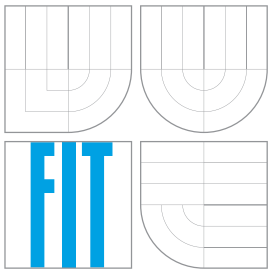
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTELIGENTNÍ REAKTIVNÍ AGENT PRO HRU MS. PAC-MAN

INTELLIGENT REACTIVE AGENT FOR THE GAME MS. PACMAN

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
BARBORA BLOŽOŇOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR
doc. Ing. MARTIN DRAHANSKÝ, Ph.D.

BRNO 2016

Abstrakt

Tato práce se zabývá umělou inteligencí pro složitější rozhodovací problémy, jako je hra s neurčitostí Ms. Pacman. Cílem práce je navrhnout inteligentního reaktivního agenta využívajícího metodu posilovaného učení, demonstrovat jej ve vizuálním demu Ms. Pacman a jeho inteligenci srovnat se známými informovanými metodami hraní her (Minimax, Alfa-Beta řezy, Expectimax). Práce je rozdělena primárně na dvě části. V teoretické části je řešena problematika metod hraní her, reaktivita agenta a možnosti posilovaného učení (vše v kontextu Ms. Pacman). Druhá část práce je zaměřena na samotný popis návrhu a implementace verzí agenta a na závěr jejich experimentální srovnání se zmíněnými známými metodami hraní her, zhodnocení dosažených výsledků a několik návrhů na vylepšení do budoucna.

Abstract

This thesis focuses on artificial intelligence for difficult decision problemes such as the game with uncertainty Ms. Pacman. The aim of this work is to design and implement intelligent reactive agent using a method from the field of reinforcement learning, demonstrate it on visual demo Ms.Pacman and compare its intelligence with well-known informed methods of playing games (Minimax, AlfaBeta Pruning, Expectimax). The thesis is primarily structured into two parts. The theoretical part deals with adversarial search (in games), reactivity of agent and possibilities of reinforcement learning, all in the context of Ms. Pacman. The second part addresses the design of agent's versions behaviour implementation and finally its comparison to other methods of adversarial search problem, evaluation of results and a few ideas for future improvements.

Klíčová slova

umělá inteligence, hry s nulovým součtem, hry s neurčitostí, reaktivní agent, Expectimax, Markovské rozhodovací procesy, strojové učení, posilované učení, Q-Learning, gridworld

Keywords

artificial intelligence, zero-sum games, games with uncertainty, reactive agent, Expectimax, Markov decision processes, machine learning, reinforcement learning, Q-Learning, gridworld

Citace

BLOŽOŇOVÁ, Barbora. *Inteligentní reaktivní agent pro hru Ms. Pacman.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Drahanský Martin.

Inteligentní reaktivní agent pro hru Ms. Pacman

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením doc. Ing., Dipl.-Ing. Martina Drahanského, Ph.D. Dále prohlašuji, že jsem uvedla všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Barbora Bložoňová
30. května 2016

Poděkování

Ráda bych podělovala panu doc. Ing. Martinu Drahanskému, Ph.D. za jeho odborné vedení, nadhled a celkovou spolupráci. Dále bych ráda poděkovala své rodině za jejich nekonečnou podporu, motivaci, kávu a palačinky. V neposlední řadě bych chtěla poděkovat svým blízkým přátelům, kteří mě skutečně podrželi.

Obsah

1 Úvod **3**

2 Metody hraní her **5**
2.1 Základní pojmy . . . 5
2.2 Hry . . . 6

3 Učící agent **9**
3.1 Reaktivní agent . . . 9
3.2 Racionalita . . . 9
3.3 Markovský rozhodovací proces . . . 10
3.4 Strojové učení . . . 13

4 Analýza a návrh **18**
4.1 Analýza . . . 18
4.2 Návrh . . . 20

5 Realizace **27**
5.1 Zvolené prostředky . . . 27
5.2 Framework a reprezentace stavu hry . . . 28
5.3 Reflexivní agent a jeho ohodnocovací funkce . . . 28
5.4 Základní algorimy . . . 29
5.5 Framework a MDP problematika . . . 30
5.6 Value Iteration agent a gridworld problémy . . . 31
5.7 Q-Learning agenti . . . 33
5.8 Aproximační Q-Learning agent . . . 34

6 Experimenty a srovnání agentů **37**
6.1 Srovnání z vypočetní náročnosti . . . 37
6.2 Srovnání agentů na mapách `smallGrid` a `trappedClassic` . . . 38
6.3 Zatěžkávací zkouška na mapě `trickyClassic` . . . 41
6.4 Gridworld problémy a jejich parametry . . . 42

7 Závěr **44**

Literatura **46**

Přílohy **48**
Seznam příloh . . . 49

A Obsah CD **50**

B Manuál **51**
B.1 Ms. Pacman demo (`pacman.py`) 51
B.1.1 Příklady použití 51
B.2 Gridworld problémy (`gridworld.py`) 52
B.2.1 Příklady použití 53

C Přidatné obrázky k experimentům **54**

D Vlastní implementace **57**

Kapitola 1

Úvod

Schopnost inteligentního myšlení je jednou z nejdůležitějších vlastností lidí i zvířat. Vždyť bez inteligence by lidé nedokázali převýšit evolučně zdatnější obyvatele Země. Co to vlastně je? Inteligence se dá chápat jako schopnost (jedince nebo skupiny) klasifikovat a reagovat na okolní vjemy, aktivně s nimi interagovat a dokonce ho využít ve svůj prospěch. Filosofové si již od staletí páry kladou otázku, zda mohou mít stroje vlastní vědomí, stupeň inteligence, či preference. Samotné vědní odvětví umělé inteligence, dále *UI*, bylo postupně rozvíjeno během 20. století, kdy se souběžně s vývojem mj. matematické logiky, teorie algoritmů a především vývojem výpočetní techniky vůbec odkrývaly možnosti UI jako stále rostoucí vědní displíny.

Cílem této práce je vytvořit inteligentní entitu, agenta, který by dokázal pokořit tak složitý rozhodovací problém, jakým je *real-time* hra Ms. Pacman. Svět her je pro umělou inteligenci jedno z velkých témat. Hry byly jako zdroje střetu zájmů oponentů zkoumány již v první pol. 20. století. John von Neumann a Oskar Morgernstern na tyto konfliktní problémy nahlíželi jako na matematické příklady, které se dají analyzovat, sestavit jim matematický model a pomocí výpočtů se snažit nalézt optimální strategie pro každého svého účastníka. Hra by nemohla být zábavná, kdyby nepřítel nebyl dostatečně chytrý. Umělá inteligence silného oponenta ve složité hře představuje disciplínu stále otevřenou novým technikám, metodám a experimentům. Výpočetní náročnost takových her staví velkou překážku k vytvoření *ideální* umělé inteligence, vždyť teprve v roce 2016 dokázala umělá inteligece **porazit člověka** ve hře *Go* [1]! UI *AlphaGo* byla navrhnuta v rámci projektu *DeepMind* společnosti *Google* a povedlo se ji historicky poprvé drtivě porazit profesionálního hráče. Ačkoliv se již v minulosti podařilo překonat člověka i ve hře *Šachy*, stále je to však malý krůček pro tak velký svět zajímavých rozhodovacích problémů jakým je i Ms. Pacman. Ms. Pacman patří k těmto složitým úlohám nejen rozměrností stavů, jež ve hře můžou nastat, ale především náhodností chování svých oponentů, duchů. Nelze tedy přesně stanovit, jak se daný duch bude v danou situaci chovat, lze jen jaksi předpokládat (například za použitím pravděpodobnosti) duchovo chování, tudíž je těžší jednoznačne stanovit, jak má agent postupovat v takto dynamickém prostředí. Co kdyby se agent Ms. Pacman postupně na základě vjemů z prostředí **učil** své budoucí chování v podobné situaci? Jedna z kategorií strojového učení (*machine learning*) posilované učení (*reinforcement learning*) je mocný nástroj na aktivní vypočetní stanovení agentova rozhodnutí na základě aktuální odměny a trestu získaných v daném stavu. Agent Ms. Pacman se bude snažit co nejlépe reagovat na své aktuální okolí tak, aby jeho inteligence překonala problematiku náhodnosti a vypočetní náročnosti této hry. Kvalita agenta bude srovnána s již známými metodami řešení podobných problémů. Dále se práce zaměří na podobné problémy jako je Ms. Pacman a na nich otestuje kvalitu

inteligence agenta.

Práce je tedy rozdělena na dva velké celky: teoretickou a praktickou část. Teoretická část práce seznamuje čtenáře se základními pojmy a metodami hraní her v kapitole 2 a to tak, aby na ně následně mohla navázat nejdůležitější teoretická kapitola 3 zaměřující se na důležitou vlastnost agenta – schopnost učení. Tato kapitola nejdříve pojednává o racionalitě hráče v sekci 3.2 a popisuje pojem *Markovský rozhodovacích proces* v sekci 3.3. S těmito pojmy pak pracují již sekce jednotlivých algoritmů UI, například finální vylepšení Q-Learningu: *Aproximační Q-Learning* 3.4. Předěl mezi teoretickou a praktickou částí je kapitola 4, která prakticky rozebírá hru Ms. Pacman a již se detailně věnuje návrhům všech metod zvolených pro implementaci a otestování na demu Ms. Pacman. Vzhledem k primárnímu zaměření této práce na UI byl využit framework Ms. Pacman, se kterým seznamuje praktická část práce implementační kapitola 5. Tato kapitola tedy řeší objekový návrh aplikace a především prezentuje implementovaná řešení jednotlivých metod. Kapitola se zaměřuje i na zmíněné další *gridworld* problémy zkoumatelné z hlediska UI, pro které lze také názorně použít zvolené algoritmy. Implementační kapitolu uzavírá popis implementace agenta pro metodu *Aproximační Q-Learning*. Poslední částí práce je kapitola experimentální charakteru 6, která se zaměřuje na porovnání jednotlivých agentů na různých typech problémů (vč. Ms. Pacman) a obsahuje finální srovnání vypočetní náročnosti algoritmů.

Kapitola 2

Metody hraní her

Tato kapitola si klade za cíl seznámit čtenáře s metodami hraní her. Hry úzce souvisí s metodami řešení úloh, zahrnují tedy snahu dostat se z počátečního stavu do cílového, např. za použití posloupnosti nějakých pravidel.

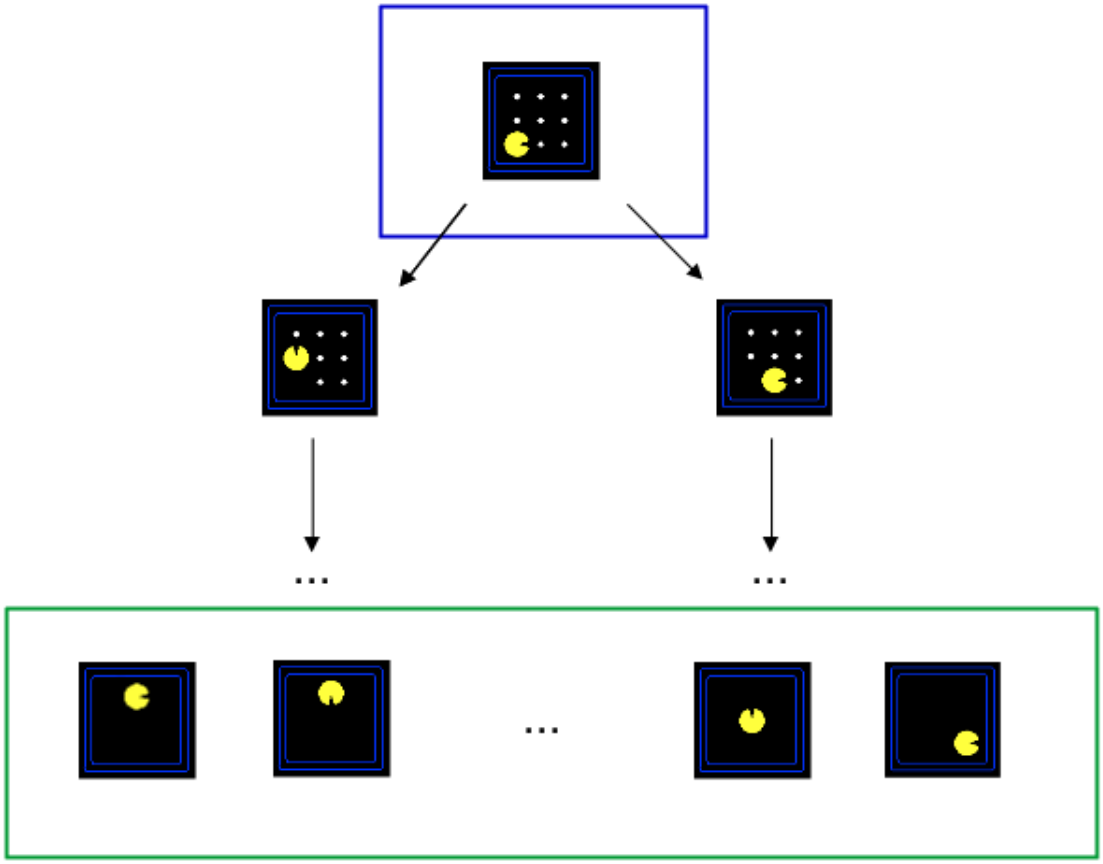
2.1 Základní pojmy

Stavový prostor

Stavový prostor [8] je množina všech stavů, které mohou ve hře nastat. Stavový prostor lze reprezentovat jako orientovaný graf, jehož *uzly* představují jednotlivé pozice ve hře (stavy) a orientované hrany přechody mezi jednotlivými stavy, tedy přípustné tahy ve hře. První uzel se nazývá *kořen*, koncové uzly, *listy*, pak reprezentují koncové pozice ve hře (cíle, stavy). Na obrázku 2.1 lze vidět ukázku omezeného (zjednodušeného) stavového prostoru hry Pacman. Velikost takto velmi omezeného stavového prostoru musí vzít v potaz: rozměry 2D hracího pole, 4 možné směry pohybu Pacmana a fakt, zda 8 kuliček jídla už (ne)bylo sněženo, tedy: $3 \times 3 \times 2^8 \times 4 = 9\,216$ (!) stavů a to nebyl brán v potaz ani jediný nepřítel, natožpak reálná velikost hracího pole desky.
K dosažení každého uzlu je potřeba vydat nějaké úsilí, **cenu** [8], která tedy udává nezáporné ohodnocení hrany z jednoho uzlu k druhému. **Hloubka uzlu** [8] udává počet hran na cestě od počátečního uzlu k danému uzlu. Kořen má hloubku 0, jeho následníci mají hloubku 1 atp.

Optimální versus nejlepší řešení

Další důležitou součástí metod řešení úloh je **hodnotící funkce** (*successor function*) [8]. Jedná se o funkci vyhodnocující celkovou cenu od jednoho uzlu k druhému, např. součet ceny a *heuristické funkce*. **Heuristická funkce** [6] je založena na empirických znalostech (až odhadech) o hře, používaná při prohledávání stavového prostoru. Může to být například vzdálenost Ms. Pacman vůči svému cíli. Podle jejího využití dělíme algoritmy na **informované** a **neinformované** [8]. Proč se ale používá heuristika místo jistějšího systematického prohledávání stavového prostoru, které by mělo definovat vlastně *nejlepší* řešení? Je to především kvůli velké vypočetní náročnosti toho způsobu prohledávání a vyhodnocování celého stavového prostoru, který u složitějších problémů nabývá příliš velkých rozměrů, aby se vše stihlo provést, např. v reálném čase při hraní hry. Pokud je heuristika *optimální*, zaručuje zvýšení efektivity a rychlosti.



Obrázek 2.1: Ukázka stavového prostoru pro hru Pacman. Každý pohyb Pacmana znamená nový stav hry, tedy uzel. Modrý obdélník ohraničuje počáteční stav, kořen, a zelený obdelník ohraničuje možné koncové stavy, listy. Převzato a upraveno z volně dostupných prezentací kurzu [6].

Vlastnosti algoritmů

Rozlišujeme několik následujících vlastností algoritmů[8]:

- **Úplnost** – Pokud existuje řešení, je garantováno, že ho algoritmus vrátí?
- **Optimálnost** – Je garantováno nalezení nejlepší řešení, tedy řešení s nejmenší cenou?
- **Časová složitost** – Kolik uzlů je nutno expandovat? Kolik to zabere času v nejhorším/nejlepším případě/průměrně?
- **Prostorová složitost** – Kolik dat je nutno si uchovávat v paměti v nejhorším/nejlepším případě/průměrně?

2.2 Hry

Hra (také se označuje jako *adversarial search*) je z matematického hlediska rozhodovací problém, ve kterém figurují 2 a více účastníků, hráčů [8]. Hráči se snaží chovat racionálně,

viz sekce 3.2. **Složité hry** [8] jsou hry s tak velkým úplným stavovým prostorem, že by jeho neinformované prohledávání nebylo možné výpočetně zvládnout. Například Šachy, Go, Ms. Pacman. V takových případech je potřeba omezit strom stavového prostoru pomocí hloubky, tedy maximálního počtu kroků dopředu oproti aktuálnímu stavu, a použít *statickou hodnotící funkci*, která určuje *pravděpodobnost* dosažení cíle z aktuálního stavu. Následující metody hraní her potřebují mít úplnou informaci o stavovém prostoru hry.

Minimax

Metoda [8] prohledávání do hloubky s omezením hloubky prohledávání. Hodnotící funkce nám udává určitou *hodnotu* listů $V(s)$, tedy nejlepší výsledek/užitek, kterého lze dosáhnout z aktuálního stavu. Následně se úrovni hráče při procházení stavového prostoru vybírá **maximum hodnoty:** $V(s) = \max\ V(s')$, kde $s' \in naslednici(s)$.
Na úrovni protihráče **minimum hodnoty:** $V(s') = \min\ V(s)$, kde $s \in naslednici(s')$. Hráči se takto rekurzivně střídají od listů až po finální výpočtení hodnoty kořene.

AlfaBeta řezy

Metoda pro zmenšení stavového prostoru [8], odříznutím nadbytečných větví. Využívá dvou *mezí*:

- $\boldsymbol{\alpha}$ reprezentující dolní mez ohodnocení uzlu, tedy tah hráče (zpočátku $\boldsymbol{\alpha = -\infty}$),
- $\boldsymbol{\beta}$ reprezentující horní mez ohodnocení uzlu, tedy tah protihráče (zpočátku $\boldsymbol{\beta = \infty}$).

Algoritmus si pamatuje hodnoty svých mezí, a postupně je aktualizuje porovnáváním meze s následníky uzlu aktuální úrovně a to podle toho, na které úrovni se nachází: (*max* – aktualizuji α na největší hodnotu následníků, *min* – aktualizuji β na nejmenší hodnotu následníků). Porovnávání probíhá dokud $\alpha < \beta$, takto se vynechají nadbytečné větve, jejichž procházení již nezmění výslednou cestu. Konečná hodnota kořene je stejná jako u Minimaxu, a pokud by se uzly správně seřadily, výrazně se tím změní vypočetní náročnost algoritmu.

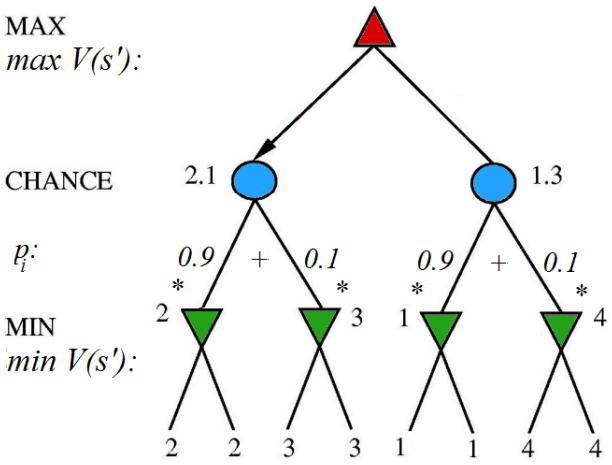
Expectimax

Expectimax [13] je jednou z možností jak řešit **hry s neurčitostí** jako je i Ms. Pacman. Hry s neurčitostí také zahrnují střídání hráčů a mají také úplnou informaci o stavu hry, avšak navíc využívají při získání těchto informací *pravděpodobnosti*, které popisují náhodný charakter stavů ve hře. Příkladem jsou hry, kde figuruje házení kostkou nebo např. řízení libovolného robota v reálném světě, kdy se může náhodně pokazit nějaká jeho součástka. Expectimax obohacuje každý tah hráče (jako Minimax) o náhodnost. **Hodnoty stavů** nyní navíc reflektují průměrnou pravděpodobnost stavů:

$$V_{\min}(s) = \sum_{i=0} p_i * \min\ V(s') \tag{2.1}$$

$$V_{\max}(s') = \sum_{i=0} p_i * \max\ V(s') \tag{2.2}$$

kde $s' \in naslednici(s)$ a p_i je pravděpodobnost daného stavu i-té úrovně.
Tyto rovnice udávají tzv. **očekávaný užitek** (*expected utility*) dané úrovně [13]. Na obrázku 2.2 lze vidět použití algoritmu.



Obrázek 2.2: Na obrázku lze vidět ukázku vyhodnocování uzlů a výslednou levou cestu, kterou si algoritmus posléze zvolí. Převzato a upraveno z webu [1].

[1] http://agents.csie.ntu.edu.tw/~yjhsu/courses/u0220/1997/10-29/img28.gif

Kapitola 3

Učící agent

Tato kapitola se zaměřuje na teoretický podklad hlavního cíle práce, inteligentní formu reaktivního agenta [9] a jeho prostředí.

3.1 Reaktivní agent

Reaktivní agent [9] se dá představit jako autonomní bot, který reaguje na své aktuální okolní prostředí. Nebere tedy v potaz následky svého konání. Matematicky lze zapsat *šesticí* [9] jako: $(S, T, A, vjem, ukon, akce)$, kde:

- S je množina okolních stavů agenta,
- $T, T \subseteq S$ je okolí, které může agent vnímat,
- $vjem : S \to T$ je agentem právě vnímaný stav okolí ($s \in S$),
- A je množina možných úkonů, jež může agent v daný stav vykonat,
- $ukon : A \times A \to S$ je vykonaný úkon, jež má za následek změnu prostředí S,
- $akce : T \to A$ je vybraná akce na základě $vjemu$.

Agent představuje účastníka hry (hráče).

3.2 Racionalita

Každý hráč má nějaký cíl a k němu vedoucí **strategii**. Strategie se skládá z hráčových akcí a voleb, které pro hráče vedou k vlastnímu užitku (*utility*), potažmo výhře. Hráč se potýká s různými stavy ve hře a snaží se z nich vytěžit co nejvíce na základě svých preferencí [6]. Pokud tyto hráčovy preference dodržují axiomy dle [13], zaručují tak hráčův cíl, maximalizaci očekávané hodnoty *užitkové funkce*:

$$U([p_1, S_1; \ldots; p_n, S_n]) = \sum_i p_i U(S_i) p \tag{3.1}$$

Pro umělou inteligenci hráče je též důležité explicitně stanovit určitá pravidla, ze kterých má hráč vybírat užitek. Předpoklad racionality je hlavní rozdíl teorie her a teorie rozhodování.

3.3 Markovský rozhodovací proces

Agent se bude pohybovat ve *stochastickém* (náhodném) prostředí, je tedy vhodné si definovat Markovský rozhodovací proces [2], dále **MDP** (*Markov decision process*), který řeší takovéto nedeterministické problémy prohledávání, jedná se o čtveřici: (S, A, T, R), kde:

- $s, s \in S$ je množina stavů, která zahrnuje počáteční a koncový stav,
- $a, a \in A$ je množina akcí,
- $T(s, a, s')$ je **přechodová funkce** (*transition function*), nebo též *model*, tedy pravděpodobnost přechodu ze stavu s do s' ($P(s'|s, a)$),
- $R(s, a, s')$ je **odměnová funkce** (*reward function*), tedy funkce vracející odměnu přechodu ze stavu s do s'; s ní souvisí exponenciální snižování hodnot odměn koeficientem γ (*discount*).

Důležitý rozdíl proti předchozím metodám hraní her je fakt, že výsledná hodnota aktuálního stavu závisí pouze na aktuálním stavu a jeho akci, ne na jeho historii [4]. Další významný rozdíl je ten, že předchozí metody se snažily o nalezení optimálního plánu nebo sekvence akcí z počátečního stavu až do cíle. V MDP modelu přechody stavů nezávisí na hodnotách minulých stavů, ani na agentových minulých akcích, MDP hledá **optimální strategii** (*policy*) pouze pro stavy budoucí (následníky) [12]:

$$\pi^* : S \to A \tag{3.2}$$

kde π mapuje akci na každý stav a pokud se akce provede (dáno náhodností), maximalizuje očekávaný užitek (akumulací odměn).

Následující algoritmy této kapitoly řeší jak nalézt optimální strategii na základě daného modelu. Tyto algoritmy, nebo též *dynamické programovací techniky* [5] pak tvoří základy a inspiraci pro algoritmy strojového učení pro MDP prostředí. Souhrnně se všechny tyto algoritmy dají zařadit do kategorie strojového učení tzv. posilovaného učení, nebo též zpětnovazebního učení (*reinforcement learning*). Algoritmy staví na zpětné vazbě, **zkušenosti**, a raději než aby stavily na těžce dosažitelné ohodnocovací funkci, stanovují hodnoty stavů a jejich akcí. Algoritmy také spoléhají na fakt, že pro modely s exponenciálně snižovanými odměnami (pomocí γ) a možným až nekonečným počtem stavů lze najít optimální deterministickou strategii [2]. Je potřeba ještě ujasnit pojmy **užitek** a **hodnota**. Užitek [12] je definován jako suma koeficientem γ snížených odměn tak, aby MDP měl větší šanci skončit (pro agenta je lepší vzít blízkou odměnu co nejdříve, tudíž lépe sbírá odměny). Hodnota [12] definuje **optimální očekávaný užitek** (akumulovaný průměr očekávaných výsledků) ze stavu (pro maximalizační a minimalizační uzly).

Value iteration a Policy iteration

Příkladem algoritmu hledající optimální strategii pro MDP prostředí je iterativní algoritmus **Value iteration** [2]. Algoritmus staví na přepočtu V-hodnot dle následujících bodů:

1. Začni od úrovně s hodnotou $V_0 = 0$ pro všechny stavy.

2. Plň vektor optimálních hodnot všech stavů $V_k(s)$ novými hodnotami vykonáním Expectimaxu pro aktuální úroveň pomocí rovnice:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right] \tag{3.3}$$

   kde $\gamma V_k(s')$ je hodnota budoucí odměny.

3. Opakuj předchozí krok dokud vektor V_k nekonverguje (konvergenci především zapříčiňuje γ, což zaručuje optimálnost metody). Podmínka konvergence je dána rovnící:

$$|V_{k+1}(s) - V_k(s)| < presnost \tag{3.4}$$

Velký rozdíl oproti Expectimaxu je ten, že se nemusí provádět neustálá *rekurze* výpočtu očekávaného užitku, protože je už vypočten ve vektoru hodnot $V_k(s')$ (jak již bylo řečeno,algoritmus je **iterativní**: staví na každé předchozí vrstvě). Stále se však metoda potýká s vysokou prostorovou složitostí. Alternativou jsou proto algoritmy **Policy iteration** [2], které se snaží o přímé nalezení optimální strategie raději než zdlouhavé přepočty nových hodnot. O těchto metodách se v souvislosti se strojovým učením zmiňuje další kapitola 3.4.

Q-hodnota

Posledním důležitým pojmem je Q-hodnota [14], která definuje optimální očekávaný užitek v budoucnosti z uzlu náhodnosti, tzv. *Q-stavu*, příklad na obrázku 3.2. Budoucností je myšlen následník stavu po provedení přechodu (s, a, s').

Value iteration potřebuje získat optimální V-hodnoty a Q-hodnoty, pro které v MDP lze tedy uplatnit (alternativně k rovnici 3.3) následující rovnice vycházející přímo z Bellmanových rovnic [13]:[1]

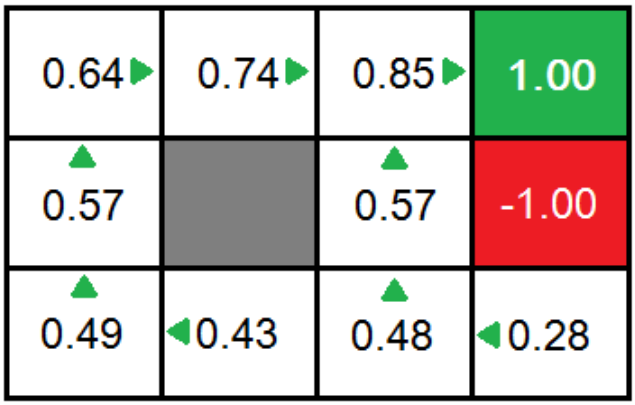
$$Q^*(s) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right] \tag{3.5}$$

$$V^*(s) = \max_a Q^*(s)(s,a,s') \tag{3.6}$$

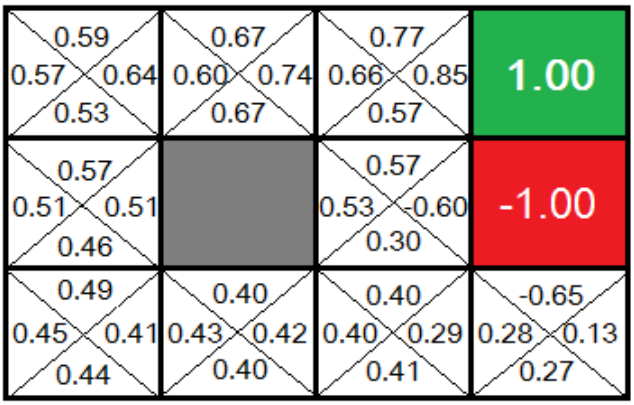
Na obrázku 3.1 lze vidět optimální strategie (reprezentovány zelenými šipkami) z vypočtených hodnot pro každý stav pro daný *gridworld* problém.

MDP lze též reprezentovat prohledávacím stromem, viz obrázek 3.3, který se velmi podobá Expectimaxu avšak jak již bylo řečeno není potřeba rekurze, pouze se iterativně vyhodnocuje nová vrstva V_{k+1}.

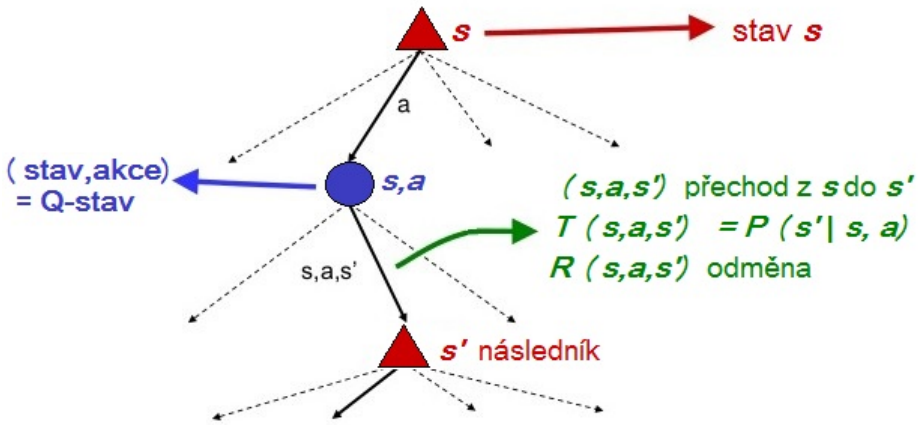
[1] hvězdička * značí optimální hodnotu



Obrázek 3.1: Problém představuje pole o velikosti 12 polí (z nichž šedé je nedosažitelné), 4 směry možnosti chůze a dva koncové stavy s odměnami ohodnocenými -1 a 1. Problém je stochastický, přechod ze stavu do následníka se provede dle své pravděpodobnosti v přechodové funkci. Již po 5 iteracích je vidět optimální strategie, reprezentovaná šipkou směru (agent začíná v levém spodním rohu), vypočtena pomocí hodnoty V_5 každého pole. Hodnoty pro vlastní obrázek jsou převzaty z volně dostupných prezentací kurzu [6].



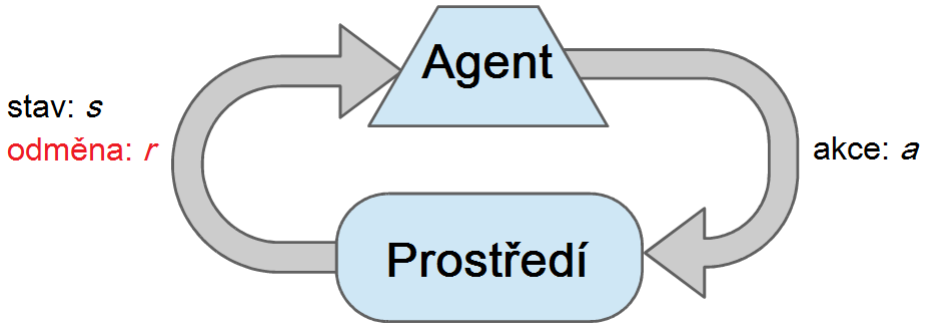
Obrázek 3.2: Ukázka Q-hodnot pro předešlý grid-world problém (po 5. iteraci). Q-hodnoty se počítají pro každou možnou akci ze stavu do následníka, proto jsou pro jedno políčko na hracím plánu 4 pro každý směr, kam může agent jít. Hodnoty pro vlastní obrázek jsou převzaty z volně dostupných prezentací kurzu [6].



Obrázek 3.3: Na obrázku lze vidět analogii MDP Value Iteration k Expectimaxu [14]. Obrázek převzat z volně dostupných prezentací kurzu [6].

3.4 Strojové učení

Další důležitou vlastností agenta bude schopnost *učit se* [14]. Agent se učí na pozorovaných výsledcích svých akcí, tzv. **vzorky** (s, a, s', r), kde r je odměna nového stavu. Stále se jedná o MDP, avšak největší rozdíl zde nastává v tom, že **není známá** odměnová funkci R, ani přechodová funkci T. Agent (viz obrázek 3.4) se tedy musí metodou *pokus-omyl* **naučit** tyto hodnoty modelu.



Obrázek 3.4: Agent využívající strojové učení. Obrázek přeložen a převzat z volně dostupných prezentací kurzu [6].

Metody strojového učení se dělí dle **závislosti na modelu** [14] [7]:

- (*model-based*) metoda chce empiricky vytvořit model, který již lze řešit pomocí MDP – to se děje na základě průběžného výpočtu očekávaného užitku:
  1. Spočítej všechny výsledky/vzorky s' pro každý pár (s, a).
  2. Spočítej odhad okamžitého užitku, tedy odhad přechodové funkce $\hat{T}(s, a, s')$.
  3. Odhadni odměnovou funkci $\hat{R}(s, a, s')$ na základě objevených odměn stavů.

- (*model-free*) metoda model nepotřebuje, stačí posbírat vzorky, protože *rozdělení pravděpodobnosti* pro daný vzorek určuje jeho výskyt (není potřeba počítat očekávaný užitek). Dále se budeme zabývat těmito metodami.

Dalším dělicím kritériem je **řízení akcí**:

- *pasivní metody* – je daná statická strategie $\pi(s)$, kterou se agent pevně řídí a získává z ní vzorky, ze kterých se učí hodnoty stavů $V(s')$ pro daný stav s.
  Ukázka postupu metody přímého vyhodnocení (**direct evaluation** [6]):

  1. Následuj π.
  2. Pro každý navštívený stav vypočti sumu koeficientem γ budoucích snížených hodnot.
  3. Spočítej průměr sumy.

  Není potřeba T ani R, avšak je zde přílišná abstrakce (plýtvání informací) a vyhodnocení stavů díky tomu má příliš velkou časovou náročnost.
  Další možností je **Time-Difference value learning** [14], dále TD učení, které se místo vyhodnocování hodnoty Value iteration (a nakonec teprve získání výsledné strategie) snaží vyhodnotit přímo nové strategie na základě hodnot (**Policy iteration**). Vyhodnocení probíhá *po každé akci*, protože nelze zaručit, že se strategie bude vracet znovu do již vyhodnoceného stavu, aby opět přehodnotila svou hodnotu [5].
  Ukázka postupu **TD učení**:

  1. Následuj π.
  2. Aktualizuj $V(s)$ pokaždé, když narazíš na přechod pro vzorek (s, a, s', r). **Aktualizace** (*update*), spočívá v tom, že se vezme aktuální hodnota stavu a přičte se k ní o koeficient α zmenšený rozdíl mezi očekávaným stavem a reálným vzorkem. α reguluje fakt, že nové odměny budou mít větší váhu než staré.

$$\text{Vzorek } V(s) \quad : \quad vzorek = R(s, \pi(s), s') + \gamma V^{\pi}(s') \tag{3.7}$$

$$\text{Update } V(s) \quad : \quad V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(vzorek - V^{\pi}(s)) \tag{3.8}$$

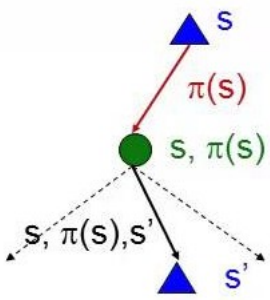
  TD učení je pasivní metoda, která je schopna získat ohodnocení hodnot V, avšak není schopna aktivně přeměnit hodnoty na novou strategii $\pi(s)$ dle rovnic MDP pro Policy iteration (viz obrázek 3.5 [6]):

$$\pi(s) = arg\max_{a} Q(s, a) \tag{3.9}$$

$$Q(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V(s')\right] \tag{3.10}$$

  Problém je opět chybějící přechodová a odměnová funkce (T, R), to řeší až *aktivní metody* strojového učení.

- *aktivní metody* – je dána statická strategie $\pi(s)$, ale agent provádí vlastní akce a získává z nich vzorky, ze kterých se učí optimální strategie a hodnoty stavů $V(s')$. Mezi aktivní metody patří **Q-Learning**.



Obrázek 3.5: Policy iteration metoda s potřebnými proměnnými (pomocí strategie π(s) se dostávám ze stavu s do jeho následníka (s')). Obrázek převzat z volně dostupných prezentací kurzu [6].

Q-Learning

Dosud se primárně vycházelo z V-hodnot, avšak Q-Learning je založen na vyhodnocování posloupností akcí do nových stavů, tedy na využití Q-hodnot. Jediné, co je tedy potřeba vědět je aktuální stav a jeho možné akce [14]:

$$vzorek = \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right] \tag{3.11}$$

Následně se provede aktualizace (update) TD učení, kde α nyní představuje rychlost učení (*learning rate*):

$$Q(s,a) \leftarrow Q(s,a) + \alpha(vzorek - Q(s,a)) \tag{3.12}$$

Q-Learning postup

1. Navštiv nový uzel s'.
2. Ze vzorku (s,a,s',r) vypočti novou hodnotu jeho odhadu $Q_{k+1}(s,a)$ jako:
   průměr očekávaných užitků akcí stavu $T*$ (okamžitá odměna + nejlepší možná Q-hodnota následníka $Q_k(s,a)$), tedy:

   $$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right] \tag{3.13}$$

   Průměr T a hodnoty odměn R se postupně počítají na základě akcí (nejsou zpočátku známy), algoritmus se tedy blíží následující rovnici:

   $$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a') \tag{3.14}$$

   kde r je přímá odměna ze stavu [10].
3. proveď update TD učení (průměr odhadu vůči vzorku)

   $$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a')\right] \tag{3.15}$$

Q-Learning konverguje k optimální strategii, i když následuje posloupnost neoptimálních akcí [16]. Tento jev se nazývá *off-policy learning* [14]. Jak ale agent vybírá, kterou akci provede, aby maximalizoval svůj užitek z tréninku? Je nutné zvolit **dobrý poměr** mezi zjišťováním terénu (*exporation*) a využíváním již zjištěných aktuálních dobrých strategií (*exploitation*) [5]. Dobrý poměr lze řešit několika způsoby [6]:

- Zavedením koeficientu $\varepsilon - greedy$ pro výběr náhodných akcí (vůči optimálním). Výhodné je ovšem koeficient časem zmenšovat, aby nedocházelo k nelogičnostem akcí.
- Využitím **explorační funkce**, která bere v potaz nejen budoucí užitek stavu U, ale i množství navštívění stavu N. Díky tomu se prozkoumají stavy, jejichž dosud zjištěná špatná hodnota ještě není úplně stabilní (málo výskytů). Funkce tedy snižuje důležitost opakujících se stavů a dává větší důraz na neprozkoumané stavy. Nakonec s tímto skončit, výsledná aktualizace hodnot (update) potom vypadá:

$$Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a')) \tag{3.16}$$

  Update se následně propaguje dál.

Novým pojmem pro metody posilovaného učení je **lítost** (*regret*) jako míra toho, kolik agenta stály všechny chyby během tréninku (rozdíl mezi očekávanými odměnami a optimálními odměnami). Je snaha tuto co nejvíce míru minimalizovat (optimálně se i učit optimální řešení daného problému).

Approximační Q-Learning

Zatímco základní Q-Learning pomohl zbavit se zbytečného přepočítávání hodnot, stále je potřeba si neustále udržovat tabulku všech Q-hodnot pro každou kombinaci stavu a akce. Pro složitou úlohu jakou je Ms. Pacman je tato velká prostorová složitost stavového prostoru stále nevhodná. Z toho důvodu je potřeba **generalizovat**: naučit se Q-hodnoty malého množství tréninkových stavů a hodnoty generalizovat na podobných situacích. To je důležitá myšlenka metody aproximační Q-Learning a potažmo i strojového učení [2]:
popsání stavu pomocí vektoru vlastností (*properties*), vlastosti jsou funkce ze stavů do reálných čísel (např. 0, 1), které zachycují důležité vlastnosti stavu. Následně lze tyto vlastnosti ohonodnotit **váhou** (*weight*) a sesavit lineární rovnice [2]:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) \tag{3.17}$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \cdots + w_n f_n(s,a) \tag{3.18}$$

Nevýhoda takové generalizace je ale fakt, že je potřeba stanovit co nejlepší vektor vlastností tak, aby se hodnoty rozdílných stavů nepodobaly.

Aproximační Q-Learning postup

1. Navštiv nový uzel a ze vzorku (s, a, s', r) vypočti rozdíl odhadu Q-hodnoty vůči vzorku:

$$rozdil = \left[r + \gamma \max_{a'} Q(s',a') \right] - Q(s,a) \tag{3.19}$$

2. Udělej aktualizaci (update) TD učení (průměr odhadu vůči vzorku)

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') \right] \tag{3.20}$$

3. Následně místo updatu jedné Q-hodnoty (základní Q-Learning), aproximuj Q-hodnoty pomocí změny váhy w_i vůči vektoru vlastností:

$$w_i(s) \leftarrow w_i + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] f_i(s,a) \tag{3.21}$$

kde:

- $[r + \gamma \max_{a'} Q(s',a')]$ reprezentuje cíl (maximální), kterého se snaží daný stav dosáhnout (nejlepší odhad Q-hodnoty následujícího stavu).
- $[Q(s,a)]$ vzorek aktuálního stavu.
- $f_i(s,a)$ lineární funkce, jejíž rozdíl vzorků je aproximován[2], což napomáhá možnosti využití i nelineárních funkcí ve vektoru vlastností. Funkce reflektuje předpoklad odhadu Q-hodnoty následujícího stavu.
  Zjednodušeně řečeno např. u Ms. Pacman: Pokud Ms. Pacman narazí do ducha, zvětší se váha vlastnosti situace ohrožení Ms. Pacman duchem. Pokud se tedy stane nějaká pozitivní akce, pozitivní váha zvětší hodnotu vlastnosti ve vektoru vlastností a naopak.

[2] K aproximaci se používá obecná **Metoda nejmenších čtverců, tzv. lineární regrese**

Kapitola 4

Analýza a návrh

Tato kapitola se podílí na praktické části práce a to především praktické analýze problematiky Ms. Pacman (první část 4.1) a návrhu **chování agentů** (druhá část 4.2).

4.1 Analýza

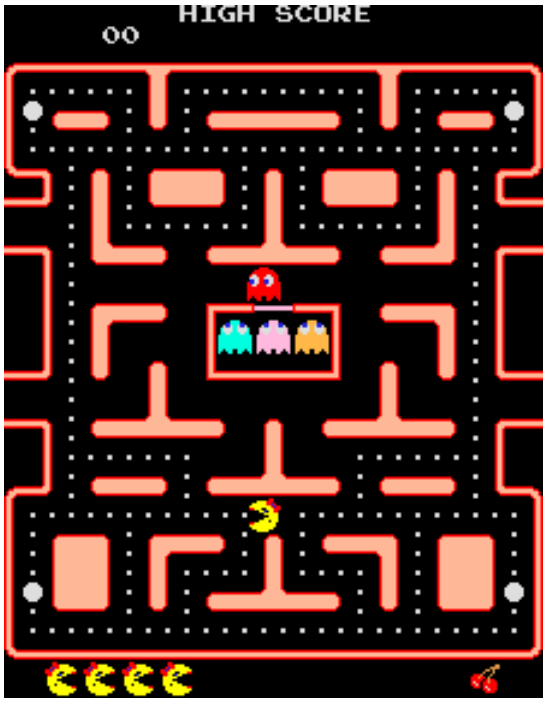
O hře a důvod jejího výběru

Ms. Pacman je arkádová hra z roku 1982, která vylepšuje původní koncept hry Pacman. Hra je obohacena o nový design hlavní postavy, mapy a především implementuje lepší umělou inteligenci nepřátel. Hlavní postavou ovládanou hráčem je nyní Ms. Pacman, nepřátelé zůstávají v podobě 4 barevných duchů. Cílem hry je dosáhnout co největšího skóre pojídáním kuliček, duchů (časově omezené; po snězení speciální větší kuličky tzv. power-upu), nebo ovoce. Zvýšení obtížnosti oproti Pacmanovi spočívá především v **opuštění deterministického chování duchů**, čímž pádem nelze předpovídat jejich chování. Tento fakt spolu se složitostí hry jsou hlavními faktory proč se na hru zaměřit z hlediska strojového učení. Ukázka původní verze hry je vidět na obrázku 4.1.

Pravidla

Pro účely této práce se bere v potaz zjednodušení původní hry na vizuální demo tak, aby byla dobře vidět účinnost jednotlivých algoritmů a nebyly příliš vysoké nároky na výpočetní výkon. Demo není rozděleno na levely, odehrává se na zvolené mapě a je založeno na umělé inteligenci obou stran (Ms. Pacman vs 2 duchové). Cíl Ms. Pacman je maximalizovat skóre na aktuální mapě, zatímco cíl nepřátel je její skóre minimalizovat. Stochastické chování nepřátel je simulováno tak, aby co nejvíce odpovídalo svému vzoru (zdrojové kódy hry nejsou veřejné, tudíž nelze simulovat chování duchů naprosto stejně jako v originálu: chování duchů je tedy **náhodné**, avšak duchové dodržují např. následující pravidlo: nemohou jít zpět, pouze se otočit na křižovatce např. o 90 stupňů.). Duchové se snaží chytit a zabít Ms. Pacman. Pokud se jim to povede, hra končí (skóre - 500 bodů). Skóre Ms. Pacman se navyšuje jezením jídla (skóre + 10bodů za jednu kuličku), power-upů a následným jezením duchů (skóre + 200 bodů za každého ducha) po omezený čas trvání power-upu. Bonusová ovoce se neberou v potaz. Skóre se snižuje po dobu chození Ms. Pacman po bludišti bez jezení ovoce/duchů (každé takové kolo skóre - 1 bodů). Mapy se též liší, jsou menší a obsahují pro umělou inteligenci Ms. Pacman problémová zákoutí, jako větší neohraničené plochy, stísněný prostor atp. Ms. Pacman a duchové mají stejnou rychlost. Tato zjednodušení slouží

především ke snížení vypočetní náročnosti algorimů na běžně dostupný hardware.



Obrázek 4.1: Screenshot 1. levelu originální verze Ms. Pacman. Převzato z webu [1].

Klasifikace dema Ms. Pacman z hlediska Teorie her

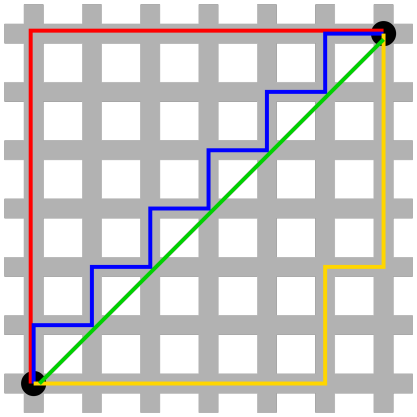
Ms. Pacman je hra, tzn. strategická interakce mezi 2 a více hráči, kteří chtějí dosáhnout optimálního výsledku ve hře. Vizuální demo Ms. Pacman lze specifikovat podle několika kritérií:

- **dle informovanosti hráče o hře:** *hra s neúplnou informací* (*game with uncertainty*) – duchové mají stochastické chování,
- **dle počtu tahů:** *hra strategická* – hráči provádějí souběžná rozhodnutí,
- **dle míry konkurence/typu výhry:** *hra s nulovým součtem* (*zero-sum game*) – jeden hráč maximalizuje svou výhru a druhý minimalizuje ztrátu (míra výhry jednoho značí míru prohry druhého hráče),
- **dle počtu hráčů:** Ms. Pacman je maximizér a 2 duchové jsou minimizéři,
- **dle komunikace hráčů:** *nekooperativní hra* – Ms. Pacman a duchové spolu nekomunikují a souběžně hrají proti sobě (striktně kompetitivní hra),
- **dle zdrojů:** *antagonistický konflikt* – Ms. Pacman a duchové sdílí jedno pevné maximální skóre dle herního plánu: *gridworld game* – vše probíhá diskrétně, po polích na mřížce v herním bludišti, jež mohou nabývat více stavů: prázdná cesta, zeď, prázdná cesta s hráčem, cesta s kuličkou, cesta s power-upem.

[1] https://upload.wikimedia.org/wikipedia/en/6/6c/Mspacman.png

4.2 Návrh

Po nastudování teorie a analýze chování nepřátel ve hře (duchů) je důležité jasně poupravit cíle práce: budou se porovnávat nejen metody Minimax a Alfa-Beta Řezy s učícím agentem, ale především také Expectimax, který bere v potaz stochastičnost akcí duchů. Tyto algoritmy se budou dále v textu označovat jako *základní algoritmy*.
Pro výpočet vzdáleností bude použita tzv. *manhattonovská metrika* [11] jako standard pro gridworld problematiku, kde figuruje mřížka a 4 směry pohybu agenta (Lze vidět na obrázku 4.2). Tato vzdálenost je měřena jako suma rozdílu absolutních vzdáleností koordinát dvou bodů $|x1 - x2| + |y1 - y2|$.

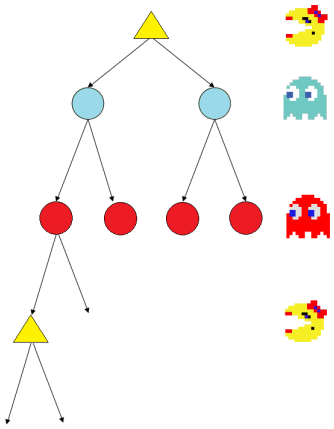


Obrázek 4.2: Červená, modrá i žlutá čára reprezentují stejnou manhattonskou vzdálenost o velikosti 12 (narozdíl od Euklidovské vzdálenosti reprezentované zelenou čárou). Převzato z webu [2].

Minimax, AlfaBeta řezy

Pro základní algoritmy bylo nutné vzít v potaz větší počet minimalizujících nepřátel, kteří se v každém tahu dané hloubky střídají s maximalizující Ms. Pacman, viz obrázek 4.3. AlfaBeta řezy i Minimax budou využívat vyhodnocovací funkci založenou na skóre následníka stavu ve hře. Algoritmy vyhodnocují stavy až do stanové hloubky/terminální uzlu a následně vrátí nejlepší možnou akci pro svého maximalizačního agenta Ms. Pacman.

[2] https://upload.wikimedia.org/wikipedia/commons/0/08/Manhattan_distance.svg



Obrázek 4.3: Ukázka střídání hráčů na stromu algoritmu Minimax. Obrázek převzat a upraven na základě volně dostupných prezentací kurzu [6].

Expectimax a lepší ohodnocovací funkce

Jelikož Expectimax průměruje hodnoty následníků ve svých uzlech náhodnosti *chance node*, bere tedy v potaz hodnoty všech stavů (ne jen těch nejlepších a nejhorších, jak tomu bylo doposud). Proto bude pro tohoto agenta bude naimplementována kvalitnější vyhodnocovací funkce, která by měla společně s algoritmem dosáhnout lepšího skóre než u předchozích agentů vzhledem ke stochastičnosti pohybu nepřátel. Funkce bere v potaz nejen herní skóre následníka, ale také další součásti stavu hry (vzdálenosti jídla, duchů, power-upů atp. Tato funkce by se dala do budoucna vylepšit například o algoritmy prohledávání stavového prostoru jako *BFS*, *DFS*[3], atp. tak, aby se dále brala v potaz například pozice stěn na herní desce. Další možností je do funkce přidat vliv vzdálenosti nejbližší křižovatky [15], pokud je agent příliš blízko nepřátel. Toto řešení však nebude dobře fungovat, pokud budou křižovatky příliš vdálené od agentovy pozice.
Samotný algoritmus Expectimax je (stejně jako Alpha Beta řezy) založený na algoritmu Minimax a je také z hlediska porovnání s pokročilými metodami umělé inteligence nejzajímavější, proto lze jako názornou ukázku uvést právě návrh jeho pseudokód 1. Algoritmus se dále bude muset provázat s **akcí**, jež má každý agent provést na základě vyhodnocené hodnoty svého uzlu.

[3] *BFS* = slépé prohledávání do šířky, *DFS* = slepé prohledávání do hloubky; tyto a další metody popisuje [8]

Algorithm 1 Expectimax – pseudokód vyhodnocování hodnot stavů, 1. část

Vstup: stav hry
Výstup: užitek stavu

```
1: function EXPECTIMAX(stav)
2:     if terminální uzel then
3:         return VYHODNOCOVACIFUNKCE(stav)                    // navrať užitek stavu
4:     end if
5:     if další agent == MAX then
6:         return MAXNODE(stav)                  // vrstva Ms. Pacman = maximalizační
7:     end if
8:     if další agent == EXP then
9:         return CHANCENODE(stav)                    // vstva nepřátel = odhad tahu
10:    end if
11: end function
```

Vstup: stav hry
Výstup: hodnota uzlu pro Ms. Pacman

```
12: function MAXNODE(stav)
13:     hodnota = −∞
14:     for all následníci stavu do
15:         tmp = EXPECTIMAX(následník)
16:         if tmp > hodnota then     // získej nejlepší možnou hodnotu pro Ms. Pacman
17:             hodnota = tmp
18:         end if
19:     end for
20:     return hodnota
21: end function
```

Vstup: stav hry
Výstup: hodnota uzlu pro nepřítele

```
22: function CHANCENODE(stav)
23:     hodnota = 0
24:     for all následníci stavu do
25:         // pravděpodobnost výskytu následníka
26:         p = PRAVDĚPODOBNOST(stav,následník)
27:         hodnota += p * EXPECTIMAX(následník)
28:     end for
29:     return hodnota                  // navrať průměrnou hodnotu všech následníků
30: end function
```

Provedení chování pokročilých agentů

Všichni následující agenti průběžně (na základě zjištěných Q-hodnot ke každému poli herní desky) vyhodnocují, jakou akci mají z daného pole hrací desky nejlépe provést = strategie π^* (*policy*). Chování agentů, neboli **provedení strategie** (posloupnost akcí na desce) se bude demonstrovat v rámci tzv. **epizod** (jedna epizoda = jeden pokus agenta o provedení zjištěné optimální strategie). Samotný rozdíl v návrhu vyhodnocování zjištěných V-hodnot a Q-hodnot popisují jednotlivé sekce každého agenta (Value Iteration agent 4.2, Q-Learning agent 4.2, Aproximační Q-Learning agent 4.2). Zjednodušený pseudokód návrhu vyhodnocování chování následujících agentů lze vidět na algoritmu 2. Jak lze vidět, výsledné chování agenta je prováděno podobně, avšak u Q-Learningových agentů se na základě zjištěných okamžitých odměn provede **aktualizace Q-hodnot** a také se navíc bude dále rozlišovat, zda je agent v tréninku (= provádí epizody tréninku, tedy strategie v rámci *exploration* může být i náhodná), či pouze provádí zjištěnou optimální strategii (= provádí chování během epizod stejně jako Value Iteration agent).

Algorithm 2 Pokročilí agenti – provedení strategie

```
Vstup: herni deska, agent, počet epizod
procedure BEHAVIOUR(deska,agent,maxEpizod)
    for i = 0; i < maxEpizod; i++ do stav = počáteční stav
        terminální stav nemá žádné připustné akce
        while stav != terminální stav do
            akce = agent.GETPOLICY(stav)                    // získej strategii
            // proveď strategii pro stav, získej odměnu
            (následník,odměna) = deska.DOPOLICY(stav)
            // pokud učící agent trénuje (neprovádí pouze zjištěnou strategii)
            if agent.typ == QLearnAgent or agent.typ == ApproxQLearnAgent then
                if agent.trenink then
                    // aktualizuj novou hodnotu pro pár (stav,akce)
                    agent.QUPDATE(stav,akce,následník,odměna)
                end if
            end if
            stav = následník
        end while
    end for
end procedure
```

Value Iteration agent

Tento agent je **pasivní plánovací agent** – provede veškeré své výpočty odhadů hodnot pomocí daného modelu světa a naplánuje si po každé iteraci akci pro danou hodnotu. Přepočet odhadů V-hodnot probíhá vždy do určitého stanoveného počtu k iterací (podobné jako hloubka, agent by měl provést k-kroků výpočtu odhadů V-hodnot), nebo dokud metoda nekonverguje. Teprve po tomto provedení přepočtu hodnot je k dispozici optimální strategie pro herní desku a přichází doba provedení chování agenta ve formě epizod.

Velký vliv na průběh tohoto a následujících algoritmů má koeficient odměny γ – čím je koeficient menší, tím se vyhodnocení hodnot zpomaluje a důkladněji se tak prozkoumá více

hodnot (výsledné vyhodnocení posloupnosti akcí může například trvat déle vyhodnotit, ale pravděpodobnost optimality výsledné strategie bude větší).

Dalším důležitou konstantou je odměna po každém agentově kroku (*living reward*), tedy ohodnocení pohybu z jednoho neterminálního stavu do druhého. Odměna může být i záporná – čím je menší, tím rychleji se agent snaží najít terminální uzel a skončit tak hru/problém.

Poslední důležitý vliv na algoritmus má náhodnost pohybu agenta (*noise*, která modeluje stochastičnost akce, tedy pravděpodobnost, že se daná akce provede – ovlivňuje, jak často agent riskuje. *Living reward* a *noise* budou spravovány herní deskou.
Algoritmus bude implementován dle rovnic z teoretické sekce 3.3. Agent tedy potřebuje (mj.) pro každou iteraci k průběžně ukládat vektor **vHodnoty** všech stavů ($V\,[k]$), který vyhodnocuje své hodnoty pro stav na základě Q-hodnot stavu (odhady nejlepších akcí ze stavu do následníka dle rovnice 3.6). Dále je potřeba především **model** definující přechodovou a odměnovou funkci. Model musí být sestaven na základě stavů herní desky ještě před začátkem samotné metody. Pro ukázku si lze uvést vyhodnocení Q-hodnoty stavu na základě modelu v pseudokódu 3:

Algorithm 3 Value Iteration – pseudokód získání Q-hodnot

Vstup: model,aktuální stav,vybraná akce, gamma, vektor V-Hodnot (vHodnoty)

```
1: qHodnota = 0.0
2: for all naslednici z modelu do
3:     pravděpodobnost přechodu z jednoho stavu do druhého
4:     p = model.prechodovaFunkce(stav,akce,naslednik)
5:     získaná odměna přechodu
6:     r = model.odmenovaFunkce(stav,akce,naslednik)
7:     qHodnota += p * ( r + gamma * vHodnoty[naslednik])
8: end for
```

Výstup: Q-Hodnota(stav,akce) (qHodnota)

Q-Learning agent

Zatímco metoda Value Iteration má k dispozici Q-Hodnoty z modelu, z nichž si sestavuje vektor V-Hodnot, Q-Learning agent se během tréninku **aktivně učí** Q-hodnoty akcí (jejich pravděpodobnosti výskytu i odměny) a tím průběžně vytváří vlastní odhady Q-hodnot. Na výpočet Q-hodnot během tréninku má kromě již uvedených parametrů (z předchozí sekce) velký vliv parametr $\epsilon - greedy$, který určuje určuje pravděpodobnost, zda se má provést optimální (pravděpodobnost $1 - \epsilon$), či náhodná akce (pravděpodobnost ϵ) *(exploitation/exploration ratio)*. Čím je tento parametr vyšší, tím pravděpodobněji se agent chová dle své akuální optimální akce. Dalším důležitým faktorem je α, která ovlivňuje rychlost učení. Čím je větší alfa, tím víc se dává důraz na nové vzorky. Experimentální část práce 6 se také zaměřuje na co nejvhodnější stanovení těchto parametrů a jejich vliv na kvalitu výsledné umělé inteligence pro daný problém.
Hlavní náplní agenta je sestavit tabulku aktuálních odhadů Q-Hodnot ke všem párům (stav,akce). Tabulka se postupně zaplňuje a obnovuje novými Q^*-hodnotami zjišťovanými aktivně po provedení akce z jednoho stavu do svého následníka dle rovnic v sekci 3.4 (vyvoláno aktualizací hodnot).

Vyhodnocování a sestavování tabulky Q-Hodnot během agentova tréninku lze vidět na pseudokódu inicializace a aktualizace hodnot 4, který návazuje na pseudokód 2. Aktualizace hodnot vychází přímo z rovnice 3.15).

Algorithm 4 Q-Learning – pseudokód

```
Vstup: herní deska, agent, počet Epizod, stavy, akce, tabulka Q-Hodnot qHodnoty
Vstup: globálně – alfa, gamma, epsilon
procedure QLearning(deska,agent,maxEpizod,stavy,akce,qHodnoty)
    for all qHodnoty do
        for all pripustne akce do
            qHodnoty[stav,akce] = 0.0                          // inicializace
        end for
    end for
    // aktivně prováděj akce dle ϵ a updatuj qHodnoty
    behaviour(deska,agent,maxEpizod)
end procedure
```

```
Vstup: stav, akce, naslednik, získaná odměna
Vstup: globálně – alfa, gamma, epsilon, tabulka Q-Hodnot qHodnoty
procedure QUpdate(s,a,novýS,r)
    // získej odhad hodnoty budoucí optimální akce
    maxHodn = getValue(novýS)
    // proveď aktualizaci (update) TD učení
    qHodnoty[s,a] += alfa * ((r + gamma * maxHodn) – qHodnoty[s,a])
end procedure
```

Aproximační Q-Learning agent

Pro vyhodnocování chování agenta není potřeba ukládat tabulku Q-hodnot pro každý pár (stav, akce), ale **váhy vlastností** a **vektor vlastností** (*feature vector*), tedy vektor namapování dané vlastnosti na její aktuální hodnotu. Vektor vlastností předpovídá odhad Q-hodnot jednotlivých párů $(stav, akce)$, které se budou využívat při vyhodnocení optimální strategie. Důležité je především korektě stanovit typy vlastností a jejich ohodnocení (podobně jako ohodnocovací funkce stavu):

- vzdálenost nejbližšího ducha a jeho stav,
- vzdálenost nejbližší kuličky jídla,
- vzdálenost nejbližšího power-upu,
- počet zbývajícího jídla a power-upů na herní desce,
- počet duchů,
- pozice Ms. Pacman – je v rohu?

Následně se stanoví návrh aproximační funkce vektoru na základě váhy každé vlastnosti a její funkční hodnoty, např.:

$$Q(s,a) = w_{DUCH} f_{DUCH}(s,a) + w_{JIDLO} f_{JIDLO}(s,a) + \cdots + w_{ROH} f_{ROH}(s,a) \qquad (4.1)$$

kde

- w je váha vlastnosti, f je funkce vlastnosti,
- DUCH (reprezentuje vlastnost:) vzdálenost nejbližšího vystrašeného ducha (lze sníst),
- JIDLO vzdálenost nejbližší kuličky jídla,
- ROH zda je Ms. Pacman je v rohu.

Dále bude především potřeba nalézt **optimální strategii** během epizod tréninku pomocí *(exploitation/exploration ratio)* a návrhu postupu metody Aproximační Q-Learning:

1. Aproximačně najdi model odhadů Q-hodnot během tréninku na základě hodnoty vlastnosti a její váhy (při novém přechodu proveď aktualizaci váh dle rovnice 3.21).
2. Předpokládej pouze Q-hodnoty akcí z daného stavu.
3. Opakuj předchozí body dokud nebude dost vzorků popř. neskončí trénink, jinak pokračuj na další bod.
4. Poměňuj váhy vlastností a testuj optimálnost nové strategie.

Poměňování váh vlastností takto může **maximalizovat odměny** a dojít tak k lepší strategii, něž byla doposud použita.

Kapitola 5

Realizace

Tato kapitola popisuje praktickou část práce a především se zaměřuje na **implemetace různého chování agentů**.

5.1 Zvolené prostředky

Nejdříve bylo nutné si zvolit programovací jazyk, který by zvládl výpočetní náročnost problematiky vůči vykreslování grafického dema. Zpočátku byl zvolen jazyk *Java* díky jeho vysoké míře abstrakce a intuitivnímu objektově zaměřenému přístupu jazyka. Během implementace se však narazilo na několik problémů:

1. propojení herního prostředí s reprezentací jeho stavů pro spravné vyhodnocení umělou inteligencí + následné propojení stavů s MDP,
2. výpočetní náročnost algoritmů versus množství stavů ve hře,
3. snaha co nejvíce dodržet matematické vzorce z teorie,
4. názorné vykreslení výsledků složitějších algoritmů,
5. časová náročnost implementace logiky a GUI hry versus zaměření práce na umělou inteligenci.

Nakonec po sérii experimentování s jazykem se kvůli zmíněným problémům přešlo k hledání náhradního řešení. Tím se stalo hledání aplikace, nebo frameworku, který řeší podobnou problematiku. Základem pro studii a experimentování s algoritmy nakonec posloužil pythonový framework od univerzity UC Berkeley [3], používaný v jejich výuce Úvod do umělé inteligence (kurz CS188) [6]. Hlavním důvodem zvolení frameworku tedy bylo především zaměření práce na experimentování s algoritmy umělé inteligence (oproti implementaci samotného jádra hry). Dalším důvodem volby je použití jazyka *Python 2.7.5*, který též umožňuje objektový přístup, je dobře čitelný a nenáročný – nejen díky těmto kladům se dají dobře pochopit vlastnosti a propojení jednotlivých objektů frameworku. Posledním faktorem volby byl také samotný nápad zaměření této práce – přednášky kurzu UC Berkeley napomáhaly při analýze teorie pro tuto práci. Framework primárně posloužil jako prostředek pro realizaci a srovnání algoritmů a jejich názorné zobrazení výsledků. Pro účely této práce budou v následující sekcích popsány nejdůležitější části aplikace, detailní popis implementace frameworku a vlastních algoritmů lze najít v komentářích v kódu. Ovladání frameworku lze nalézt v příloze B.

5.2 Framework a reprezentace stavu hry

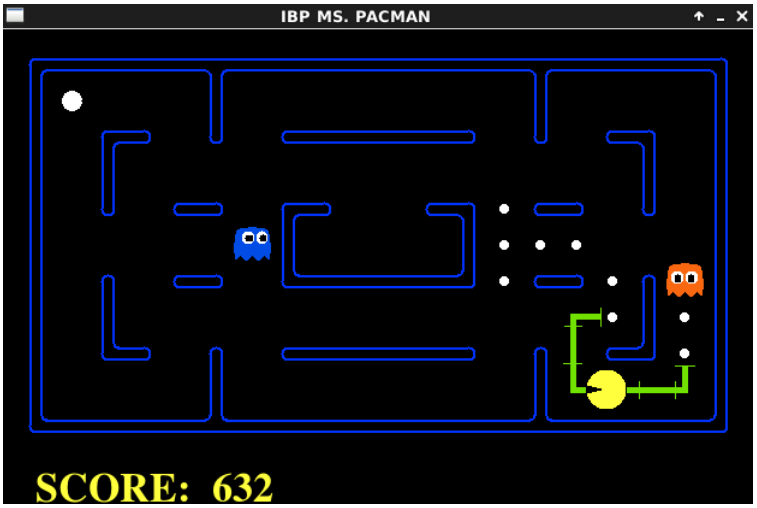
Pro všechny algoritmy je nejdůležitější částí frameworku objekt `GameState` reprezentující **stav hry**, který lze najít ve spustitelném souboru `pacman.py`. Tento soubor a soubor `game.py` tvoří samotné jádro frameworku pro problematiku Ms. Pacman – definují základní chování agentů, možné akce apod. Dalším důležitým spustitelným souborem pro problematiku názornějšího zobrazení V a Q hodnot ve formě několika gridworld problémů je `gridworld.py`. GameState dále pracuje s objekty `PacmanRules` a `GhostRules`, které zajišťují logiku přípustných akcí pro svého agenta(agenty). Tento základní objekt tedy obsahuje:

- pozice Ms. Pacman a duchů,
- pozice zbývajícího jídla v mřížce,
- pozice zbývajících power-upů v mřížce,
- pozice zdí v mřížce,
- počet duchů,
- možné akce, jež může daný agent provést z aktuálního stavu,
- schopnost provést akci a vytvořit tak svého následníka (opět objekt `GameState`).

Každý agent dědí ze základního objektu `Agent` povinnou metodu `getAction` (zavazuje agenta přijmout objekt `GameSate` a vrátit na něj reakci v podobě akce) a proměnnou `index` definující index agenta (Ms. Pacman má tradičně index = 0). Implementace vlastního chování konkrétního typu agenta je již definována v jeho vlastním objektu (jako override). Např. `AlphaBetaAgent` přetěžuje metodu `getAction` jako algoritmus Alfa-Beta řezy atp.

5.3 Reflexivní agent a jeho ohodnocovací funkce

Pro další srovnání s algoritmy byl implementován tento základní typ neadversárního reflexivního agenta `ReflexAgent`. Agent pouze vnímá aktuální stav hry (hloubka = 1) a jeho ohodnocovací funkce vyhodnocuje stav hry na základě informací o celém stavu hry. Chování reflexivního agenta je založeno na jednoduché ohodnocovací funkci svých následníků: agent se nejdříve podívá na ohodnocení stavů, tedy jejich užitek, po podniknutí všech přípustných *Akcí* (jít vlevo, vpravo, nahoru, dolů, zastavit se) a následně pseudonáhodně vybere výslednou akci stavu s nejlepším ohodnocením. **Ohodnocovací funkce** agenta musí vzít v potaz především informace o vzdálenosti jídla, duchů atp. Dále se bere v potaz stav ducha, zda ho dokáže Ms. Pacman dohonit tzn. kolik mu ještě zbývá kroků ve stavu vystrašenosti (Ms. Pacman snědla power-up) vůči jeho vzdálenosti od Ms. Pacman. Čím lepší vlastnosti hodnoceného následníka stavu, tím větší ohodnocení funkce vrací. Právě ohodnocovací funkce se překvapivě stala největším problémem při implementaci tohoto agenta, neboť neustále narážela na problematické nesnadno vyřešitelné stavy stejného ohodnocení např. stejně vzdálené jídlo od agenta, jak lze vidět na obrázku 5.1. Prakticky to pak dopadá, že agent doslova *čeká* na ducha, aby se konečně mohl dostat ze zacykleného chování. Pro účely srovnání s následujícími agenty však tento agent postačí.



Obrázek 5.1: Na screenshotu lze vidět problematické místo pro ohodnocovací funkci agenta. Zelená čára reprezentuje stejnou manhattonovskou vzdálenost (o velikosti 3 políčka) mezi Ms. Pacman a nebližší kuličkou jídla.

5.4 Základní algorimy

Základní algoritmy vč. reflexivního agenta jsou implementovány pro problém Ms. Pacman (a její různé **mapy bludiště** ze složky `src/layouts`) a jejich spuštění je detailně popsáno v přílohové části manuál B.1. Algoritmy musí mít přehled, která strana je na tahu na základě indexu agenta a také o kolikátou vrstvu střídání Ms. Pacman vs. (1-N) duchů se jedná – to zajišťuje jejich instanční proměnná `depth`. Pro časovou a výpočetní únosnost byla zvolena výchozí hodnota hloubky `depth` 2. Pro správnou funkcionalitu frameworkového návrhu získávání agentem zvolené akce metodou `getAction` se musí průběžně vracet nejen aktuální hodnota uzlu, ale i **akce**, která je pro daného agenta optimální (zda se jedná o maximalizační nebo minimalizační vrstvu (popř. včetně vlivu pravděpodobnosti u `ExpectimaxAgent`)). Během vyhodnocování se souběžně pracuje s polem `[akce,hodnota]` a následně, až se algoritmy dostanou ke kořeni (agent Ms. Pacman) je vrácena pouze vyhodnocená akce, jež má agent právě provést. Během implementace se ukázalo, že je názornější zakázat agentovi se zastavit (vynechat z vyhodnocení přípustnou akci `STOP`), aby v případech s podobným ohodnocením stavů, jako u reflexivního agenta bylo jasně vidět zacyklené chování agenta. Výňatek z kódu 5.4 ukazuje metodu `getAction` pro získání akce u agenta `AlphaBetaAgent`[1].

```
def getAction(self, gameState):
    # ziskani hodnot z metody AlfaBeta objektu AlphaBetaAgent
    value = self.AlphaBeta(gameState, self.index, 0,
    float("-inf"), float("inf")) # inicializace parametru alfa a beta
    return value[0] # vraceni akce pro hru
```

[1]Komentáře v kódu jsou psány anglicky, aby navazovaly na framework. Výňatek má komentáře přeloženy.

Ohodnocovací funkce

Vyhodnocovací funkce následníků `evaluationFunction` pro agenty `MinimaxAgent` a `AlphaBetaAgent` je založena na jejich skóre, jak již bylo zmíněno v sekci 4.2. Tato funkce se měla původně nasadit i na agenta `ExpectimaxAgent`, avšak během implementace přišel nový nápad, a sice využití lepší vyhodnocovací funkce `betterEvaluationFunction` pro stav tohoto agenta. Tento nápad byl dodatečně doplněn i do návrhové části agenta v sekci 4.2. Funkce implementuje lineární kombinaci hodnot dle faktorů ovlivňujících stav Ms. Pacman. Funkce je založena především na kombinaci proměnných:

- `foodFactor` – pozitivně vnímá: vzdálenost nejbližšího jídla, počet zbývajících power-upů, počet zbývajících kuliček jídla (nutnost odečítat nebo být dělitelem, aby platilo: Čím menší hodnota, tedy vzdálenost jídla atp., tím větší a potažmo lepší bude `foodFactor`)
- `ghostFactor` – pozitivně vnímá: vystrašený chytitelný duch; negativně vnímá: vzdálenost nejbližšího nevystrašeného ducha menší jak 4 políčka (přičtením duchovy vzdálenosti tak, aby platilo: Čím je blíž, tím je `ghostFactor` nižší) atp.

Dohromady tyto faktory společně se skórem stavu dávají finální ohodnocení stavu `value`, které prochází algoritmem Expectimax.

5.5 Framework a MDP problematika

Hlavním objektem pro pokročilé agenty je abstraktní objekt `ValueEstimationAgent` (dědící od základní třídy `Agent`; v souboru `learningAgents.py`). Tento agent slouží především pro propojení prostředí zkoumaných problémů (Ms. Pacman, gridworld) s Q-hodnotami párů (stav,akce), dále propojení stavu s jeho V-hodnotou ($V(s)$) a nakonec propojení výsledných hodnot s výslednou nalezenou strategiií ($\pi(s)$). Agenti od něj dále dědí a přetěžují základní metody:

- `getQValue(state,action)` – vrací Q-hodnotu pro pár (stav,akce),
- `getValue(state)` – vrací V-hodnotu pro stav,
- `getPolicy(state)` – vrací strategii (akci) pro daný stav po i, při vyhodnocení chování agenta daným algoritmem na základě `getQValue(state,action)`,
- `getAction(state)` – vrací strategii pro stav (přímo bez *exploration*, používáno prostředím pro inicializaci).

ValueIterationAgent (ze souboru `valueIterationAgents.py`) navíc potřebuje pro svoje chování model (přechodová funkce T, odměnová funkce R) a využívá pro to abstrakní třídu `MarkovDecisionProcess` ze souboru `mdp.py`. `MarkovDecisionProcess` je dále využívána prostředím (třída `Gridworld` reprezentující gridworld problémy v souboru `gridworld.py`), které napojuje své stavy (souřadnice polí mřížky a jejich odměna) a akce pomocí metody `getTransitionStatesAndProbs(state, action)` na jejich pravděpodobnost výskytu. Metoda vrací list párů $((s, a, s'), T(s, a, s'))$, tedy (následník aktuálního stavu,jeho přechodová funkce = pravděpodobnost výskytu). Abstraktní `ReinforcementAgent`, který je základním objektem pro všechny agenty strojového učení, se stará o chování učícího se agenta během

průběhu jednotlivých epizod tréninku a během provádění výsledné strategie, dále řeší základní vstupní parametry pro agenty jako nastavení počtu epizod k provedení (instanční proměnná `numTraining`) atp. a v případě Ms. Pacman spravuje průběžný a finální výpis během epizod. K výpisu pro Ms. Pacman je potřeba uchovávat akumulované odměny z tréninku `accumTrainRewards` a akumulované odměny z testování získané strategie tréninku `accumTestRewards`. Ty se nakonec podělí počtem provedených epizod (tréninku nebo testování) a takto vyjdou průměry odměn pro výpis[2]. Důležitými metodami jsou:

- `observeTransition(state,action,nextState,deltaReward)` – voláno prostředím (Ms. Pacman, gridworld), které informuje agenta, že je získána nová hodnota přechodové funkce (nový přechod do stavu po akci), aby se následně vyvolala **aktualizace (update) Q-hodnot** a předala se mu okamžitá odměna jako rozdíl skóre následníka vůči aktuálnímu stavu (proměnná `deltaReward`),
- `update(state, action, nextState, reward)` – aktualizace nových odhadů Q-hodnot na základě získané okamžité odměny přechodu `delyaReward`; agenti tuto metodu přetěžují ve vlastním objektu,
- `final(state)` – finální výpis pro Ms. Pacman.

`ReinforcementAgent` staví na odhadu tabulky Q-Hodnot, protože neobsahuje model a je rodičem objektu řešící Q-Learning pro gridworld problémy `QLearningAgent`. Podobný agent, avšak s odlišnými vstupními parametry pro problém Ms. Pacman jsou `PacmanQAgent` a nakonec dále dědící `ApproximateQAgent`, který navíc implementuje algoritmus Aproximační Q-learning a využívá u toho třídy `FeatureExtractor` pro implementaci vektoru vlastností pro Ms. Pacman.

5.6 Value Iteration agent a gridworld problémy

Objekt `ValueIterationAgent` provádí své vyhodnocení V-Hodnot již v konstruktoru a to do pevného počtu iterací. Důvodem opuštění podmínky konvergence je především možnost pozorovat kolik iterací stačí pro optimalitu strategie. Z vypočetních (nutnost stanovit model) a názorných důvodů je tento agent zobrazen výhradně formou **gridworld** problémů (mřížky hodnot začínající se definovanými odměnami spustitelné pomocí `gridworld.py` – viz manuál B.2), kde jedno políčko po provedení metody ukazuje svou výslednou V-hodnotu (po provedení počtu daných iterací), po zmáčknutí klávesy (např. "Q") své 4 Q-hodnoty pro každý směr/akci a agent následně provede stanovený počet epizod svého chování (agentovo provedení vyhodnocené strategie). Průběh implementace tohoto agenta byla pravděpodobně nejdelší co vůbec do praktického pochopení iterativního charakteru V-hodnot. Během implementace se ukázalo, že vektor hodnot stavů stačí ukládat pouze do jedné proměné a po každé iteraci jej znovu přepisovat. Původní myšlenka totiž byla, že vektor si bude ukládat i jakousi částečnou historii stavů včetně Q-hodnot, ze kterých jsou V-hodnoty sestavovány. Tento nápad byl po naprogramování okamžitě zavržen vzhledem k jeho výpočetní náročnosti a celkové nerelevantnosti k MDP problémům. Vždyť právě myšlenka MDP je pouze vycházet z přechodů z aktuálních stavů do jejich následníků. Ustanovila se tedy jedna nejdůležitější instanční proměnná `values` sloužící pro uložení aktuálních V-hodnot ve formě *slovníku* pro každé pole mřížky a její aktuální V-hodnotu např. [(0,1):-10.0,(5,3):0.0]). Po každé iteraci se V-Hodnoty **přepíší** na aktuální, to lze vidět na výňatku z kódu konstruktoru agenta 5.6:

[2] o výpis pro gridworld problémy se stará metoda `runEpisode` z `gridworld.py` během provádění epizod

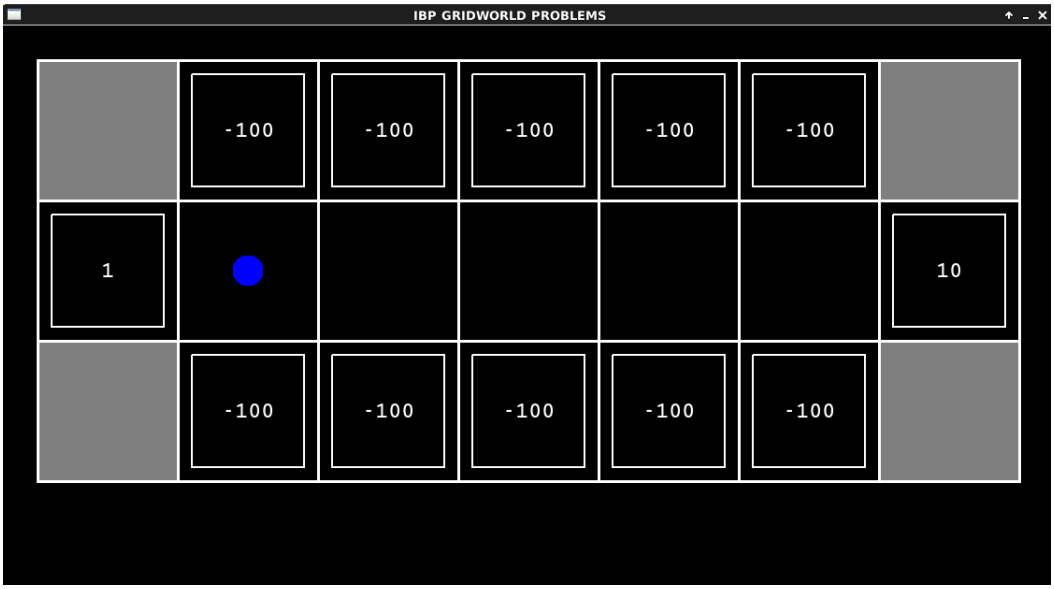
```
# inicializace instancnich promennych
self.mdp = mdp                    # MDP model pro gridworld
self.discount = discount      # gamma
self.iterations = iterations # pocet iteraci
self.values = util.Counter() # slovnik V-hodnot

for k in range(iterations):
   futureValues = self.values.copy()    # V_{k+1} ← V_k
   for state in self.mdp.getStates():  # pro kazdy stav modelu (1 pole mrizky)
      # ziskej pripustne akce pro stav
      actions = self.mdp.getPossibleActions(state)
      # pokud je stav terminalni, uloz V[stav]_{k+1} = 0 a pokracuj na novy stav
      if mdp.isTerminal(state) or len(actions) == 0:
         futureValues[state] = 0
         continue
      # ziskej nejlepsi hodnotu z odhadu uzitku vsech akci prechodu
      # = z Q-hodnot
      value = float("-inf")
      for action in actions:
         # suma Q-hodnot pro kazdou akci
         expectedQValue = 0.0
         # napojení na model - ziskani Q-hodnot pro stav
         expectedQValue = self.getQValue(state, action)
         if expectedQValue > value:
            value = expectedQValue
      # V*(s) - nejlepsi hodnota ocekavaneho budouciho uzitku
      futureValues[state] = value
   self.values = futureValues # prepis V-Hodnoty na aktualni
```

Výsledné provedení chování agenta je podobné jako u Q-Learning Agentů – viz výňatek z kódu 5.7, avšak lze pro optimalitu strategie provést až po provedení iterací `values` v konstruktoru agenta 5.6. Agent neprovádí trénink (nevyužívá $\epsilon-greedy$), takže se vždy provede nejlepší možná akce stavu na základě odhadů Q-hodnot z modelu (které jsou sestaveny dle pseudokódu 3) pomocí volání metody na získání strategie `getPolicy(state)`.

Framework nabízí hned několik **gridworld problémů**, avšak nejzajímavějšími jsou:

- `BookGrid`, který byl již ukázán v teoretické části (obrázky 3.1 a 3.2).
- `DiscountGrid`, problém zahrnující 2 terminální cíle (s pozitivní odměnou) a útes 4 terminálních stavů s velmi negativní odměnou. Agent se potýká s tím, zda jít delší bezpečnou cestou, nebo kratší riskantnější cestou podél útesu a také s faktem, že cíle mají rozdílné odměny (vzdálenější má hodnotnější odměnu). Rozdílné chování agenta je ovlivňováno instanční proměnnou `gamma` (koeficient odměny), počtem provedených iterací a stochastičností agentových akcí. Proto je tento problém ideální na zkoumání.
- `BridgeGrid`, problém modelující most mezi pozitivními terminálními cíli podél útesů z každé strany. Ukázku tohoto gridworld problému lze vidět na obrázku 5.2.

Tyto problémy budou experimenálně otestovány v následující kapitole 6.



Obrázek 5.2: Screenshot počátečního modelu gridworld problému `BridgeGrid` pro Value Iteration agenta (Q-Learning agent toto ohodnocení odměn stavů nemá) ukazuje stavy a jejich odměnu. Agent je reprezentovaný modrou tečkou.

5.7 Q-Learning agenti

Jelikož bude Q-Learning nasazen jak na gridworld problémy, tak na Ms. Pacman, je potřeba tyto agenty rozlišovat na základě jejich vstupních parametrů, které byly poskytnuty frameworkem:

- pro `QLearningAgent`: `alpha` = 0.5, `epsilon` = 0.5, `gamma` = 1.0.
- pro `PacmanQAgent` a `ApproximateQAgent`: `alpha` = 0.2, `epsilon` = 0.05, `gamma` = 0.8.

Po analýze těchto parametrů se došlo k těmto závěrům:

1. Parametry pro gridworld problémy jsou nastaveny do výchozích hodnot, které budou během experimentování měněny, takže příliš nezáleží jejich výchozí hodnotách.
2. Parametry pro Ms. Pacman byly experimentálně zjištěny autory frameworku tak, aby se Ms. Pacman učila spíše pomaleji (velmi malá hodnota parametru alpha) a měla následně uživatelem nastavený delší trénink, spíše málo experimentovala (vliv malé hodnoty parametru epsilon) a snažila se spíše co nejrychleji sníst kuličky jídla (vliv sníženého parametru gamma). Parametr `livingReward` je v demu Ms. Pacman záporná odměna při její neaktivitě (když nejsou nejseny kuličky jídla).

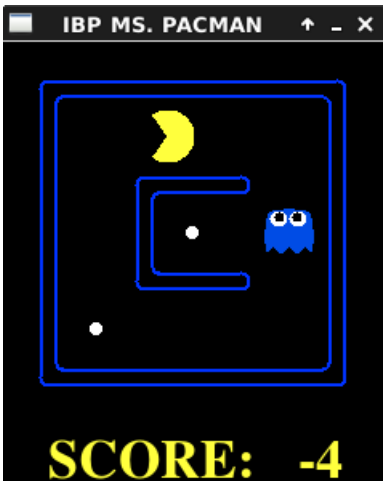
Jak již bylo řešeno v 4.2, tito agenti si musí model odhadnout sami na základě získaných odměn při výběru akcí na pomocí *exporation/exploitation ratio*. Tento poměr je dán instanční proměnnou `epsilon` a předán funkci `flipCoin`, jež určí finální rozhodnutí výběru akce. Pokud se má provést aktuální akce, volá se pro stav metoda `getPolicy(state)`, která na základě tabulky Q-hodnot (Q-hodnota je vracena přímo opět metodou `getQValue(state,`

`action)` narozdíl od **ValueIterationAgent**, který si Q-hodnotu musí nejdříve sestavit na základě modelu, jak je vidět v návrhu pseudokódu 3).

Pro názornost si lze uvést výňatek z kódu (část vyhodnocení strategie pro stav během tréninku Q-Learning agentů) 5.7. Agenti si ukládají pouze tabulku aktuálních odhadů Q-hodnot jako slovník `qValues` pro každý pár `(state,action)` a vychází z návrhu pseudokódu 4.
POZNÁMKA: V jazyce Python je instance třídy metodě posílána automaticky, avšak přijímána již automaticky není. Proto je potřeba napsat do argumentů funkce klíčové slovo *self*.

```
def getAction(self, state):
    actions = self.getLegalActions(state)
    # terminalni stav
    if len(actions) == 0:
        return None
    # pravdepodobnost dana ϵ − greedy
    if flipCoin(self.epsilon):
        # prozkoumavani novych stavu behem treninku
        return random.choice(actions) # nahoda akci pro stav (exploration)
    else:
        # nejlepsi mozna akce na zaklade tabulky odhadu Q-hodnot (exploitation)
        return self.getPolicy(state)
```

Již během implementace bylo zjištěno, že vzhledem k množství párů (stav,akce) bude Q-Learning agent Ms. Pacman (`PacmanQAgent`) testován spíše na menších mapách a určitě bude potřeba nastavit větší počet epizod tréninku, pro ukázku lze uvést příklad: 6.



Obrázek 5.3: Screenshot menší mapy (`smallGrid`) pro Ms. Pacman.

5.8 Aproximační Q-Learning agent

Agent `ApproximateQAgent` využívá proměnnou `weights`, tedy slovník párů *(vlastnost,váha)* (výchozí hodnota váh je rovna 0.0), kterými se pronásobí její hodnota. Hodnota vlastnosti

je získána voláním metody `getFeatures(state, action)` objektu `BetterExtractor` (dědící ze základního objektu `FeatureExtractor`)), který ji vrací opět formou slovníku párů *(vlastnost,hodnota)*, které jsou uloženy v instanční proměnné objektu `features`. Metoda `getPolicy(state)` zůstává stejná (získání nejlepší možné akce pro stav), avšak místo aby metoda stavěla na získání uložené Q-hodnoty z tabulky (pomocí metody `getQValue(state, action)`), Q-hodnota se sestaví pomocí váh vektoru vlastností, jak je vidět na výňatku z kódu stejnojmenné metody 5.8.

```
def getQValue(state, action):
    qvalue = 0.0
    # ziskani vektoru vlastnosti s jejich hodnotami
    features = self.featExtractor.getFeatures(state, action)
    for i in features: # iteruj po vlastnosti
        # pronasob váhu vlastnosti s jeji hodnotou
        qvalue += self.getWeights()[i] * features[i]
    return qvalue
```

Aktualizace váh vlastností na základě nového přechodu (opět voláno rodičovskou funkcí `observeTransition(state, action, nextState, deltaReward)`) pak vypadá takto:

```
def update(self, state, action, nextState, reward):

    # rozdíl vzorků = [r + γ max_a' Q(s', a')] − Q(s, a)
    difference = (reward + self.discount * self.getValue(nextState))
                 − self.getQValue(state, action)
    # ziskani vektoru vlastnosti s jejich hodnotami
    features = self.featExtractor.getFeatures(state, action)
    for i in features:
        # samotna aktualizace váhy dané vlastnosti
        self.getWeights()[i] += self.alpha * difference * features[i]
```

Pravděpodobně nejdůležitější pro celého agenta je kvalitní stanovení hodnot pro stav ve vektoru vlastností. Framework nabízí několik typů extraktorů vlastností pro vektor vlastností – např. `CoordinateExtractor`, který je založený na triviálních vlastnostech typu stav, akce, koordináty stavu na gridworldu Ms. Pacman. Jelikož jsou extraktory navázány na MDP, je k dispozici informace o aktuálním stavu a jeho následníkovi po dané akci. Lze tedy generalizovat daný stav ve hře pomocí nahlédnutí na stav hry jeho následníka raději než staticky popisovat koordináty atp. Během návrhu vektoru vlastností 4.2 se počítalo až se sedmi vlastnosmi, které popíšou stav následníka. Během implementace se však ukázalo, že aproximace takovým množstvím vlastností není efektivní a některé vlastnosti, např. vzdálenost power-upu nejsou ze stavu následníka vůbec dosažitelné. U počítání vzdálenosti power-upu je potřeba větší rozhled, než pouze následníkova hloubka o velikosti 1, tudíž by byla potřeba metoda prohledávání stavového prostoru. Nalezení hodnoty této vlastnosti by pak bylo příliš vypočetně náročné vzhledem k tomu, že pravděpodobnost její změny je nižší a vlastnost nemá takový vliv na výsledný vektor. Mnohem účinnější je vzít základní dostupné vlastnosti a logicky je spolu prokombinovat. Vlastnosti používáné objektem `BetterExtractor` se tedy mezi sebou navíc logicky ovlivňují a relevantněji popisují aktuální dění stavu hry. Tento extraktor vylepšuje frameworkový `SimpleExtractor` především tím, že bere v potaz i stav duchů. Vektor vlastností objektu `SimpleExtractor` se primárně zaměřuje na všechny duchy bez rozdílu. Agent pak ale bohužel naráží na zbytečné vyhý-

bání se vystrašeným duchům, jež by mohl sníst. `BetterExtractor` tedy navíc bere v potaz i vystrašenost duchů a blízkost power-upů. Vektor vlastností objektu `BetterExtractor` popisuje vlastnosti[3]:

- `"active-ghosts-1-step-away"` - počet aktivních duchů z okolí následníka (vzdálenost 1 krok),
- `"scared-ghosts-1-step-away"` - počet vystrašených duchů z okolí následníka,
- `"can-eat"` - Ms. Pacman může jíst jídlo a nemusí se bát aktivních duchů,
- `"powerup"` - v okolí následníka je powerup a je blízko vystrašený duch,
- `"closest-food"` - vzdálenost nejbližšího jídla.

Samotná ukázka logické návaznosti vlastností je vidět na výňatku z kódu 5.8. Výňatek demonstruje, jak jednoduše lze generalizovat stav tak složitého problému jako je Ms. Pacman, aniž by bylo potřeba konkrétně popsat všechny informace o stavu hry. Pouze stačí 5 vlastností s hodnotami a jejich váhy. Není potřeba tabulky Q-hodnot pro každý pár (stav,akce), ani není potřeba iteračně přepočítávat V-hodnoty každého stavu na základě Q-hodnot akcí. Aproximační Q-Learning jde nasadit i na větší mapy a v experimentální části se i ukáže o kolik méně je potřeba tréninku k aproximaci stavů oproti sestavení celé tabulky Q-hodnot. POZNÁMKA: proměnná `numGhosts` označuje počet duchů, `(next_x, next_y)` jsou koordináty následníka na gridworldu Ms. Pacman, metoda `getLegalNeighbors(gPos, walls)` vrací list koordinát dostupných polí z okolí předané pozice `gPos`.

```
for i in range(0,numGhosts):
    gPos = state.getGhostPosition(i+1%(numGhosts+1)) # ziskani pozice ducha
    g = state.getGhostState(i+1%(numGhosts+1))    # ziskani duchova stavu
    # zjisti zda je duchova pozice v okoli naslednika
    if (next_x, next_y) in Actions.getLegalNeighbors(gPos, walls):
        if g.scaredTimer < 1:    # duch aktivni
            features["active-ghosts-1-step-away"] += 1
        else:                    # duch je vystraseny
            features["scared-ghosts-1-step-away"] += 1

# pokud neni aktivni duch blizko a naslednik obsahuje jidlo
if not features["active-ghosts-1-step-away"] and food[next_x][next_y]:
    features["can-eat"] = 1.0

# pokud naslednik obsahuje powerup a je blizko aktivni duch, jez bude po sebrani
# power-upu ke snezeni
if((next_x, next_y) in powerups and not
features["active-ghosts-1-step-away"]):
     features["powerup"] = 1.0
```

[3]hodnota vlastností se také musela pomenšit dělením tak, aby se předešlo divergenci hodnot.

Kapitola 6

Experimenty a srovnání agentů

Tato závěrečná kapitola praktické části práce se zaměřuje na experimenty a srovnání chování agentů na demu Ms. Pacman, v poslední části 6.4 dále zkoumá vztahy Q a V-hodnot na názornějším zobrazení mřížky gridworld problémů a v neposlední řadě pozoruje vlivy parametrů (jako např. γ) na inteligenci agentů pro daný gridworld problém. Experimenty byly tedy primárně vedeny na těchto dvou nezávislých demech. Pro obrázky této kapitoly byla z kapacitních důvodů práce přidána příloha C.

Agenty lze srovnat nejlépe podle průměrného dosaženého herního skóre v době vykonávání optimální strategie (trénink se nepočítá). U učících se agentů bude též pozorován počet epizod, za jak dlouho jsou schopni průměrně uspět v alespoň ve dvou třetinách z celkového počtu her. Touto mírou jsou následně prohlášeni za *úspěšné*.

Pro testování chování agentů na demu Ms. Pacman bylo vybráno několik rozdílných map. Pro porovnání všech agentů včetně výpočetně nejnáročnějšího `PacmanQAgent` byla zvolena již zmíněná menší mapa `smallGrid` . Dále bude zmíněna zajímavá mapa `trappedClassic`, kdy je Ms.Pacman již od začátku hry obklíčena nepřáteli (obrázek C.1 v příloze C). Jako zatěžkávací test iteligence agentů slouží mapa `trickyClassic`, viz obrázek C.2. Tato mapa o větším rozměru 20x13 polí obsahuje 3 nepřátele, 6 power-upů a především problematické místo o velikosti 5x4 polí plných kuliček jídla, které je navíc nebezpečně blízko poli zrodu nepřátel (*spawning point*).

6.1 Srovnání z vypočetní náročnosti

Aplikace byla z důvodu vypočetní náročnosti testována na stolním počítači s procesorem AMD FX-6300 o frekvenci 4 GHz a operační paměti 8 GB. Jako operační systém je nainstalována linuxová distribuce *ArchLinux* (64-bit). Za výchozí míru zatížení systému aplikací bylo zvoleno procentuální vytížení paměti systému, která ukazuje zvyšující se požadavky algoritmů. Každý agent si danou mapu zkusil celkem 100 krát a během této doby bylo zapsána nějvětší hodnota vytížení paměti.

Nejdříve se testovalo na menší mapě, kdy se podařilo úspěšně provést 100 her najednou s každým agentem (výchozí hodnota hloubky je 2). V tabulce 6.1 lze vidět proměrně nízké vytížení paměti pro malé mapy. Je zde například vidět, že `AlphaBetaAgent` svým přořezáváním dokázal ušetřit cca 25 % paměti. Již zde se také projevil nárůst vypočetní náročnosti Q-Learningového agenta `PacmanQAgent`, který oproti kolísavému charakteru vypočetní náročnosti základních agentů konstantně požadoval více paměti pro nové Q-hodnoty párů (stav, akce).

Tabulka 6.1: Maximální vytížení paměti při hře na mapě `smallGrid`

Agent	Procentuální vytížení paměti
`ReflexAgent`	0,3 %
`MinimaxAgent`	0,4 %
`MinimaxAagent`, hloubka=3	0,5 %
`AlphaBetaAgent`	0,3 %
`AlphaBetaAgent`, hloubka=3	0,4 %
`ExpectimaxAgent`	0,2 %
`ExpectimaxAgent`, hloubka=3	0,3 %
`PacmanQAgent`, až 1500 epizod	0,2 % - 0,8 %
`ApproximateQAgent`, až 1500 epizod	0,2 %

Tabulka 6.2: Maximální vytížení paměti při hře na mapě `trappedClassic`

Agent	Procentuální vytížení paměti
`ReflexAgent`	11,4 %
`ExpectimaxAgent`	19 %
`ApproximateQAgent`, až 1400 epizod	15,4 %

Požadavky na výkon pro větší mapu se výrazně zvedly, jak je vidět v tabulce 6.2. Bohužel pythonový framework s roustoucím časem vyžadoval roustoucí nároky na paměť, což se stalo osudným pro provádění 100 zátěžových her najednou pro většinu základních agentů. Nakonec bylo nutné přejít ke stonásobnému pouštění jedné hry u `ExpectimaxAgent` a vzhledem k průmerné době trvání 1 hry u tohoto agenta (cca minuta) je tento agent použit jako výchozí vzorek základních agentů. Z tabulky vyplývá, že pokud se vůbec podařilo nasadit agenta na takto pokročilou mapu, jeho výpočetní náročnost nebyla zase tak zásadní. V opačném případě se vytížení paměti agenty někdy pohybovalo až kolem 89 %. Bylo též vypozorováno, že neučící agenti strávili nepoměrně více času ve hře a to především vlivem jejich nedokonalé ohodnocovací funkce.

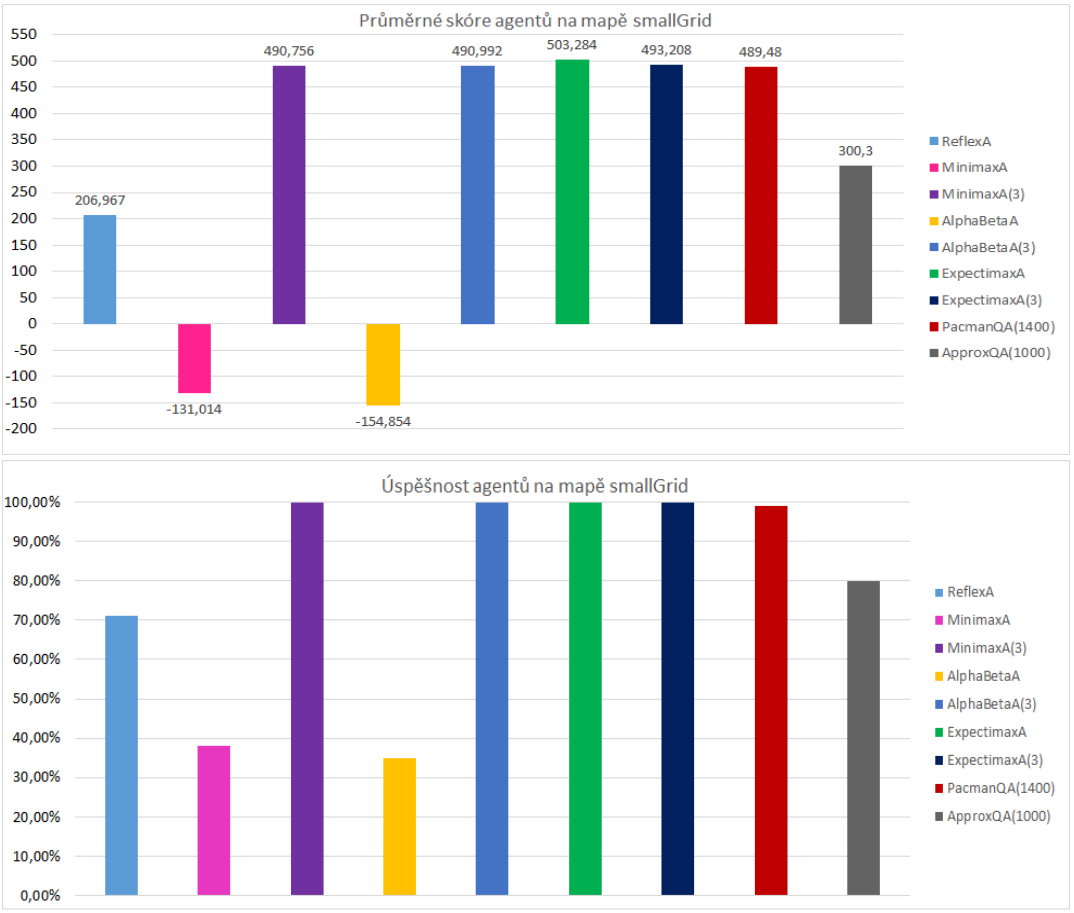
6.2 Srovnání agentů na mapách `smallGrid` a `trappedClassic`

Během testování se nejdříve zkoumal vliv ostatních parametrů na charakter učení (např. koeficient γ), vzhledem k náročnosti problému Ms. Pacman však nebyly pozorovány příliš průkazné rozdíly vlivů těchto parametrů, proto se těmto parametrům více věnuje sekce 6.4, která pracuje s jednoduššími modely. Jediný parametr $\epsilon-greedy$ projevil poměrně negativní vliv na délku učení agenta Ms. Pacman, proto se experimenty drží raději výchozího nastavení hodnot.

Výstupem experimentů na mapě `smallGrid` jsou grafy 6.1, které ukazují průměrně dosažené skóre (za 100 her) a dosaženou procentuální úspešnost výhry. Pro tento typ grafu platí, že vodorovná osa znázorňuje zkratku typu agenta a volitelný parametr (v kulatých závorkách) označuje větší hloubku 3, u základních agentů a počet provedených epizod u pokročilých agentů. Svislá osa je obvykle popsána v nadpisu grafu.

Jak lze vidět z nízkého průměrného skóre a nevalné úspěšnosti, `MinimaxAgent` a `AlphaBetaAgent` si při hloubce 2 překvapivě proti náhodnému nepříteli nevedli vůbec dobře. Tyto problémy šlo řešit pouze zvýšením hloubky na hodnotu 3. Jejich průměrná doba tahu se však někdy i ztrojnásobila a výpočetní náročnost se také zvedla. Tímto zvýšením hloubky

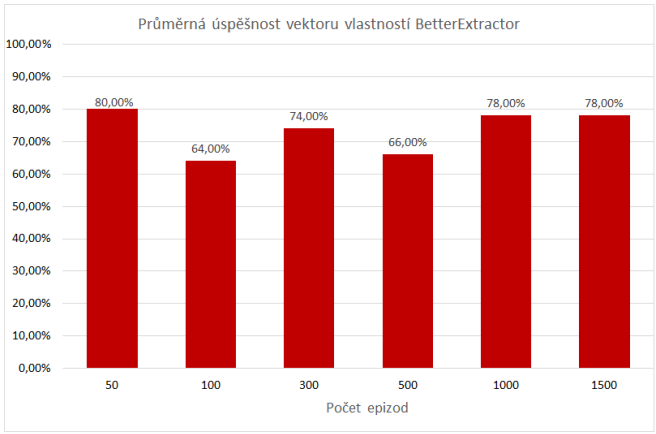
dokázaly jednoduché algoritmy obsáhnout a předvídat veškeré chování nepřítele na tak malé mapě, což se nedá příliš označit za inteligentní chování (zvýšení hloubky příliš nepomáha zvýšit inteligenci základních agentů na větších mapách). Naopak `ExpectimaxAgent` si vedl velmi dobře i s hlubkou 2, průměrné skóre agenta bylo nejvyšší oproti všem ostatním a to dokonce oproti stejnému agentu s vyšší hloubkou. Tato mapa posloužila jako příklad, kdy pro menší rozměry stavového prostoru postačí základní algoritmy. Pokud je prostředí navíc stochastické, bohatě postačí algoritmus Expectimax.



Obrázek 6.1: Srovnání agentů na menší mapě.

Druhou částí experimentů bylo zkoumání, kolik je potřeba epizod tréninku, aby byli učící agenti úspěšní (vyhrávali ve většině případů). Procentuální úspěšnosti agentů jsou vidět na grafu 6.3. Q-Learningový agent `PacmanQAgent` se stal úspěšný při již při cca 1300 epizodách tréninku, avšak jeho roustoucí výpočetní nároky byly již na takto malé mapě zde patrné. Optimalita agentových akcí při tomto počtu iterací překonala agenta `ApproximateQAgent`. Poslední agent však dokázal optimalizovat své akce a dosáhnout úspěšnosti již při při cca 25 epizodách tréninku (!). Bohužel narozdíl od Q-learningového agenta, inteligence agenta se narůstajícím počtem vzorků tréninku příliš nezlepšovala. Reakcí na tento nezdar byly pro vektor vlastností `BetterExtractor` provedeny další experimenty, které potvrdili stagnaci výsledků. Tato částečná stagnace úspěšnosti je zapříčiněna samotným charakterem aproximační funkce, avšak do budoucna by bylo potřeba určitě znovu prověřit a upravit vyhodnocení jednotlivých vlastnosti vektoru.

Obrázek 6.2: Srovnání úspěšnosti vůči počtu epizod metody Q-Learning a Aproximační Q-Learning.

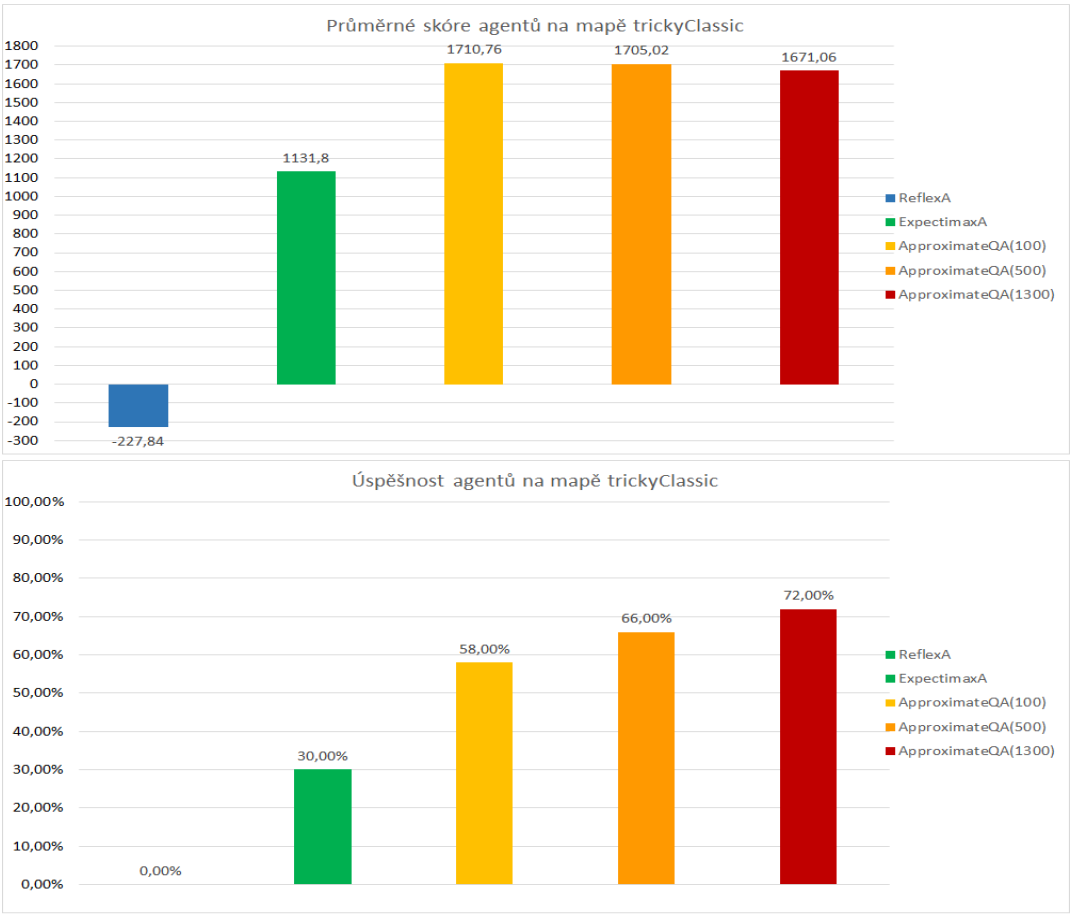


Obrázek 6.3: Stagnace výsledků vektoru vlastností Aproximačního Q-Learningu.

Kromě předchozí mapy bylo zkoumáno chování agentů i na mapě `trappedClassic`. Během experimentů se ukázalo, že je tato mapa zajímavá především kvůli sebevražedným pokusům základních agentů `MinimaxAgent` a `AlphaBetaAgent`. Agenti si pesimisticky spočtou své naděje na přežití a raději se zabijí a skončí tak hru, ačkoliv je jejich nepřítel stochastický. Naproti tomu `ExpectimaxAgent` a pokročilí agenti počítají s náhodou. Tato mapa posloužila jako příklad velkého rozdílu vyhodnocení chování základních agentů.

6.3 Zatěžkávací zkouška na mapě `trickyClassic`

Tato mapa slouží především jako zatěžkávací test pro agenty zvládající její výpočetní náročnost (problémy byly zmíněny v úvodní části této kapitoly). Jak je vidět na grafech 6.4 `ReflexAgent` si již nedokázal poradit s otevřenými plochami mapy a větším množstvím nepřátel této velké mapy. `ExpectimaxAgent` dokázal vyhrát alespoň ve 30 % případů a udržet si již poměrně vysoké průměrné skóre nad 1000 bodů. Experimenty tedy ukázaly, že kvalita ohodnocovací funkce tohoto agenta je nedostačující pro takto náročnou mapu. Agent se také často zbytečně zasekával v rozích. To řeší až `ApproximateQAgent`, který se své optimální chování postupně naučil už po cca 100 epizodách tréninku. Naopak tabulka Q-Learningového agenta tuto mapu nedokázala pojmout a agent nedokázal uspět ani po 2000 epizodách tréninku, proto není zanesen do grafu. `ApproximateQAgent` zde jasně prokázal, že dokáže rychle získat určitou úroveň inteligence, oproti němu `PacmanQAgent` těžce zaostává mnohonásobným počtem iterací nutných pro svou úspěšnost a neúnosností své tabulky Q-hodnot[1].



Obrázek 6.4: Srovnání agentů na zatěžkávací mapě.

[1]Postupně rostoucí tabulka hodnot párů (stav, akce) Q-learning agenta si exponenciálně nárokuje víc a víc výpočetního výkonu až do té míry, že přístup do této tabulky není výpočetně použitelný pro větší mapu.

6.4 Gridworld problémy a jejich parametry

Jak již bylo řečeno v sekci 5.6, závěrečné experimenty se prováděly na 3 popsaných problémech (`BookGrid`, `BridgeGrid`, `DiscountGrid`). Výchozími hodnotami parametrů experimentů jsou `alpha` = 0.5, `epsilon` = 0.5, `gamma` = 1.0, `livingReward` = 0.0 (pro jednodušší vyhodnocení experimentů a urychlení jednání agenta), `noise` = 0.2, z nichž se při experimentování vycházelo při poměňování hodnot posledních třech parametrů a sledování jejich vlivu výslednou strategii především na agenta `ValueIterationAgent`.

Nejdříve se zkoumaly vztahy mezi Q a V-hodnotami pro agenty `ValueIterationAgent` a `QLearningAgent`. Postupně bylo ověřeno, že mřížka V-hodnot gridworld problémů udává, jak (průměrně) dobrý je tento stav pro agenta `ValueIterationAgent`, pokud bude agent dodržovat svou aktuální strategii pro stav. Akumulovaná V-hodnota stavu tedy udává maximální Q-hodnotu stavu a už nebere v potaz ostatní nižší Q-hodnoty pro zbylé možné akce daného stavu. Problém metody Value Iteration tedy nastává například v situaci, kdy Q-hodnota akce stavu je špatná (menší než ostatní), avšak V-hodnota ji přesto stále bere v potaz a neustále iterativně vyhodnocuje, ačkoliv vyhodnocení špatné Q-hodnoty akce již nemá vliv na V-hodnotu. Dalším problémem je častá situace, kdy metoda Value Iteration již nalezla optimální strategii, avšak její V-hodnoty se stále zbytečně mění a přepočítávají (přepočet hodnot a následný výpočet optimální strategie jsou provázány). Dokonce bylo zjištěno, že minimální přepočet hodnot pro alespoň semi-optimální strategii agenta Value Iteration je tolik iterací k, kolik agentovi nejméně zabere dostat se do úspěšného terminálního stavu (!). Pro např. `BookGrid` tak stačilo pouze 6 iterací V-hodnot, jak lze vidět na obrázku C.3.

Dále bylo na základě poměňování parametrů `noise`, `livingReward` a `gamma` zjištěno, že `ValueIterationAgent` dokázal přejít most do lépe ohodnoceného terminálního stavu gridworld problému `BridgeGrid` 5.2 pouze, když se stochastičnost akcí (`noise`) potlačila na hodnotu 0.01 a méně a koeficient odměny `gamma` zvedl na hodnotu 0.8 a výše a `livingReward` nastavil na hodnotu alespoň -0.2, aby agent na mostě co nejdéle zůstával. Zajímavým zjištěním také bylo, že pokud se agentovi nastavil parametr stochastičnosti `noise` na větší hodnotu (např. 0.8), výsledná strategie všech neterminálních částí mostu raději ukazovala sebevražedně do útesu. Je tedy vhodnější spíše omezovat tento parametr, aby byly výsledky alespoň trochu průkazné.

Názornějším výchozím bodem pro výpočet strategie jsou 4 Q-learningové Q-hodnoty pro každý stav, neboť reflektují průměrný užitek každé akce stavu a navíc z nich takto lze jednodušeji získat optimální strategii. Bylo též ověřeno, že metoda Q-Learning je schopna postupně navyšovat odhady Q-hodnot svých optimálních akcí i přesto, že agent právě riskoval a nepodstoupil optimální akci (skočil z útesu, narazil do stěny atp.). Překvapením bylo, že `QLearningAgent` potřeboval pro `BridgeGrid` (s podobným nastavením paremetrů jako předchozí agent a `epsilon` = 0.9) minimálně cca 1000 epizod tréninku. Výsledné Q-hodnoty jsou vidět na obrázku C.4 (strategie pro stavy jsou dány nejvyšší Q-hodnotou stavu, proto není nutné je uvádět).

Problém `DiscountGrid` se ukázal jako nejprůkaznější z těchto třech gridworld problémů. Postupně bylo zjištěno, že:

- Záporná hodnota `livingReward` přinutí agenta víc riskovat a pokoušet se o přejití kratší cesty podél útesu, aby rychle ukončil epizodu (preferuje kratší cestu). Naopak kladná hodnota podněcuje agenta k podnikání spíše bezpečnější alternativy – vyhnutí se útesu, potažmo až k nekonečnému průchodu deskou (dokud není náhodně poslán do terminálního stavu).

- Pokud se zvýší koeficient `gamma` (např. na hodnotu 0.9), agent není tolik postižen snižováním odměn a preferuje delší cesty k terminálnímu uzlům a naopak.
- Stochastičnost `noise` často zabraňuje nalezení optimální strategie, avšak zvýšení tohoto parametru alespoň na 20 % šanci náhodné akce pomáhá agentům kvalitněji sestavit V a Q-hodnoty.

Až tento experiment definitivně prokázal, jak klíčové jsou testované parametry pro výslednou strategii agenta `ValueIterationAgent`. Příklad strategie, kdy agent preferuje riskovat kratší cestu podél útesu ke vzdálenejšímu, ale hodnotnějšímu terminálnímu uzlu, je vidět na obrázku C.5. Opět se také projevilo, že `QLearningAgent` potřeboval vždy stejně nebo více epizod tréninku. Navíc se výsledná strategie daných parametrů často neslučovala se strategií `ValueIterationAgent`. Stejné parametry, ale rozdílná průběžná strategie Q-learningu lze vidět na obrázku C.6.

Experimenty gridworld světa překvapivě ukázaly, že `ValueIterationAgent` dokáže vypočíst své hodnoty za méně epizod oproti Q-Learningu. Gridworld problémy jsou však jednoduché modely světa a pro reálné problémy, kdy obvykle není k dispozici model, se musí vycházet z pokročilejších metod strojového učení počínaje algoritmem Q-learning. Nejdůležitější je fakt, že tento algoritmus vypočítává hodnoty svých akcí optimálně a to i v případech, kdy Value Iteration nemusí přijít na výsledné optimální chování. Bohužel jeho exponenciálně rostoucí výpočetní náročnost se jeví jako největší problém.

Kapitola 7

Závěr

Cílem této bakalářské práce bylo navrhnout dostatečne chytrou umělou inteligenci, která by si dokázala poradit i s tak komplexním problém jako je hra Ms. Pacman. Nejdříve byla nastudována problematika metod hraní her a techniky strojového učení ve vztahu k učícímu agentovi. Nastudování teorie pomohlo především k definici a klasifikaci samotné hry jako rozhodovací problém umělé inteligence a následnému stanovení návrhu řešení problematiky metodami Q-Learning a Aproximační Q-Learning. Tyto metody byly vybrány jako jeden ze způsobů, jak dosáhnout určité inteligence ve složitém stochastickém prostředí. Pro agenta algoritmu Aproximační Q-Learning byl dále sestaven vektor vlastností, který aproximačně popisuje dění na ploše.

Pro práci byli také navrhnuti a implementováni základní agenti, kteří byli srovnáni s učícími agenty. Srovnávací experimenty brali v potaz celkem 5 agentů na 3 různých mapách Ms. Pacman. Efektivita učících se agentů byla takto srovnána například s agentem implementujícím metodu Expectimax jak z hlediska výpočetní náročnosti, tak z hlediska míry dosažené inteligence agenta. Navíc byly experimentálně zkoumány parametry ovlivňující chování agentů na 3 průkaznějších gridworld modelech než je Ms. Pacman. Na základě těchto experimentů bylo zjištěno, že výsledná optimální strategie agenta se může naprosto lišit pod vlivem daného parametru. Například pokud byl agent silně bodově postižen za každý svůj krok, snažil se rychle ukončit hru, aby si uchoval co největší skóre. Až za pomocí změny parametrů se podařilo přejít pomyslný most do vzdálenějšího hodnotnějšího terminálního stavu jednoho z gridworld problémů. Experimentální část práce též poukázala na rozdíly aktivně učícího Q-learningové agenta a pasivního agenta algoritmu Value Iteration ve smyslu vlivu jejich Q a V-hodnot na výslednou strategii stavu.

Experimenty potvrdily, že především Aproximační Q-learning pomohl výrazně eliminovat problém velké vypočetní náročnosti stavového prostoru hry. Agentovi stačilo pouhých 25 epizod tréninku, aby se naučil úspěšně zdolat menší mapu. U menších map se však projevilo, že pro řešení tak malého stavového prostoru bohatě postačil algoritmus Expectimax. Dosažení úspěšné strategie totiž Q-learningovému agentovi trvalo cca 1300 epizod tréninku, což ve srovnání s ihned nabytou optimální strategií Expectimaxu působí počtem epizod poměrně nadbytečně. Hlavním cílem agenta Ms. Pacman však bylo zdolat **mapu větší**. Až zátěžové testy ukázaly největší přednost algoritmu Aproximační Q-Learning. Nejen, že se nezvedla vypočetní náročnost, ale stačilo mu pouze cca 100 epizod tréninku, aby vyhrál v minimálně nadpoloviční většině případů. Základní agent Expectimaxu si navíc příliš nedokázal poradit s takto náročnou mapou především vlivem své nedokonalé ohodnocovací funkce, kterou by stálo za to do budoucna vylepšit, aby dokázala reflektovat i agentovu pozici v rozích, kdy je nejvíce ohrožen nepřáteli apod. Nejen proto lze považovat imple-

mentaci vektoru vlastností aproximující dění na ploše za úspěšnou a výslednou inteligenci učícího agenta částečně přirovnat k lidské. Problémem však stále zůstává zmíněná stagnace dosažených výsledků vektoru, které se s počtem iterací příliš nezvedly. Agent dokázal rychle nabýt vysoké úrovně inteligence a již se nepříliš zlepšoval. Rozhodně má tedy jeho vektor vlastností spoustu prostoru na vylepšení do budoucna a to tak, aby se dosažené skóre postupně zvedalo tak jako u Q-Learningový agenta na malých mapách. Problém agenta se dá přirovnat i k ohodnocovací funkci základních agentů, která nedokázala správně ohodnotit některé dění na herní desce následníka stavu. Do budoucna by se práce dala přepsat do neinterpretovaného jazyka (např. Java, C++), který by tak pravděpodobně umožnil snížit časovou a výpočetní náročnost celé aplikace a udělat ji tak přenositelnější i pro méně výkonné stroje i při více epizodách tréninku učícího agenta a větších mapách.

Literatura

[1] AlphaGo, The first computer program to ever beat a professional player at the game Go. [online]. 2016-03-15 [cit. 2016-05-12].
URL https://deepmind.com/alpha-go

[2] BUSONIU, L.; BABUŠKA, R.; SCHUTTER, B. D.; aj.: *Reinforcement learning and dynamic programming using function approximators*. Boca Raton, FL: CRC Press, c2010, ISBN 978-1-439-82108-4.

[3] DENERO, J.; KLEIN, D.; MILLER, B.; aj.: UC Berkeley CS188 Intro to AI Course Material [online]. 2016-04-20 [cit. 2016-04-25].
URL http://ai.berkeley.edu

[4] GOSAVI, A.: Reinforcement Learning: A Tutorial Survey and Recent Advances. *INFORMS Journal on Computing*, ročník 21, č. 2, 2009, ISSN 1091-9856, doi:10.1287/ijoc.1080.0305.
URL http://pubsonline.informs.org/doi/10.1287/ijoc.1080.0305

[5] KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W.: Reinforcement Learning: A Survey. *Journal of Artifcial Intelligence Research*, ročník 4, č. 1, 1996: s. 237–285, ISSN 1076-9757.
URL http://www.jair.org/media/301/live-301-1562-jair.pdf

[6] KLEIN, D.; ABBEEL, P.; col.: BerkeleyX: CS188x_1 Artificial Intelligence [online]. 2016-01-15 [cit. 2016-01-17].
URL https://courses.edx.org/courses/BerkeleyX/CS188x_1/1T2013/

[7] MAIMON, O.; COHEN, S.: *A Review of Reinforcement Learning Methods. Data Mining and Knowledge Discovery Handbook*. [online], Boston, MA: Springer US, 2010, doi:10.1007/978-0-387-09823-4_20, ISBN 978-0-387-09822-7.
URL http://link.springer.com/10.1007/978-0-387-09823-4_20

[8] MAŘÍK, V.; ŠTĚPÁNKOVÁ, O.; LAŽANSKÝ, J.; aj.: *Umělá inteligence (1)*. Praha: Academia, 1993, ISBN 80-200-0496-3.

[9] MAŘÍK, V.; ŠTĚPÁNKOVÁ, O.; LAŽANSKÝ, J.; aj.: *Umělá inteligence (3)*. Praha: Academia, 2001, ISBN 80-200-0472-6.

[10] PANDEY, P.; PANDEY, D.; KUMAR, D. S.: Reinforcement Learning by Comparing Immediate Reward. ročník 8, č. 5, 2010, ISSN 1573-0565.
URL http://arxiv.org/pdf/1009.2566.pdf

[11] PATEL, A.: Heuristics for grid-maps (Game Programming) [online]. 2016-03-28 [cit. 2016-04-25].
URL http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#heuristics-for-grid-maps

[12] POOLE, D.; MACKWORTH, A.: *Artificial Intelligence: Foundations of Computational Agents [online]*. Cambridge: Cambridge University Press, 2010-06-01 [cit. 2016-04-25], ISBN 978-0-521-51900-7.
URL http://artint.info/html/ArtInt.html

[13] SHOHAM, Y.; LEYTON-BROWN, K.: *MULTIAGENT SYSTEMS: Algorithmic, Game Theoretic, and Logical Foundations*. New York: Cambridge University Press, 2008, ISBN 978-0-521-89943-7.

[14] SUTTON, R. S.; BARTO, A. G.: *Reinforcement Learning: An Introduction*. Cambridge, Mass.: MIT Press, c1998, ISBN 978-0-262-19398-6.

[15] SVENSSON, J.; JOHANSSON, S. J.: Influence Map-based controllers for Ms. PacMan and the ghosts. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, [online], IEEE, Sept 2012, ISSN 2325-4270, s. 257–264, doi:10.1109/CIG.2012.6374164, ISBN 978-1-4673-1194-6.
URL
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6374164

[16] WATKINS, C. J.; DAYAN, P.: Technical Note: Q-Learning. *Springer: Machine Learning*, ročník 8, č. 3/4, 1992: s. 279–292, ISSN 0885-6125, doi:10.1023/A:1022676722315.
URL http://dx.doi.org/10.1023/A:1022676722315

Přílohy

Seznam příloh

A Obsah CD **50**

B Manuál **51**
B.1 Ms. Pacman demo (`pacman.py`) 51
B.1.1 Příklady použití 51
B.2 Gridworld problémy (`gridworld.py`) 52
B.2.1 Příklady použití 53

C Přidatné obrázky k experimentům **54**

D Vlastní implementace **57**

Příloha A

Obsah CD

Praktická část práce je uložena na CD. Obsah CD a jeho popis:

- `doc` - dokumentace k práci `IBP_MsPacman_Blozonova.pdf`
- `src` - zdrojové kódy frameworku (včetně složky `layouts` s mapami pro `pacman.py`) s implementacemi agentů a vektoru vlastností v souborech:
  - `multiAgents.py` - implementace chování základních agentů
  - `valueiterationAgents.py` - implementace chování `ValueIterationAgent`
  - `qlearningAgents.py` - implementace chování `QLearningAgent`, `PacmanQAgent` a `ApproximateQAgent`
  - `featureExtractors.py` - implementace vektoru vlastností `BetterExtractor`.

Příloha B

Manuál

B.1 Ms. Pacman demo (`pacman.py`)

Pro demo Ms. Pacman byly používány základní parametry:

- `-h` pro výpis celé nápovědy a všech možných parametrů
- `-m` manuální režim (vyzkoušení grafického rozhraní dema)
- `-frameTime 0` bez animace
- `-q` minimalistický režim bez grafického rozhraní
- `-t` textový režim bez grafického rozhraní
- `-f` fixed random seed (pro fixní modelaci náhodnosti scénářů)
- `-z 0.7` míra zvětšení grafiky (*zoom*)
- `-n 2010` počet epizod celkově
- `-x 2000` počet tréninkových epizod
- `-l smallClassic` druh mapy (ze složky `src/layouts`) - např. `smallClassic`, `mediumClassic`, `minimaxClassic`, `openClassic`,...
- `-p ApproximateQAgent` druh agenta - např. `ReflexAgent`, `ExpectimaxAgent`, `PacmanQAgent`, `ApproximateQAgent`,...
- `-a depth=3,alpha=0.7,epsilon=0.5,discount=0.9` agentovy další parametry, např. hloubka **depth**, gamma **discount**

B.1.1 Příklady použití

`ReflexAgent`, 2 nepřátelé, defaultní mapa

```
python pacman.py -p ReflexAgent -k 2
```

`ExpectimaxAgent`, 5 her se základním výpisem bez GUI, hloubka 3

```
python pacman.py -p ExpectimaxAgent -l smallClassic -a depth=3, -n 5 -q
```

`PacmanQAgent`, 2000 epizod tréninku, 10 zobrazených epizod provádění optimální strategie, na mapě `smallGrid`

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

`PacmanQAgent`, 10 epizod tréninku na mapě `smallGrid`

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a
numTraining=10
```

`ApproximateQAgent` 100 epizod z nichž 90 trénink (10 zobrazených epizod provádění optimální strategie), na mapě `mediumClassic`, použitý lepší Extractor vlastností

```
python pacman.py -p ApproximateQAgent -a extractor=BetterExtractor
-n 100 -x 90 -l mediumClassic
```

B.2 Gridworld problémy (`gridworld.py`)

Pro vizualizaci hodnot pokročilých agentů - především agenta `ValueIterationAgent` byly používány základní parametry včetně příkladů hodnot:

- `-h` pro výpis celé nápovědy a všech možných parametrů
- `-q` minimalistický režim bez grafického rozhraní
- `-t` textový režim bez grafického rozhraní
- `-w 120` pixelový rozměr políčka mřížky (lepší nastavit například na 120 px, jinak je mřížka příliš velká)
- `-k 10` počet epizod
- `-i 100` počet iterací V-hodnot
- `-v` zobrazení hodnot po každé iteraci
- `-g BookGrid` druh gridworld problému - např. `BookGrid`, `BridgeGrid`, `DiscountGrid`
- `-a value` druh agenta - `value` pro Value Iteration, `q` pro Q-Learning (pokud se nespecifikuje, spouští se manuální ovládání)
- `-d 0.8` velikost koeficientu gamma
- `-r 0.9` **livingReward**, odměna přechodu ze stavu do neterminálního následníka
- `-n 0.2` **noise**, pravděpodobnost přechodu
- `-e 0.5` **epsilon** (epsilon-greedy), náhodnost akcí při tréninku
- `-l 0.7` **alpha** (learning rate), rychlost TD učení

B.2.1 Příklady použití

`ValueIterationAgent`, defaultní mapa, 100 iterací, 10 epizod celkem

```
python gridworld.py -a value -i 100 -k 10
```

`ValueIterationAgent`, typ gridworld problému `DiscountGrid`, 5 iterací, 10 epizod (výchozí)

```
python gridworld.py -a value -i 5 -g DiscountGrid
```

`ValueIterationAgent`, překonání gridworld problému `BridgeGrid`, 1000 iterací, `gamma` = 0.9, `noise` = 0.1

```
python gridworld.py -a value -i 1000 -g BridgeGrid --discount 0.9
--noise 0.01
```

`QLearningAgent`, 5 epizod celkově, manuálně

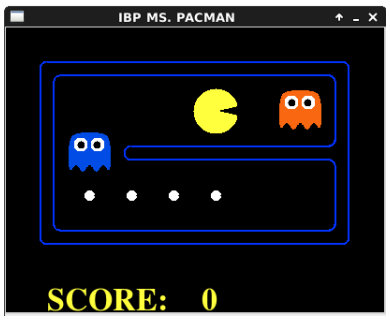
```
python gridworld.py -a q -k 5 -m
```

`QLearningAgent`, 20 epizod na problému `DiscountGrid`, `alpha` = 0.8, `epsilon` = 0.5

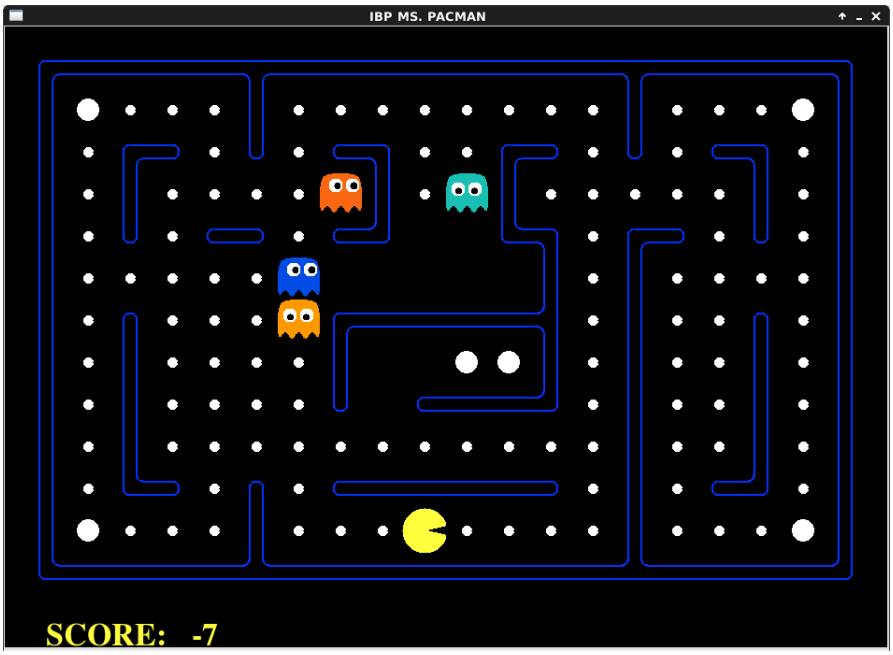
```
python gridworld.py -a q -k 20 -g DiscountGrid -l 0.8
--epsilon 0.5
```

Příloha C

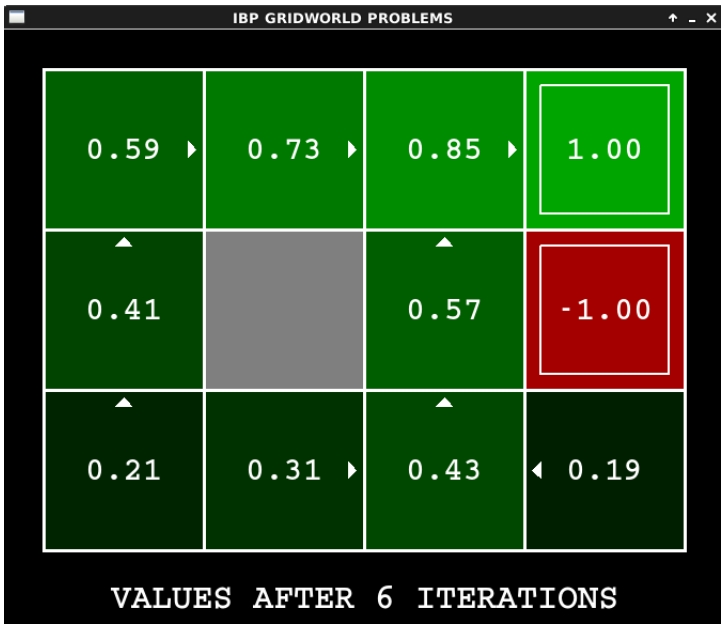
Přidatné obrázky k experimentům



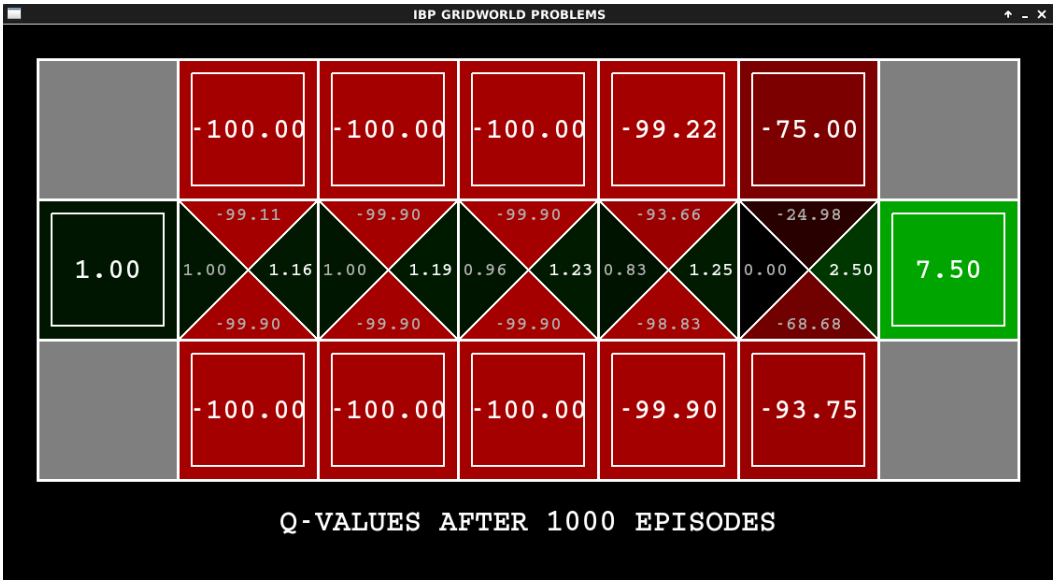
Obrázek C.1: Screenshot mapy `trappedClassic`.



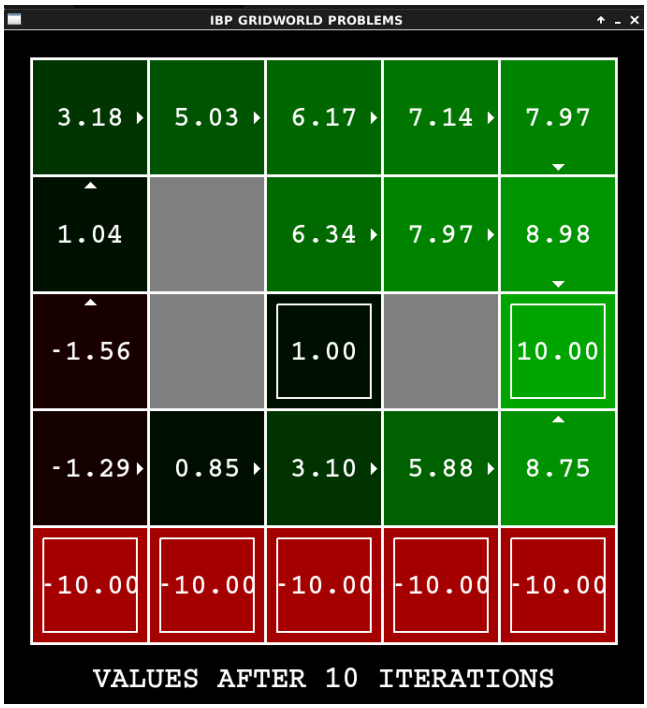
Obrázek C.2: Screenshot zatěžkávací mapy `trickyClassic`.



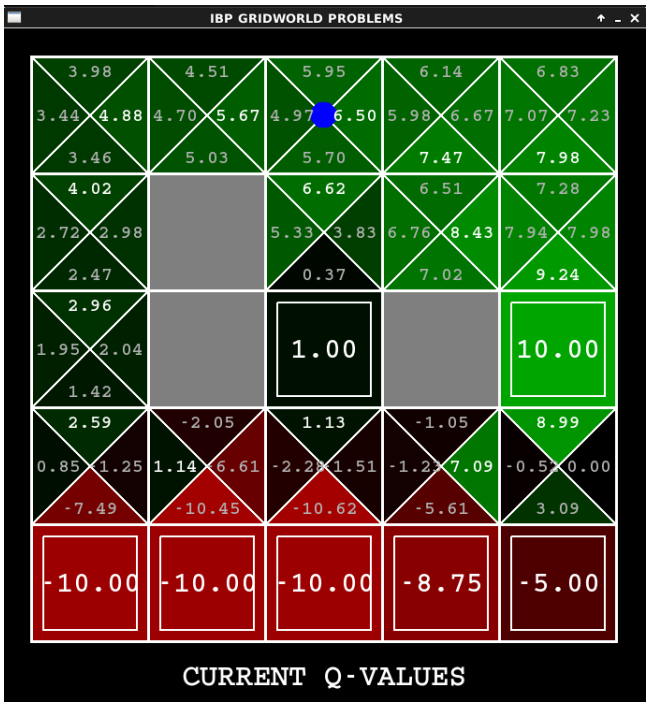
Obrázek C.3: Screenshot 6 iterací Value Iteration pro gridworld problém `BookGrid`.



Obrázek C.4: Screenshot Q-hodnot agenta `QLearningAgent` po 1000 epizodách na `BridgeGrid`.



Obrázek C.5: Screenshot V-hodnot agenta `ValueIterationAgent` na `DiscountGrid`. Agent zde preferuje vzdálenější terminální uzel kratší cestou, ačkoliv riskuje pád z útesu.



Obrázek C.6: Screenshot průběžných Q-hodnot agenta `QLearningAgent` na `DiscountGrid`. Agent zde preferuje vzdálenější terminální uzel a delší bezpečnější cestu.

Příloha D

Vlastní implementace

Ve složce zdrojových kódů `src` byly v objektové kostře frameworku pozměněny následující soubory:

- `pacman.py` - změny výpisu (průměrná doba hry),
- `game.py` - proměnná `totalGameTime` pro celkovou dobu spůštěných her,
- `multiAgents.py` - implementace chování agentů: `ReflexAgent`, `MinimaxAgent`, `AlphaBetaAgent`, `ExpectimaxAgent`,
- `valueiterationAgents.py` - implementace chování `ValueIterationAgent`,
- `qlearningAgents.py` - implementace chování `QLearningAgent` a `ApproximateQAgent` (`PacmanQAgent` pouze nastavuje hodnoty vstupních parametrů pro demo Ms. Pacman; implementováno frameworkem),
- `featureExtractors.py` - implementace vektoru vlastností `BetterExtractor`.

Tabulka D.1 udává čísla řádků vlastní implementace každého souboru. Středník mezi čísly řádků rozděluje třídy jednotlivých agentů (pokud jsou implementováni v daném souboru).

Tabulka D.1: Rozsah vlastní implementace

soubor	řádky
`pacman.py`	279, 297, 298, 301, 305, 306, 678, 679
`game.py`	532
`multiAgents.py`	83-131; 157-215, 231-234; 240-314, 319-322; 329-401, 407-410, 422-481
`valueiterationAgents.py`	80-100, 114-124, 146-158
`qlearningAgents.py`	82, 86, 94-106, 116-130, 140-149, 159, 160; 202-207, 212-216, 226
`featureExtractors.py`	117-160

Každý soubor obsahuje prvotní hlavičku UC Berkeley a vlastní komentářovou hlavičku definující implementaci daného agenta atp. Pokud je to v daném souboru vhodné, hlavička dále pokračuje vlastním popisem příkladů využití daného agenta pro daný problém. Ukázka začátku hlavičky pro základní agenty:

```
# BP: basic agents implemented here
```