi znalostí nevyskytuje. Posledním rozdílem od formulí výrokové logiky jsou predikátové sym-

boly, ty však můžeme brát jako obyčejné výrokové symboly. Nyní již stačí některá z metod pro řešení výrazů výrokové logiky.

Tento přístup je však neefektivní, a proto jsou častěji používána přímo pravidla pro odvození výrazů predikátové logiky. Jedním z nejdůležitějších je *unifikace*, což je vlastně nalezení a substituce termů za proměnné. Vstupem algoritmu jsou dvě klauzule a úkolem je najít substituci takovou, aby po jejím provedení byly klauzule shodné. Pokud je není možné unifikovat, vrací metoda prázdný výsledek.

Jednoduchou a účinnou metodou jak řešit výrazy predikátové logiky je algoritmus *forward-chaining*. Tato metoda je schopna efektivně vyhodnotit systém složený z Hornových klauzulí, což jsou disjunkce literálů, které obsahují nejvýše jeden pozitivní literál. Tyto literály mohou obsahovat proměnné, v tom případě jsou považovány za univerzálně kvantifikovné. Algoritmus pracuje na principu postupného vytváření nových faktů v bázi znalostí. Nejprve prochází všechny známá pravidla a pokud jsou jejich premisy splněny, je přidán nový fakt, který je výsledkem daného pravidla. Takto se pokračuje dokud není dotaz zodpovězen nebo dokud již nelze přidat nový fakt.

4.4 Implementace

4.4.1 Třídy a algoritmy výrokové logiky

Základní prvek výrokové logiky – báze znalostí, je zastoupen třídou *KnowledgeBase*. Obsahuje kolekci vět, které jsou vkládány metodou *tell:*. Jelikož jsou věty vkládány ve formě řetězců, je třeba je zpracovat, aby byly použitelné v implementovaných algoritmech. To provádí lexikální analyzátor *Lexer* ve spojení se syntaktickým analyzátorem *Parser*. Pro výrokovou logiku se používají potomci těchto tříd *PELexer* a *PEParser*. Výstupem parseru je instance třídy *Sentence* nebo některý z jejích potomků. Jsou to *TrueSentence* a *FalseSentence* pro vždy pravdivé a vždy nepravdivé formule, *AtomicSentence* pro atomické formule, *UnarySentence* pro unární výrazy (negace) a *BinarySentence* pro reprezenci formulí s binárním operátorem. V případě *UnarySentence* a *BinarySentence* jsou jejich operandy opět instancemi některé z potomků *Sentence* a tímto způsobem je složena jakákoli komplexní formule výrokové logiky. Abychom mohli bázi znalostí využít pro získávání faktů, je implementována metoda *askWithTTEntails:*, která vyhodnotí zadaný výraz pomocí kontroly modelů na základě pravdivostní tabulky. Samotný algoritmus implementuje třída *TTEntails* a její rekurzivní metoda *ttCheckAll:querySentence:symList:model:*, která vytvoří všechny kombinace hodnot všech symbolů báze znalostí a ty postupně testuje, zda zadaná formule je platná v modelu, ve kterém je zároveň platná báze znalostí. Příklad vyhodnocení formule výrokové logiky pomocí této metody lze spustit třídní metodou *LogicDemo ttEntailsDemo*.

4.4.2 Třídy a algoritmy predikátové logiky

Pro přidávání faktů do báze znalostí predikátové logiky *DLKnowledgeBase* metodou *add:* je nejprve nutné nadefinovat symboly, které se bázi budou vyskytovat, prostřednictvím třídy *FOLDomain* a jejich metod *addConstant:*, *addFunction:* a *addPredicate:*. Kolekce těchto symbolů pak využívá lexikální analyzátor *FOLLexer* pro správné rozpoznání jednotlivých tokenů, syntaktický analyzátor *FOLParser* pak podle těchto tokenů zpracuje zadaný výraz jako instanci *FOLSentence* nebo jejího potomka. Stejně jako ve výrokové logice, existují zde třídy pro jednotlivé typy výrazů (nebo jejich části): *FOLPredicate*, *FOLNotSentence*, *FOLParanthizedSentence*, *FOLConnectedSentence*, *FOLQuantifiedSentence* a

FOLTermEquality. Algoritmus pro vyhodnocení výrazu dopředným řetězením je implementován jako metoda *forwardChain:* na třídě báze znalostí *DLKnowledgeBase*, která postupně odvozuje nová fakta z již existujích pravidel a využívá pomoci unifikátoru *Unifier*. Příklady obsažené v knize AIMA [1] je možné spustit pomocí třídních metod *FOLDemo kingsDemo* a *FOLDemo weaponsDemo*.

Kapitola 5

Plánování

Plánování je úloha, jejímž řešením je posloupnost akcí, kterými je dosáhnut cílový stav. Zde jsou popsány úlohy v klasickém plánovacím prostředí, tj. plně pozorovatelném, deterministickém, statickém a diskrétním, v ostatních prostředích je třeba použít jiných algoritmů a agentů, přesahujících rámec této kapitoly. Pokud bychom pro řešení plánovací úlohy použili agenta prohledávajícího stavový prostor, byl byl zahlcen mnoha irelevantními akcemi. Například pokud bychom měli jako úkol koupit knihu, a existovala by akce *Koupit(isbn)*, která by jako parametr dostala desetimístné ISBN knihy, agent by hledal mezi 10 miliardami možností tu, která by měla jako výsledek stav *Vlastnit(ISBN0137903952)*, což je právě kniha, kterou chceme. Jednal by tak, protože neví, že akce *Koupit(x)* má výsledek *Vlastnit(x)*. Plánovací agent tuto znalost má, a proto může splnit tuto úlohu velmi snadno. Další potíž by byla najít správnou heuristickou funkci. Předpokládejme, že agent má za úkol koupit 4 knihy. Pro člověka to není problém pochopit, heuristická funkce pro ohodnocení stavu je počet již zakoupených knih, tohle ovšem vyhledávací agent netuší, jeho funkce testující cílový stav je pro něj pouze černá skříňka vracející pravdu či nepravdu. Proto nemůže být samostatný, člověk musí vytvořit heuristickou funkci pro každý jeho problém zvlášť. Oproti tomu plánovací agent vidí cíl jako jako konjunkci dílčích cílů, může tedy využít heuristiku nezávislou na prostředí, počet nesplněných dílčích cílů.

5.1 Jazyk plánovacích problémů

Základním jazykem klasických plánovacích úloh je jazyk *STRIPS* (je zkratkou pro STanford Research Institute Problem Solver). Jako první je nutné nějak reprezentovat stavy prostředí. Stav je rozložen na logické podmínky a vyjádřen konjunkcí pozitivních literálů výrokové nebo predikátové logiky, např. $Slavny \wedge Bohaty$ nebo $Na(kostka_1, stul) \wedge V_ruce(kostka_2)$. Formule predikátové logiky musejí být složeny pouze z atomických formulí neobsahujících individuové proměnné ani funkční symboly. Je předpokládáno, že všechy podmínky, které nejsou v daném stavu zmíněny, jsou považovány za nepravdivé. Dále jsou vyjádřeny cílové stavy. Cíl je částečně určený stav, vyjádřený konjunkcí pozitivních literálů. Stav s splňuje cíl c, pokud s obsahuje všechny atomy z c (může obsahovat i další jiné), například stav $Slavny \wedge Bohaty \wedge Nemocny$ splňuje cíl $Slavny \wedge Bohaty$. Jako poslední je potřeba popsat akce. Akce se skládá ze tří částí, názvu akce a definice parametrů, z podmínky, která musí být splněna před provedením akce, a výsledku, který udává, co se změní po provedení akce. Název a definice parametrů akce jsou ve tvaru např. $Zvedni(X, Y)$, podmínka je konjunkce pozitivních literálů neobsahujících funkční symboly, všechny proměnné v ní uvedené se

musí vyskytovat v definici akce, a výsledek je konjunkce literálů neobsahujících funkce popisujících změnu stavu po provedení akce. V některých příkladech bývá výsledek rozdělen na dvě množiny, jedna obsahuje pozitivní literály a udává, co je třeba ke stavu přidat, a druhá, obsahující negativní literály, předepisuje, které literály se stávají nepravdivými a mají být odstraněny. Rovněž ve výsledku se musí každá proměnná vyskytovat i v definici akce.

5.2 Řešení plánovacích problémů

Základní plánovací algoritmy pracují na principu prohledávání stavového prostoru. Jelikož mají akce problému definovanou podmínku a výsledek, je možné k vyhledávání přistoupit oběma směry, z počátečního nebo cílového stavu.

Prohledávání stavového prostoru je podobné algoritmům na řešení problémů, které byly zmíněny dříve. Je založeno na principu nalezení posloupnosti akcí, která vede z počátečního stavu do stavu cílového. Až donedávna se předpokládalo, že tento způsob řešení plánovacích problémů není natolik efektivní, aby byl použitelný v praxi. V posledních letech však byly objeveny heuristické funkce, které znatelně zvýšily efektivitu tohoto algoritmu.

Zpětné prohledávání stavového prostoru není při řešení obecného problému příliš použitelné, jelikož není vždy jasné jak vygenerovat předky cílových stavů. Při použití jazyka *STRIPS* je to však snadné, protože množiny stavů mohou být popsány literály, které musejí být v těchto stavech pravdivé. Hlavní výhoda této metody je to, že zkoumá pouze relevantní akce, čili takové, které končí ve stavu jednoho z dílčích cílů. Sice mohou existovat i nerelevantní akce, které vedou do cílového stavu, ale pokud existuje řešení, bude vždy, i bez takových akcí, nalezeno.

5.3 Implementace

Následující kód byl napsán za účelem vytvoření jazyka na bázi *STRIPS*,přičemž byl kladen důraz na co nejlepší využití objektové implementace a prostředků systému Squeak. Jazyk *STRIPS* využívá reprezentace stavů pomocí literálů. Takový přístup je vhodný zejména pro jazyky jako je Prolog. Pro Smalltalk by bylo výhodnější všechny tyto stavy reprezentovat objekty. A aby nebylo potřeba databáze stavů, je možné využít přímo objektů, se kterými je plánováno. Tento příklad využívá klasickou ukázku planování - přesouvání kostek na stole. Prostředí se skládá z několika kostek položených na stole. Je možné skládat kostky na sebe, ale pouze jednu na druhou, ne více. Nad stolem je robotická ruka, která může kostku zvednout a položit. Cílem je přeskládat kostky do jiné pozice.

5.3.1 Objekty

Všechny objekty „na stole“ jsou instance tříd se společným předkem *STRIPSObject*. Tato třída obsahuje metody pro zjištění a změnu stavu objektu. Metody pro zjištění stavu objektu jsou postaveny na následujícím principu: Pokud je stav, na který se dotazuji, pravdivý, vrátí *true*, jestliže není, vrátí množinu bloků obsahujících akce. Jakmile budou tyto akce provedeny, vrátí stav hodnotu *true*. Samotné akce pak mohou mít další podmínky, které při dotazu opět vracejí *true* nebo množinu bloků s akcemi.

5.3.2 Prostředí

STRIPSEnvironment obsahuje, kromě prvků obecného prostředí, instanční proměnné *table* a *goalStack*. Proměnná *table* ukazuje na objekt stůl, který je samozřejmě potomkem *STRIPSObject*. Stůl má od obyčejného objektu dvě výjimky, nemusí být prázdný, aby se na něj dalo něco položit, a zároveň jeho proměnná *onTop* není pouze ukazatel na objekt, ale na kolekci objektů. To znamená, že na něm může ležet více věcí zároveň. Druhá proměnná, *goalStack*, reprezentuje zásobník cílů. Každý z prvků tohoto zásobníku je typu *STRIPS-GoalSet*. Ten obaluje množinu bloků, přičemž jako takový vytváří ucelenou podmínku. Na dno zásobníku cílů se uloží požadovaný cílový stav.

5.3.3 Agent

Třída *STRIPSAgent* definuje metodu runProgram tak, aby mohla efektivně řešit plánovací úlohu podle *STRIPS*. Prověří podmínku na zásobníku, pokud je pravdivá, pak ji ze zásobníku odstraní. Prázdný zásobník značí konec, úloha byla úspěšně vyřešena. Pokud podmínka splněna nebyla, a byla přitom složena z několika bloků, pak je na zásobník uložena podmínka obsahující první nesplněný blok. Pokud se podmínka skládá pouze z jednoho bloku, pak je tento vyhodnocen a jeho výsledek, což musí být zase blok, poněvadž podmínka nebyla splněna, je uložen jako podmínka na zásobník.

Kapitola 6

Neurčitost a pravděpodobnost

V předchozích kapitolách bylo ukázáno, jak pomocí výroků ohodnocených jako pravdivé, nepravdivé nebo neznámé lze s logickým přístupem odvodit plán akcí, který vyřeší danou situaci. Ve skutečnosti bohužel agent téměř nikdy nemá přístup ke všemu, co se v jeho prostředí děje, a musí tedy jednat s jistou mírou neučitosti.

6.1 Neurčité znalosti

Abychom mohli vyjádřit neurčité znalosti, použijeme klauzule predikátové logiky a ke každé přiřadíme míru pravděpodobnosti daného výroku v rozmezí 0 až 1. Toto číslo nevyjadřuje stupeň pravdivosti tvrzení jako u fuzzy logiky, ale spíše míru přesvědčení, že se tento výrok stane pravdivým. Například pokud člověka bolí zub, má zubní kaz ($\forall p(Priznak(p, BolestZubu) \Rightarrow Nemoc(p, ZubniKaz))$). Tato věta není vždy pravdivá, ale my jsme přesvědčeni, že v 80% případů tomu tak je, a proto přiřadíme k této větě hodnotu 0,8. Tato hodnota může být získána například pomocí statistických měření (z pozorvání bylo zjištěno, že 80% pacientů trpících bolestí zubů má zubní kaz), obecných pravidel nebo kombinací některých faktů. Těchto 80% shrnuje všechny případy, kdy jsou přítomny faktory pro vznik kazu způsobující bolest, a zároveň případy ve kterých má pacient bolest a kaz, ale nezávisle na sobě. Zbývajících 20% zastupuje případy, ve kterých má pacient zubní kaz bez bolesti zubů. Pokud je toto číslo 0, znamená to, že agent je bezvýhradně přesvědčen, že výrok je nepravdivý, pokud je 1, je přesvědčen, že je to pravda. Agentovo přesvědčení o faktech okolního světa je založeno na vjemech z tohoto prostředí a může být průběžně upravováno. Při tažení z balíčku karet může být agent přesvědčen, že drží v ruce srdcové eso s pravděpodobností zhruba 2% (1/52), ale po shlédnutí karty tomuto výroku přiřadí 0 nebo 1.

Pro zápis a práci s neurčitými znalostmi je potřeba formální jazyk, který dokáže vyjádřit logické formule s mírou přesvědčení a závislost této hodnoty na již nabytých zkušenostech. Každý výrok obsahuje proměnnou vyjadřující neznámý stav, např. *ZubníKaz*. Tato proměnná má svůj obor hodnot, v případě zubního kazu je to pravda nebo nepravda, jindy může být diskrétní, či spojitý. Jednoduché výroky mohou tvořit složité pomocí standardních logických spojek.

Nepodmíněná pravděpodobnost přiřazená k výroku vyjadřuje stupeň pravděpodobnosti, pokud neznáme žádné další informace, a je zapisována jako $P(a)$, např. $P(ZubniKaz) = 0,1$ nebo $P(Pocasi = dest) = 0,2$. Rozložení pravděpodobnosti $\mathbf{P}(a)$ je vektor hodnot určující pravděpodobnost všech možných stavů proměnné a. Podmíněná pravděpodobnost

$P(a|b)$ určuje pravděpodobnost a, za předpokladu, že je nám známo b. Podmíněnou pravděpodobnost lze vyjádřit jako nepodmíněnou pomocí vzorce

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \tag{6.1}$$

který je také základem jednoduché metody pro odvozování neurčitých faktů. Jako bázi znalostí použijeme tzv. úplné spojené pravděpodobnostní rozložení (*full joint probability distribution*), což je rozložení pravděpodobnosti nad úplnou množinou proměnných prostředí. Máme tedy tabulku pravděpodobností všech kombinací stavů v prostředí a pomocí uvedeného vzorce můžeme spočítat podmíněnou pravděpodobnost některého stavu po pozorování dalších faktů. Tato metoda není v praxi příliš použitelná, jelikož není možné používat proměnné se spojitým oborem hodnot (tabulka by byla nekonečná) a také z důvodu velikosti pro větší počet proměnných (pro prostředí s n booleovskými proměnnými je potřeba počáteční tabulka o velikosti $O(2^n)$). V prostředí mohou existovat proměnné, které se vzájemně vůbec neovlivňují – jsou na sobě nezávislé. Takové proměnné mohou podstatně snížit velikost tabulky rozložení pravděpodobnosti, neboť dvě navzájem nezávislé proměnné můžeme rozdělit do dvou tabulek. Nezávislost proměnných a a b lze vyjádřit takto: $P(a|b) = P(a)$ nebo $P(b|a) = P(b)$ nebo $P(a \wedge b) = P(a)P(b)$. Jako příklad můžeme uvést třeba počasí a zubní kaz nebo postupné hody kostkou. V praxi se ovšem takové případy objevují zřídka, protože téměř vždy existuje nějaká spojitost.

Jedním z nejdůležitějších základních vzorců této oblasti je Bayesovo pravidlo.

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} \tag{6.2}$$

Na této jednoduché rovnici je postavena většina moderních systémů umělé inteligence pracujících na principu odvozování důsledků s neurčitými informacemi. Umožňuje spočítání neznámých pravděpodobností ze známých a pokud hledáme celé pravděpodobnostní rozložení určitého jevu, můžeme rovnici zjednodušit na $\mathbf{P}(Y|X) = \alpha\mathbf{P}(X|Y)\mathbf{P}(Y)$, kde α je tzv. normalizační konstanta. Problém nastává v případě, kdy chceme spočítat pravděpodobnost podle více než jednoho jevu, poněvadž pak potřebujeme znát podmíněnou pravděpodobnost konjunkce těchto jevů. Tím se dostáváme opět ke 2^n kombinací jevů, jejichž pravděpodobnost je třeba někam uložit. Jak bylo zmíněno výše, pomocí principu nezávislosti je možné vytvořit několik menších tabulek. Pokud jsou dvě proměnné vzájemně závislé pouze prostřednictvím nějaké třetí proměnné, můžeme jejich pravděpodobnost spočítat podle definice podmíněné nezávislosti $\mathbf{P}(X, Y|Z) = \mathbf{P}(X|Z)\mathbf{P}(Y|Z)$ a není tedy třeba ukládat pravděpodobnosti jednotlivých kombinací jevů.

6.2 Bayesovské sítě

Bayesovská síť je datová struktura reprezentující proměnné a jejich vzájemné vztahy, která na principu nezávislosti a podmíněné nezávislosti dokáže přesně specifikovat dané prostředí mnohem kompaktněji než úplné spojené pravděpodobnostní rozložení. Je to acyklický orientovaný graf, ve kterém proměnné prostředí tvoří uzly a orientované spoje mezi uzly definují vztahy, platí že šipka z uzlu X do uzlu Y znamená, že X je předkem Y. Každý uzel X_i má podmíněné pravděpodobnostní rozložení $P(X_i|Parents(X_i))$. Topologie sítě určuje podmíněnou nezávislost mezi proměnnými prostředí, šipky intuitivně naznačují jaký vliv na sebe uzly navzájem mají. Ke každému uzlu stačí definovat podmíněné

rozložení pravděpodobnosti pouze vůči svým předkům, z těchto definic je pak možné dopočítat libovolnou hodnotu úplného spojeného pravděpodobnostního rozložení, jelikož platí $\mathbf{P}(X_i, X_{i-1}, ..., X_1) = \mathbf{P}(X_i|Parents(X_i))$ za podmínky, že předkové X_i mají menší index než i.

Základním úkolem každého pravděpodobnostního odvozovacího systému je spočítat podmíněné pravděpodobnostní rozložení množiny dotazovaných proměnných po předání pozorované události, tj. přiřazení hodnot množině proměnných. Jelikož bayesovské sítě přesně určují úplné spojené pravděpodobnostní rozložení, je možné získat výsledek postupným sečtením pravděpodobností odvozených ze sítě podle výše uvedených pravidel. Časová náročnost této operace je v nejhorším případě stejná jako první odvozovací metody, nicméně prostorová náročnost je pouze lineární, jelikož je rozložení postupně sčítáno. Efektivitu odvozovací metody lze zlepšit eliminací opakovaného vyčíslení některých proměnných, ale i přesto je problém odvození v bayesovských sítích NP-složitý. Vyjímku tvoří pouze případy, kdy v síti existuje vždy nanejvýš jedna neorientovaná cesta mezi libovolnými dvěma uzly, pak je časová i prostorová složitost odvozovací metody pouze lineární.

6.3 Statistické metody

Vzhledem k malé efektivitě algoritmů pro exaktní odvozování ve velkých a složitě propojených sítích, je třeba zaměřit se na stochastické aproximační metody, založené na náhodném vzorkování. Základní aproximační metody patří do skupiny přímého vzorkování a jsou to *prior sample* a *rejection sampling*. Jsou postaveny na generátoru náhodných čísel, který vrací hodnotu v intervalu $[0, 1]$. Bayesovská síť je v topologickém pořadí vyhodnocena s použitím vygenerovaného čísla a s ohledem na již určené hodnoty předků daného uzlu. Druhá metoda využívá stejného principu, ale hledá pouze vzorky splňující požadovaná fakta, ostatní vzorky ignoruje. Podíl vzorků vyhovujícím faktům exponenciálně klesá s narůstajícím počtem faktických proměnných, algoritmus tudíž není vhodný pro složité problémy. Metoda *likelihood weighting* eliminuje problém zahazování nevhodných vzorků tím, že generuje pouze ty vhodné. Při postupném procházení sítí se generují hodnoty těch proměnných, které nejsou pevně nastaveny jako fakta, v opačném případě je hodnota přímo nastavena a zároven uložena podmíněná pravděpodobnost této hodnoty pro tuto proměnnou jako koeficient pro daný vzorek. Každý vzorek je nakonec upraven tímto koeficientem, který zaručí přiměřenou váhu jednotlivým vzorkům v závislosti na pravděpodobnosti jejich výskytu. Posledním algoritmem pro aproximační vyhodnocení bayesovské sítě je *Markov chain Monte Carlo*. Tato metoda, narozdíl od předešlých, nevytváří nové vzorky úplně od začátku, nýbrž vždy z předcházejícího vzorku, změnou hodnoty jedné z proměnných, která se nevyskytuje v zadaných faktech.

6.4 Implementace

6.4.1 Třída EnumerateJointAsk

Tato třída implementuje stejnojmennou metodu pro výpočet podmíněné pravděpodobnosti určitého jevu bez použití bayesovských sítí. Před použitím této metody je nutné naplnit tzv. úplné spojené pravděpodobnostní rozložení, které je reprezentováno třídou *ProbabilityDistribution*. Konstruktoru této třídy se předávají proměnné prostředí, a to buď jako jednotlivé parametry nebo jejich kolekce (existují metody *with:*, *with:with:*,

with:with:with:, *with:with:with:with:* a *withAll:*). Jednotlivé řádky tabulky rozložení jsou instance třídy *Row* a obsahují zadanou pravděpodobnost a *Model*, což je kolekce proměnných a jejich hodnot pro daný řádek. Tyto řádky se zadávají voláním zprávy *set:probability:* objektu *ProbabilityDistribution*. Nakonec je nutné položit otázku, co vlastně chceme zjistit a za jakých podmínek. To zajišťuje třída Query, která obsahuje dotazovanou proměnnou a kolekci nastavených proměnných. To celé je následně zpracováno v metodě třídy *EnumerateJointAsk ask:probabilityDistribution:* a výsledek nakonec ještě znormalizován. Ukázka použití této metody je *ProbabilityDemo enumerationJointAskDemo*.

6.4.2 Třída BayesNetNode

Základními elementy bayesovské sítě jsou její uzly, které zastupuje třída *BayesNetNode*. Každý uzel obsahuje kolekci svých předků i potomků, název své proměnné a tabulku pravděpodobnostního rozložení. Předek nebo předkové uzlu se nastavují voláním zprávy *influencedBy:*, která nejen doplní vzájemné vazby mezi uzly, ale naplní i tabulku rozložení (ve formě instance *ProbabilityDistribution*) podle předků uzlu. Samotná hodnota pravděpodobnosti v závislosti na hodnotě předka je nastavena metodou *setProbability:val:*, která doplní tabulku rozložení daného uzlu. Každý uzel má tedy tuto tabulku s počtem proměnných odpovídajícím počtu předků uzlu a s nastavenou pravděpodobností výskytu pro každou kombinaci hodnot proměnných. Vlastní hodnotu proměnné uzlu počítá metoda *isTrueFor:model:*, které je předána hodnota pravděpodobnosti a kolekce již nastavených uzlů, podle které je spočítána pravděpodobnost aktuálního uzlu, ta je porovnána se zadaným parametrem a podle výsledku vrácena hodnota uzlu. Kořenový uzel takto vytvořené sítě je předán konstruktoru třídy *BayesNet*. Vytvoření bayesovské sítě podle knihy AIMA str. 510 ukazuje metoda *ProbabilityDemo createWetGrassNetwork*.

6.4.3 Třída BayesNet

Jak vypovídá z názvu, tato třída reprezentuje bayesovskou síť a implementuje metody pro práci s ní. Základní metoda pro spočítání podmíněné pravděpodobnosti je *enumerateAsk:*, která, podobně jako třída *EnumerateJointAsk*, spočítá pravděpodobnosti podle tabulky rozložení a je tedy zástupcem exaktních výpočetních metod pro bayesovské sítě. Postupné rekurzivní procházení sítě při výpočtu zajišťuje metoda *enumerateAll:evidence:*. Náhodný vzorek stavu sítě lze získat voláním metody *getPriorSample*, která vygeneruje náhodný stav pro každý uzel sítě s ohledem na vzájemné závislosti. Tuto metodu využívá algoritmus *rejection sampling*, který na základě hledané proměnné, požadovaného stavu sítě a počtu pokusů statisticky určí pravděpodobnost výskytu požadovaného stavu. Vzorky, které nevyhovují zadaným podmínkám, jsou zahozeny. Spouští se voláním zprávy *rejectionSample:evidence:numSamples*. Na podobném principu funguje metoda *likelihoodWeighting:evidence:numSamples:*, která ovšem žádné vzorky nezahazuje, ale upravuje tak, aby splňovaly zadané podmínky, a místo přičtení 1 za každý vzorek obsahující hledanou proměnnou ve stavu *true* resp. *false* je přičten koeficient, který je násobkem pravděpodobostí zadaných podmínek, aby bylo statisticky vyváženo upravování vzorků. Poslední zmíněnou metodou této třídy je *mcmcAsk:evidence:numSamples:*, což je implementace algoritmu *Markov chain Monte Carlo*. Tento algoritmus nejprve vygeneruje náhodný první vzorek, vytvoří množinu uzlů, které nejsou zadány (*non-evidence*), a pro každý takový uzel spočítá *Markov blanket*, což je množina všech přímých předků, potomků a předků potomků, a použije jej jako počáteční podmínky pro generování nové hodnoty tohoto uzlu.

Tím postupně obměnuje hodnoty v množině *non-evidence* a vytváří tak nový stav ze stavu předchozího. Pro provedení n iterací vrací normalizovaný poměr hodnot *true* a *false*.

Příklady použití těchto algoritmů obsahuje třída *ProbabilityDemo*, jejíž třídní metody *priorSampleDemo*, *rejectionSamplingDemo*, *likelihoodWeightingDemo* a *mcmcAskDemo* používají předdefinovanou bayesovskou síť vytvořenou metodou *createWetGrassNetwork* a vyhodnocují zadaný dotaz pomocí výše popsaných algoritmů.

Kapitola 7

Učení

Tato kapitola se zabývá učením, jehož podstata spočívá ve zpracování vjemů a jejich využití v budoucnu. To vede ke zlepšení schopnosti jednat účelně a efektivně. Učení má mnoho podob, od prostého zapamatování zkušeností až po vytváření složitých teorií. V zásadě můžeme rozdělit učení do třech skupin, a to podle dostupné odezvy. Učení s dohledem (*supervised learning*) obnáší hledání určité funkce podle množiny příkladů jejích vstupů a výstupů. Tyto příklady mohou být zadány buď jako školicí množina, nebo v případě plně pozorovatelného prostředí může agent sám vnímat následky svých akcí a naučit se tak tyto následky předvídat v budoucnu. Učení bez dohledu (*unsupervised learning*) znamená využití metod učení pro zadané vstupy, ačkoli nejsou přesně definovány výstupy funkce. Takto můžeme naučit agenta rozlišit různé jevy, aniž bychom předem znali jejich definici. Poslední a nejobecnější skupinou je tzv. zpětnovazební učení (*reinforcement learning*), při kterém agentovi není přímo sděleno co má dělat, pouze je na konci ohodnocen (odměnou nebo pokutou). Důležitou roli hraje také vyjádření naučených informací, např. jako formule výrokové či predikátové logiky, nebo pravděpodobnostní popis ve formě bayesovské sítě.

7.1 Princip učení

Algoritmus pro deterministické řízené učení má za úkol pomocí indukce najít funkci h, která je aproximací neznámé funkce f. Protože je to učení s dohledem, jsou algortimu předány příklady, což jsou dvojice $(x, f(x))$, kde x je vstup a $f(x)$ výstup hledané funkce. Funkce h se nazývá hypotéza, konzistentní hypotéza je taková, která správně spočítá všechny zadané příklady. Složitost učení spočívá v tom, že není možné dost dobře říci, zda h je dobrá hypotéza, tj. jestli je schopná správně aproximovat f i pro další (pro ni neznámé) příklady. Pokud existuje více konzistentních hypotéz pro jednu množinu příkladů, je vhodné (podle Ockhamova pravidla) preferovat tu nejjednodušší hypotézu konzistentní s daty. Nalezení jednoduché a správné hypotézy závisí především na volbě prostoru hypotéz, který vyjadřuje určitou množinu funkcí, např. polynomické k-tého stupně. Pokud se skutečná hledaná funkce f nenachází v tomto zvoleném prostoru, není daný problém učení splnitelný. Jako řešení bychom mohli zvolit co nejobecnější prostor hypotéz, kterým bezesporu je třída všech Turingových strojů, neboť každá spočitatelná funkce může být vyjádřena nějakým Turingovým strojem, to ale bohužel není možné, protože by tímto došlo k obrovskému nárůstu výpočetní složitosti učicího algorimu. Druhým důvodem proč volit jednoduchý prostor hypotéz je následné používání nalezené hypotézy, výpočet $h(x)$ je mnohem rychlejší v případě lineární funkce než v případě programu Turingova stroje. Z těchto

důvodů pracují následující algoritmy s jednoduše vyjádřenými funkcemi, a to konkrétně ve formě klauzulí výrokové logiky. Efektivitu učicího algoritmu můžeme změřit pomocí tzv. učicí křivky, která ukazuje procentuální přesnost předpovědi pro testovací množinu v závislosti na velikosti trénovací množiny.

7.2 Rozhodovací stromy

Rozhodovací stromy (*decision trees*) patří k těm nejjednodušším, ale i přesto k nejlepším z učicích algoritmů. Slouží jako vhodný úvod do této oblasti a zároveň nejsou složité na implementaci. Vstupem rozhodovacího stromu je objekt nebo situace popsaná množinou atributů a návratovou hodnotou je rozhodnutí, což je předpokládaný výsledek na základě vstupu. Ačkoli atributy mohou mít diskrétní nebo spojitý obor hodnot, všechny zde uvedené induktivní algoritmy pracují s danou množinou příkladů s diskrétním (konkrétně booleovským) oborem hodnot. Rozhodovací strom pracuje na principu postupného provádění posloupnosti testů, kde každý uzel stromu představuje jeden takový test na hodnotu jednoho z atributů a z něj vedoucí větvě zastupují možné hodnoty daného atributu. Každý koncový list určuje rozhodnutí - návratovou hodnotu algoritmu. Taková reprezentace je člověku velmi blízká a bývá použita v různých návodech. Vyjadřovací schopnosti rozhodovacích stromů plně odpovídají prostředkům výrokové logiky, tj. každou booleovskou funkci je možné zapsat jako rozhodovací strom. Triviálně jako každý řádek pravdivostní tabulky tvořící jednu cestu stromem, ale to by znamenalo exponenciální nárust velikosti stromu v závislosti na počtu atributů, naštěstí však lze většinu funkcí vyjádřit mnohem jednodušším stromem.

Učení booleovského rozhodovacího stromu probíhá nad trénovací množinou, což je množina příkladů $(X_1, y_1), ..., (X_i, y_i)$ složených z vektoru vstupních atributů X a výstupní hodnoty y. Problém nalezení takového stromu, aby byl konzistentní s příklady, může být triviálně vyřešeno vytvořením cesty od kořene k listu pro každý příklad zvlášť. Takový strom, který si pouze zapamatuje trénovací množinu, není schopen odhalit jakýkoli vzor chování hledané funkce, a proto od něj nelze očekávat správné odhadnutí příkladů, které jej nebyly naučeny. Je tedy třeba tvořit strom jinak, aby byl co nejjednodušší a zároveň konzistentní s příklady. Základem je testovat nejprve takový atribut, který co nejlépe rozdělí trénovací množinu podle své hodnoty na několik podmnožin. Zjednodušeně řečeno, čím více stejných výstupních hodnot je v každé podmnožině, tím je atribut vhodnější. Na každou takovou podmnožinu příkladů se aplikuje další test na další vhodný atribut, dokud všechny příklady v podmnožině nemají stejnou výslednou hodnotu. Tímto způsobem můžeme vytvořit strom s dostatečně malým počtem testů a získat tak velmi jednoduchou hypotézu konzistentní s daty. Výběr vhodného atributu je založen na množství informace, které daný atribut poskytuje, a počítá se následujícím způsobem. Pokud každá možná odpověď v_i má pravděpodobnost $P(v_i)$, pak množství informace I určíme jako

$$I(P(v_1), ..., P(v_n)) = \sum_{i=1}^{n} -P(v_i) log_2 P(v_i) \tag{7.1}$$

a pokud trénovací množina obsahuje p pozitivních odpovědí a n negativních, pak množství informace ve správné odpovědi je

$$I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n} log_2 \frac{p}{p+n} - \frac{n}{p+n} log_2 \frac{n}{p+n}. \tag{7.2}$$

Každý atribut A rozdělí trénovací množinu E na podmnožiny $E_1, ..., E_v$ podle jejich hodnot atributu A, přičemž A může mít v různých hodnot. Každá podmnožina E_i obsahuje pak p_i kladných a n_i záporných odpovědí, z čehož opět můžeme získat hodnotu informace. Průměrné množství informace ze všech podmnožin je hodnota, kterou potřebujeme získat po otestování atributu A. Z toho všeho vyplývá, že informační zisk atributu A je

$$Zisk(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}) \tag{7.3}$$

a nám nezbývá nic jiného, než najít atribut s největším ziskem a dosadit jej na kořen rozhodovacího stromu. Takto postupně vytvoříme celý strom, který je samozřejmě pouze aproximací hledané funkce na základě zadaných příkladů, může se tedy, v závislosti na velikosti trénovací množiny, i velmi lišit od skutečné funkce. Mezi časté problémy patří nedostupnost všech různých hodnot všech atributů v testovací množině, nebo nepodstatné atributy komplikující strom aniž by nesly nějakou užitečnou informaci.

7.3 Další metody učení

Skupinové učení (*ensemble learning*) pracuje na principu výběru kolekce hypotéz a spojení jejich predikcí. Například soubor složený z 5 hypotéz, kde výsledná předpověd je ta s největším výskytem ve skupině, potřebuje pro chybnou klasifikaci příkladu alespoň 3 chybné klasifikace ve skupině, přičemž pravděpodobnost tohoto jevu je mnohem menší než v případě použití pouze jedné hypotézy. Dalším příkladem jak může tato metoda zlepšit výsledky je rozšíření prostoru hypotéz. Pokud každá jednotlivá hypotéza je z určitého jednoduše vyjádřeného prostoru hypotéz, pak se na jejich složení můžeme dívat jako na jedinou hypotézu patřící do obecnějšího prostoru hypotéz aniž bychom výrazně zvýšili výpočetní složitost. Nejrozšířenější metodou z této oblasti je posilování (*boosting*). Každý příklad má nastavenou váhu, na počátku je to 1. Čím vyšší tato váha je, tím více je na příklad kladen důraz při generování nové hypotézy. Po vygenerování první hypotézy je správně klasifikovaným příkladům váha snížena, v opačném případě zvýšena. Z této upravené vážené trénovací množiny je vytvořena další hypotéza a postup se opakuje, dokud skupina neobsahuje M hypotéz, kde M je vstupem posilovacího algoritmu. Výsledná skupinová hypotéza je kombinací všech M hypotéz, kde výsledný předpoklad je určen jako vážená většina ze všech hypotéz, přičemž vážení je provedeno na základě úspěšnosti klasifikace na trénovací množině.

Další implementovanou metodou učení je rozhodovací seznam (*decision list*). Skládá se z posloupnosti testů, přičemž každý test je konjunkcí literálů. Pokud je test splněn, je navrácena hodnota určená seznamem pro daný test. Pokud splněn není, pokračuje se dalším testem ze seznamu. Tato metoda je podobná rozhodovacím stromům, ale její obecná struktura je mnohem jednodušší, ačkoli jednotlivé testy mohou být složitější. Princip tvorby takového seznamu spočívá v nalezení co nejjednoduššího testu, který je splnitelný pro nějakou (co největší) množinu příkladů se stejným výsledkem z trénovací množiny. Tento výsledek je pak přiřazen k testu a příklady jsou odebrány z trénovací množiny. Postup se opakuje dokud množina obsahuje nějaké příklady. Testy mohou být tvořeny například postupným zkoušením všech atributů a všech jejich hodnot.

7.4 Implementace

7.4.1 Třída DataSet

Základní objektem pro uchování množiny příkladů je třída *DataSet*, která obsahuje kolekci příkladů (*Example*) a jejich popis (*DataSetSpecification*). V popisu datasetu jsou určeny názvy všech atributů, případně i jejich možné hodnoty, a zároveň je zde definován cílový atribut *target*. Každá definice atributu je podle datového typu vyjádřena objektem třídy *StringAttributeSpecification* resp. *NumericAttributeSpecification*, což jsou potomci třídy *AttributeSpecification*. Pokud je atribut typu řetězec, je zároveň definována množina všech jeho možných hodnot. Každý příklad uložený v datasetu je popsán kolekcí atributů *StringAttribute* resp. *NumericAttribute*, ve kterých je uložena hodnota atributu pro daný příklad. Aby bylo možné jednoduše naplnit dataset ze souboru, existuje třída *DataSetFactory*, obsahující třídní metodu *fromFile:specification:separator:* a další dvě metody pro naplnění dat z přiložených ukázkových souborů restaurant.csv a iris.csv, které jsou součástí AIMA a jsou dostupné na webových stránkách [3]. Třída *DataSet* rovněž obsahuje metody pro vyhodnocení množství informace z uložených příkladů *getInformationFor* a zjištění zisku informace pro daný atribut *calculateGainFor:*. Obě tyto metody jsou implementovány podle výše uvedených vzorců.

7.4.2 Třída DecisionTreeLearner

První metodou pro učení je vytvoření rozhodovacího stromu, což zajišťuje třída *DecisionTreeLearner*. Jako u každého potomka třídy *Learner* se učení provede metodou *train:*, které je předán dataset s příklady. Tato metoda postupně vytvoří strom rekurzivním voláním metody *decisionTreeLearning:attrNames:defaultTree:*, která vždy na základě informačního zisku vybere atribut a vytvoří pro něj uzel stromu. Jednotlivé uzly jsou instance třídy *DecisionTree*, listy stromu tvoří speciální varianta této třídy, její potomek *ConstantDecisionTree*. Výsledný (naučený) strom pak zajišťuje metodou *predict:* vyhodnocení cílového atributu podle zadaného příkladu tak, že postupně prochází od kořene až k listům podle hodnot ostatních atributů, až na konci *ConstantDecisionTree* vrátí hledanou hodnotu. Pro ověření výsledného stromu implementuje *Learner* metodu *test:*, která vyhodnotí úspěšnost klasifikace algoritmu pro zadaný dataset. Příklad s použitím tohoto algoritmu lze spustit voláním metody *LearningDemo decisionTreeDemo*.

7.4.3 Třída AdaBoostLearner

Tato třída reprezentuje algoritmus pro skupinové učení. Nejprve je třeba vytvořit kolekci velmi jednoduchých rozhodovacích stromů, které mají pouze kořen a umí tedy rozhodovat pouze na základě jednoho atributu a jedné jeho hodnoty. Tyto stromy jsou vytvořeny pro každou hodnotu každého atributu v datasetu pomocí třídní metody *getStumpsFor:retValIfMatch:retValIfUnmatch:* třídy *DecisionTree*. Každý takový strom je předán instanci *StumpLearner*, což je potomek třídy *DecisionTreeLearner*, který pouze vrací výsledek podle svého jednoduchého rozhodovacího stromu. Kolekce těchto instancí společně s datasetem jsou následně předány konstruktoru třídy *AdaBoostLearner*. Samotné učení (*train:*) probíhá postupným upravovováním vah jednotlivých hypotéz (*StumpLearner*) v závisloti na velikosti chyby klasifikace (metoda *calculateError:*). Jakmile mají všechny hypotézy váhy upraveny, vrací *AdaBoostLearner* v metodě *predict:* váženou většinu „hlasů“

svých hypotéz pomocí funkce *weightedMajority:*. Příklad s použitím tohoto algoritmu lze spustit voláním metody *LearningDemo ensembleLearningDemo*.

7.4.4 Třída DecisionListLearner

Učení pomocí rozhodovacích seznamů implmentuje třída *DecisionListLearner*, jejíž metoda *train:* volá rekurzivní funkci *decisionListLearning:*, která nejprve vygeneruje seznam všech testů a z nich vybere takový, který je platný na daném datasetu, tj. existují příklady splňující test, které mají stejný výsledek (hodnotu cílového atributu). Tento výběr provádí metoda *getValidTest:dataSet:*. Generování testů probíhá iterací nad všemi necílovými atributy a jejich hodnotami, vytvoří se vždy test zda daný atribut je roven dané hodnotě. Jednotlivé testy jsou instance třídy *DLTest*, jejíž třídní metoda *createDLTestsWith-AttributeCount:* pak podle zadaného datasetu tuto množinu vytvoří. Vybraný test je přidán do kolekce objektu *DecisionList* a zbývající příklady, které test nesplňují, jsou předány opět metodě *decisionListLearning:*, dokud nějaký zbývá. Takto je vytvořen seznam testů, který je použit při vyhodnocování (*predict:*) postupným testováním. Hodnota prvního testu, který vyhovuje pro zadaný příklad, je vrácena. Příklad s použitím tohoto algoritmu lze spustit voláním metody *LearningDemo decisionListDemo*.

Kapitola 8

Závěr

Hlavním cílem této práce bylo vytvořit implementaci algoritmů pro řešení úloh z oblasti umělé inteligence podle vzorových příkladů knihy AIMA [1]. Ačkoli tyto příklady již existují ve třech implementacích do různých jazyků, mým zájmem bylo vytvořit ekvivalentní metody v jazyce Squeak Smalltalk z několika důvodů. Předně proto, aby potencionální čtenář této knihy měl možnost většího výběru z nabízených implementací, protože si myslím, že Squeak má v sobě obrovský potenciál – především pro takové účely, jako je tento – díky čistě objektově orientovanému přístupu, velmi dobré čitelnosti a velice snadné syntaxi není složité se v něm rychle zorientovat. I přes tyto klady to není příliš známý a rozšířený jazyk, a proto by publikování vytvořeného kódu na webové stránce knihy [3] mohlo přispět k jeho zviditelnění. Další přínos vidím v uvedení těchto algoritmů na *SqueakMap*, což je Squeakovská repository, protože podobných knihoven pro řešení úloh umělé inteligence je zatím v tomto jazyce poskrovnu.

Součástí práce je připravený soubor image, který nejen obsahuje vytvořený kód, ale i jednoduché prostředí pro prezentaci těchto algoritmů s odkazy nejen pro jejich spuštění ale i s odkazy přímo do kódu vybraných algoritmů pro rychlý a jednoduchý přístup k vytvořeným metodám. Jako demonstrační program vytvořený ve Squeaku jsem naprogramoval hru Dáma (*Checkers)*, která využívá agenta a prostředí z kapitoly 2 a algoritmus na hraní her alfa-beta z kapitoly 3. Hra slouží především jako ukázka toho, že lze tento kód bez problémů použít i v praxi.

Další možností, jak využít těchto algoritmů, je vytvořit na jejich základě plnohodnotnou e-learningovou aplikaci, což by obnášelo zpracování přiložených příkladů ve Squeaku přirozené grafické podobě a použití některých textů např. z této práce k výkladu. Já jsem zvolil jiný přístup, a tím je teoretický popis jak implementované algoritmy fungují, aby si případný zájemce o tuto problematiku mohl ve zkrácené podobě přečíst jak dané postupy fungují a na co by si měl dát pozor při jejich použití (např. při tvorbě vlastní aplikace) aniž by musel hledat v jiných zdrojích.

Jelikož byla mým stěžejním zdrojem informací právě kniha AIMA [1], vyskytují se v textu některé anglické pojmy z oblasti UI, ke kterým neexistuje český ekvivalent, a u některých pojmů (např. kde český název přesně nevystihuje význam slova) je z důvodu jednoznačnosti a přesnosti v závorce uveden původní anglický název.

Výzkum umělé inteligence pokrývá mnoho oblastí a ty stěžejní z nich jsem se v této práci pokusil uvést, ve formě teoretického výkladu i praktické ukázky, a pevně věřím, že pro případné zájemce o studium umělé inteligence bude tato práce přínosem.

Příloha A

Následující tabulka ukazuje algoritmy implementované v jazyce Squeak Smalltalk, umístění jejich pseudokódu v knize a stav implementace v ostatních jazycích.

Ozn.	Strana	Název v knize	Druh	Python	Java	Lisp	Smalltalk
2	32	Environment	typ	√		√	√
2.1	33	Agent	fn	√	√	√	√
2.10	47	Simple-Reflex-Agent	fn	√			√
3.1	61	Simple-Problem-Solving-Agent	fn	√		√	√
3	62	Problem	typ	√	√	√	√
3	69	Node	typ	√	√	√	√
3.7	70	Tree-Search	fn	√	√	√	√
3	71	Queue	typ	√	√	√	√
3.9	72	Tree-Search	fn	√	√	√	√
3.13	77	Depth-Limited-Search	fn	√	√	√	√
3.14	79	Iterative-Deepening-Search	fn	√	√	√	√
4	95	Best-First-Search	fn	√	√	√	√
4	97	A*-Search	fn	√	√	√	√
6.3	166	Minimax-Decision	fn	√	√	√	√
6.7	170	Alpha-Beta-Search	fn	√	√	√	√
7	195	KB	typ	√		√	√
7.7	205	Propositional Logic Sentence	typ	√	√		√
7.10	209	TT-Entails	fn	√	√	√	√
9	273	Subst	fn	√	√	√	√
9.1	278	Unify	fn	√	√	√	√
9.3	282	FOL-FC-Ask	fn	√	√		√
13.4	477	Enumerate-Joint-Ask	fn		√		√
14.12	512	Prior-Sample	fn		√		√
14.13	513	Rejection-Sampling	fn		√		√
14.14	515	Likelihood-Weighting	fn		√		√
14.15	517	MCMC-Ask	fn		√		√
18.5	658	Decision-Tree-Learning	fn	√	√	√	√
18.10	667	AdaBoost	fn		√		√
18.14	672	Decision-List-Learning	fn		√	√	√

Literatura

[1] Russel, S. J., Norvig, P. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003. ISBN 0-13-790395-2.

[2] Křivánek, P. Squeak: návrat do budoucnosti. `http://www.root.cz/clanky/squeak-navrat-do-budoucnosti/`.

[3] WWW stránky. Artificial intelligence: A modern approach. `http://aima.cs.berkeley.edu/`.

[4] Zbořil, F. Přednášky k předmětu umělá inteligence. `http://www.fit.vutbr.cz/study/courses/UIN/ (květen 2005)`.