BAKALÁŘSKÁ PRÁCE

Jméno Příjmení

Název práce

Název katedry nebo ústavu

Vedoucí bakalářské práce: Vedoucí práce
Studijní program: studijní program
Studijní obor: studijní obor

Praha ROK

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Poděkování.

Název práce: Název práce

Autor: Jméno Příjmení

Katedra: Název katedry nebo ústavu

Vedoucí bakalářské práce: Vedoucí práce, katedra

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: Name of thesis

Author: Jméno Příjmení

Department: Name of the department

Supervisor: Vedoucí práce, department

Abstract: Abstract.

Keywords: key words

Obsah

Úvod

Počítačové hry mají kromě zábavního prožitku ze samotného hraní další funkci. Herní prostředí často tvoří samostatný, uzavřený svět se svými vlastními zákonitostmi. Agent ve světě dané hry ví, v jakém stavu se nachází, má na výběr konečný počet akcí, které může provést a cíl, kterého chce dosáhnout. A právě takovéto prostředí je pro nás vhodné pro experimentování s umělou inteligencí.

1. Navržená hra - Asteroidy

1.1 Herní logika

Jedná se o hru dvou hráčů. Každý z hráčů ovládá svou vesmírnou loď. Prostředí hry má představovat vesmírný prostor, je to ale prostor zjednodušený, proto zde neplatí gravitační ani odporové síly. To má tedy za následek, že když se vesmírná loď rozletí v nějakém směru, tak v tomto směru letí i nadále i bez dalšího akcelerování.

Vesmírný prostor je v této hře nekonečný a dalo by se říct jistým způsobem cyklický, pokud vesmírná loď proletí dolní hranicí herního prostoru, tak nezmizí, ani nenaboural, ale objeví se na stejné pozici jen na horní hranici a obráceně. Analogicky to platí i s bočními hranicemi. Vesmírné lodě sebou mohou proletět a nedojde ke srážce. Nejsou zde žádné statické překážky, kterým by bylo třeba se vyhnout. Co se ale může srazit s vesmírnou lodí jsou asteroidy.

Asteroidy vznikají v průběhu hry na náhodných místech a letí náhodným směrem. Nové asteroidy se generují častěji, čím déle hra trvá. V boji s asteroidy má hráč v zásadě dvě možnosti. Buď se může pokusit danému asteroidu vyhnout, tím že s lodí pohne mimo trajektorii asteroidu, anebo může asteroid sestřelit. Každý hráč má omezený počet životů a každá srážka lodě s asteroidem ubere hráči část jeho životů.

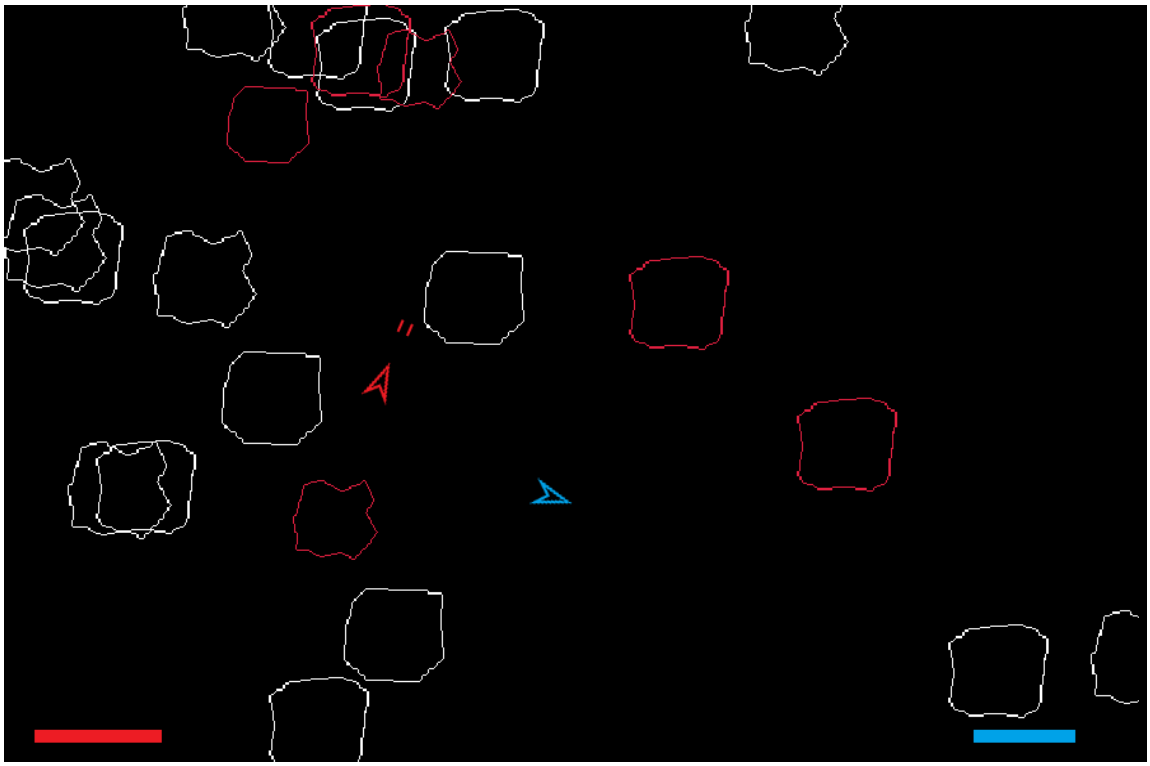
Hráč má k dispozici dva typy střel, obyčejnou a rozdvojovací. Vystřelená střela má značně vyšší rychlost než vesmírné lodě i než kolem letící asteroidy. Střely nejsou určeny k přímému zasažení lodě protihráče, vesmírné lodě jsou k nepřátelské střele imunní. Střely jsou určený k sestřelování letících asteroidů. Asteroidy mohou mít tři velikosti. Náhodně vytvořený asteroid je vždy největší. Každým rozstřelením daného asteroidu vznikají asteroidy o stupeň menší velikosti. Nově vytvořené asteroidy vznikají na místě původního asteroidu. Asteroid se může rozstřelit různými způsoby. Zde záleží na tom, jakou střelou byl asteroid setřelen.

V případě střely obyčejné vznikne namísto původního asteroidu jeden menší, který letí stejným směrem jako střela, která ho zasáhla. V případě střely rozdvojovací se původní asteroid rozstřelí na dva menší, kde každý z nich je oproti směru střely vychýlen o 15°po a proti směru hodinnových ručiček. Pokud je zasažen asteroid nejmenší velikosti, tak již žádné další asteroidy nevznikají. Rychlost asteroidů je nepřímo závislá na jejich velikosti, čím je asteroid menší, tim vyšší rychlost má.

Asteroidy vzniklé rozstřelením se stávají projektily daného hráče. Hráč nemůže být zasažen asteroidem, který sám vytvořil

1.2 Cíl hry

Během hry vzniká postupně více a více asteroidů, čímž je postupně stále obtížnější se všem asteroidům vyhnout, nebo je sestřelit. Hráč nemůže zranit nepřítele střelou přímo, může se ale snažit rozstřelit nějaký z kolem letících asteroidů tak, aby pomocí nově vzniklých asteroidů trefil nepřítele. Cílem hráče je ovládat svou loď takovým způsobem, aby vydržel ve hře déle. Hra končí a hráč vítězí, když nepříteli nezbydou žádné další životy.



Obrázek 1.1: Screenshot ze hry

2. Architektura hry

2.1 Vesmírné objekty

Všechny vesmírné objekty mají některá data společná. Každý vesmírný objekt má souřadnice své současné polohy a také vektor rychlosti.

2.1.1 Asteroidy

Asteroidy nesou navíc informace o tom, jaké jsou velikosti a zda-li byly vytvořené nějakým z hráčů. Na základě těchto dvou informací je asteroidu při vytvoření přiřazen obrázek, pomocí kterého je po dobu své existence vykreslován.

2.1.2 Střely

Vystřelené střely neletí věčně, ale mají omezenou životnost kolik kroků hry budou existovat. Tato hodnota se nastavuje z konfiguračního souboru z položky *BULLET_LIFE_COUNT*. V každém kroku hry se střele její živostnost sníží o jedna a pokud se dostane na nulu, tak střela bude zničena. Střele se při vytvoření nastaví úhel, pod kterým poletí. Tento úhel je roven úhlu natočení vesmírné lodi, který měla při vystřelení. Samozřejmě také u střely musíme evidovat, kterému z hráčů patří, toto je řešeno odkazem na objekt vesmírné lodi, která střelu vystřelila. Jak již bylo zmíněno v předchozí kapitole, střely jsou dvojího druhu. Příznakem *split* se určuje zda se jedná o střelu obyčenou nebo rozdvojovací

2.1.3 Vesmírná loď

Vesmírná loď má základní polohové informace rozšířené o úhel. Ten se s každou rotací lodě zvětší nebo zmenší o 12°. Akcelerace funguje vektorovým sčítáním. K současnému vektoru rychlosti se přičte vektor odpovídající současnému úholu lodi. Maximální rychlost vesmírné lodi je omezená, v případě že akcelerací vznikne vektor rychlosti, jehož délka je větší než hodnota maximální rychlosti, se směr vektoru zachová, ale požadovaně se zkrátí.

2.2 Prostředí

Hra běží v cyklu diskrétních kroků, které dohromady simulují plynulý pohyb hry. Herní prostředí je insipirováno projektem open ai gym od google (viz `https://gym.openai.com/`). Jedním rozdílem je však přístup k vykreslování hry. V případě *gym.openai* se prostředí vykresluje zavoláním metody *render()* na instanci prostředí zvenku. Já jsem zvolil přístup jiný. V případě, že chceme hru graficky zobrazovat, předáváme v kontruktoru prostředí grafický modul, který implementuje vykreslování jednotlivých typů vesmírných objektů. A prostředí už poté objekty graficky vyresluje interně samo. Rozhodnutí, že se má grafický modul volitelně injektovat v konstruktoru a nemá být natvrdo svázán s prostředím, jsem učinil pro větší nezávislost modulů. Při práci s různými knihovnami pro evoluční algoritmy se ukázalo být problematické, kdyby bylo herní prostředí svázáno s grafickým modulem.

Herní prostředí se stará o manipulaci všech vesmírných objektů a akcí s nimi spojenými. V každém kroku dostává od hráčů akce, které chtějí provést, a prostředí na to odpovídajícím způsobem reaguje. Akce každého hráče z hráčů jsou pole, které obsahuje elementární možné akce:

- Rotace vlevo
- Rotace vpravo
- Akcelerace
- Obyčejná střela
- Rozdvojovací střela

Hráč může provádět více akcí najednou. Na základě přítomných elementárních akcí se provadí dané reakce. Prostředí se stará o vesmírné objekty přímo. V případě elementárních akcí, které mění rychlost nebo orientaci vesmírné lodi, prostředí zavolá funkce, které požadované změny na vesmírné lodi provede. A v případě elemntárních akcí střel se na základě polohy a orientace dané vesmírné lodi vytvoří nová střela, kterou opět bude mít ve správě právě prostředí.

Hra, jak již bylo řečeno, má být konečná, toho je docíleno narůstajícím počtem asteroidů. O to se také stará herní prostředí. Pamatuje si počet kroků, které uběhly od posledního vytvořeného asteroidu. Pokud tento počet překročil danou mez, tak prostředí vytvoří nový asteroid. Postupného nárůstu nových asteroidů je docíleno incrementálním snižováním této meze.

Další důležitou funkcí herního prostředí je kontrola srážek. Všechny objekty jsou prostorově reprezentovány jako kruhy s danými poloměry. Postupně se prochází všechny objekty, u kterých nás zajímají srážky, a Euklidovskou metrikou se kontroluje, zda od sebe nejsou vzdáleny méně než je součet jejich poloměrů. V případě srážky se prostředí postará o správnou reakci - zničení nebo změnu sražených objektů a případně vytvoření nových objektů vzniklých srážkou.

V rámci srážek se upravují také odměny jednotlivých hráčů. Odměna je hodnota, která vyjadřuje jak úspěšný byl tento krok pro každého z hráčů. V každém kroku, který hráč přežil, je hráč odměnen odměnou hodnoty 1. Jsou ale konkrétní srážky objektů, které hodnoty odměny mohou změnit. V případě, že hráč sestřelil nepřátelský asteroid, nebo svým asteroidem srazil nepřátelovu loď, se výše odměny

zvýší. Naopak, pokud byla jeho vesmírná loď zasažena nepřátelským asteroidem, je hodnota odměny snížena. Koncept odměn nijak neovlivňuje samotný běh hry, ale bude se nám hodit v dalších kapitolách v umělé inteligenci.

Nezmínili jsme zatím pohyb objektů. I o to se samozřejmě stará herní prostředí. Zde se prochází seznamy všech vesmírných objektů a jednoduše se k jejich současné poloze přičte vektor rychlosti. Jediné co se musí kontrolovat je, že se daný objekt nedostal mimo herní prostor. Pokud toto nastane, tak je vrácen zpět do prostoru na své odpovídající místo (viz 1.1)

Pokud byl při vytváření prostředí předán grafický modul, tak se prostředí postará i o vykreslení vesmírných objektů. Pro všechny vesmírné objekty se zavolá příslušná metoda pro jejich vykreslení. Způsob implementace vykreslení jednotlivých objektů není zodpovědností herního prostředí.

Poslední věcí, kterou má prostředí ještě na starosti je kontrola, zda hra neskončila. Na konci kroku se zkontroluje, zda mají oba hráči kladný počet životů.

Jedna instance prostředí odpovídá jedné hře. Herní prostředí má dvě základní metody pro řízení hry.
Metoda *reset()* inicializuje hru do počátečního stavu a tento stav vrátí. Tato metoda se musí zavolat před začátkem hry.
A druhá metoda *next_step(actions_one, actions_two)*, která na na základě akcí hráčů, převede hru do následného stavu. Právě v této metodě je schovaná celá logika manipulace s vesmírnými objekty popsána výše.

```
def next_step(self, actions_one, actions_two):
    self.step_count = self.step_count + 1
    self.reward_one = 0
    self.reward_two = 0

    self.handle_actions(actions_one, actions_two)
    self.generate_asteroid()
    self.check_collisions()
    self.move_objects()
    if self.draw_modul is not None:
        self.render()

    (game_over, player_one_won) = self.check_end()
    if not game_over:
        self.reward_one += 1
        self.reward_two += 1

    current_state = State(self.asteroids_neutral,
                          self.rocket_one,
                          self.asteroids_one,
                          self.bullets_one,
                          self.rocket_two,
                          self.asteroids_two,
                          self.bullets_two)

    return self.step_count, \
           (game_over, player_one_won), \
           current_state, \
           (self.reward_one, self.reward_two)
```

2.2.1 Stav prostředí

Herní prostředí vrací po každém kroku současný stav hry. Stav hry se skládá ze seznamů všech vesmírných objektů včetně kompletních informací o nich. Pravděpodobně by bylo možné vracet i méně obsáhlou informaci o současném stavu hry. Avšak pro mě bylo motivací předat kompletní informaci o všech objektech a nechat následně až na agentech, na základě čeho všeho se budou chtít rozhodnout, jaké akce chtějí provést. Mou snahou bylo, aby herní prostředí poskytnulo všechny informace a neomezovalo tím agenty.

2.3 Agent

Agent je ústřední postavou celé hry a obzvláště v dalších kapitolách pro nás bude nejzajímavějším předmětem zájmu. Je to právě zde, kde budeme později mluvit o umělé inteligenci. V případě agenta, který je ovládán lidským hráčem, se agent, respektive člověk, který jej ovládá, neřídí datovou reprezentací stavu, tak jak jej obdržel od herního prostředí. Ale rozhoduje se na základě toho, jak hráč vizuálně vnímá co se děje ve hře. A příkaz k provedení jednotlivých akcí udává ovládáním kláves na klávesnici. Lidský hráč pro nás ale nebude tolik zajímavý k experimentování, my se budeme soustředit primárně na strojové agenty.

Každý agent musí implementovat jedinou metodu *choose_actions(state)*. Úkolem agenta je, na základě obdrženého stavu, zvolit akci, kterou chce provést. A právě tento rozhodovací problém pro nás bude půdou pro experimentování s různými abstrakcemi a přístupy umělé inteligence.

2.4 Grafické prostředí

Pro grafické zobrazování hry jsem zvolil python knihovnu pygame (`https://www.pygame.org/news`), která jak sám název napovídá slouží k programování jednodušších her v pythnu. Tato knihovna nabízí kromě grafických funkcí, také podporu pro manipulaci s herními objekty. Například je zde zabudovaná podpora pro kontrolu srážek. Mou snahou bylo této funkcionality využít, ale později se to ukázalo být nevhodné. Prvním problémem bylo, že herní objekty jsou v rámci této knihovny reprezentovány jako čtverce a kontrola srážek je tedy realizována jako dotaz, zda se dva čtverce pronikají. Kontrola zda se dva čtverce pronikají je výpočetně náročnější než průnik dvou kruhů. Jako větší problém se ale ukázala být integrace pygame modulu do herního prostředí. Při pokusu o paralelizaci více běhů her se objevily technické problémy, které se mi nepodařilo vyřešit. Proto jsem se rozhodl, že si kontrolu srážek implementuju separátně, nezávisle na modulu pygame a modul budu využívat pouze pro vykreslování hry. A pro toto použití je pro mě knihovna pygame zcela dostačující. Vytvořil jsem si sadu obrázků reprezentující jednotlivé vesmírné objekty. A knihovna pygame mi je umožňuje vykreslovat potřebným způsobem.

2.5 Hlavní herní cyklus

Vysvětlili jsme tedy všechny základní prvky, které v této hře potřebujeme a nyní je propojíme dohromady. K běhu hry potřebujeme mít instanci herního prostředí, tomu můžeme předat grafický modul, pokud chceme graficky vykreslovat, anebo žádnou implementaci nedodáme a pak hra bude běžet bez obrazu, tohoto budeme naším hlavním zájmem později v rámci trénování umělé inteligence. Dále potřebujeme inicializovat dva agenty, kteří budou poředstavovat naše hráče. A po inicializaci herního prostředí budeme v cyklu simulovat hru, dokud prostředí neoznámmí, že daným krokem hra skončila. V každém kroku cyklu agenti na základě stavu, ve kterém se herní prostředí nachází, zvolí své akce a ty předají zpět hernímu prostředí. Kostra herní simulace tedy vypadá následovně:

```
env = Enviroment(draw_module())
agent_one = Some_agent(1)
agent_two = Some_agent(2)


state = env.reset()
game_over = False
while not game_over:
    actions_one = agent_one.choose_actions(state)
    actions_two = agent_two.choose_actions(state)

    _, (game_over, player_one_won), state, _ = \
        env.next_step(actions_one, actions_two)
```

3. Senzory a akční plány

3.1 Motivace

V této kapitole se již přesouváme od hry samotné ke způsobu, jak na daný stav hry nahlížet chytřeji a také, jak na něj chytřeji reagovat. Na nejnižší úrovni dostávají agenti od prostředí stav, který obsahuje seznamy všech vesmírných objektů a agenti na něj mají reagovat nějakou elementární akcí. Bylo by proto dobré vymyslet princip, jak z obsáhlých a detailních informací nízké úrovně získávat menší objemy zajímavějších informací vyšší úrovně. A podobně by se nám mohlo hodit místo elementárních akcí nízké úrovně, vymyslet princip jak volit akce tak, aby vedly k akcím vyšší úrovně. A právě tyto abstrakce realizujeme pomocí senzorů a akčních plánů.

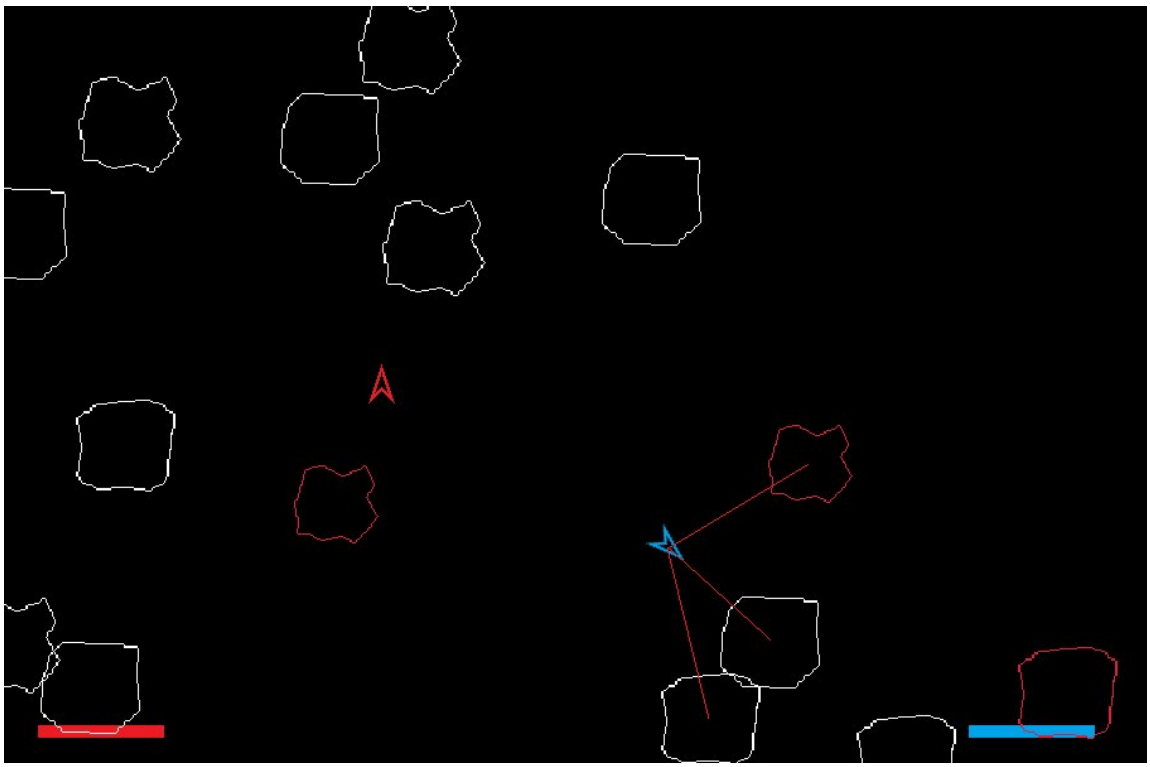
3.2 Senzor

Senzorem nazveme metodu, která nám z kompletního stavu reprezentovaného seznamem vesmírných objektů extrahuje nějakou užitečnou informaci, která není ve stavu explicitně zadána. Informace získané z těchto senzorů, respektive senzorických metod, můžeme využít v rozhodovacím problému vybrání akcí. Většina senzorických metod využívá simulování hry. Na základě současného rozpoložení vesmírných objektů ve stavu se v rámci simulace pokračuje v jejich pohybu, tak jak by se pohybovaly, pokud by žádný z hráčů neprováděl žádné akce. A na základě toho, co se stane v nejbližších krocích simulace, můžeme zjistit konkrétní informace, které platí o současném stavu hry. V simulaci žádný z hráčů neprovádí žádné akce, kromě těch, na jejichž dopad se v dané simulaci dotazujeme. Všechny simulace probíhají omezený počet kroků. Tento počet kroků je roven konstantě *IMPACT_RADIUS*. Pro tuto hodnotu se mi ukázal být vhodný počet 25. Je to dostatečně vysoká hodnota, aby senzory včas zaznamenaly potřebné informace a zároveň je dostatečně nízká, aby senzorické metody nebyly výpočetně zbytečně náročné.

3.2.1 Příklady senzorů

- První sražený neutrální asteroid - Zde se simuluje pohyb vesmírné lodi, střel a neutrálních asteroidů. V případě, že v simulaci dojde k srážce vesmírné lodi a neutrálního asteroidu, vrací tento senzor daný asteroid a počet kroků, po kterém došlo ke srážce. Berou se zde v úvahu i vlastní střely. Pokud dojde k sestřelení asteroidu střelou, jsou jak asteroid, tak i střela odstraněny z následné simulace.
- První sražený nepřátelský asteroid - Jde o téměř identický senzor, jen s rozdílem, že se nesoustředí na asteroidy neutrální, ale na asteroidy nepřátelské.
- Asteroid zasáhne nepřátelskou loď - Zde se simuluje pouze pohyb konkrétního asteroidu a nepřátelské lodí. Opět je zde nastaven daný limit na počet simulovaných kroků. Senzor vrací informaci zda, a v kolika krocích se střetl s nepřátelskou lodí. V simulaci nepřátelská loď neprovádí žádné reakce.

- Střela zasáhne konkrétní asteroid - Jde o simulaci podobnou předchozí. Simuluje se pohyb konkrétního asteroidu a konkrétní střely.
- Střela zasáhne libovolný asteroid - Simuluje se pohyb konkrétní střely a všech neutrálních a nepřátelských asteroidů. Tato senzorická metoda vrací zda střela zasáhla sestřelila asteroid, daný asteroid a počet kroků, po kterém střela sestřelila asteroid.
- Vzdálenost dvou bodů - Vrátí vzdálenost vyjádřenou v Euklidovské metrice.
- Přepočítání nejbližší polohy asteroidu od vesmírné lodi - Vzhledem k tomu, že prostor je jistým způsobem cyklický, tak v případě, že nás zajímá nejkratší vzdálenost asteroidu od vesmírné lodi, tak přímá vzdálenost těchto objektů, tak jak jsou graciky objekty zobrazeny v herním prostoru, nemusí být nejkratší. Musíme vzít v úvahu všechny čtyři polohy asteroidu, které získáme postupným posunutím asteroidu o šířku a délku prostoru. Jinak řečeno, vzdálenost souřadnice asteroidu posunutého přes hranici prostoru může být kratší než přímá vzdálenot k původní souřadnici asteroidu.
- N nejbližších asteroidů od vesmírné lodi - Tento senzor spočítá nejkratší vzdálenosti vesmírné lodi ke všem asteroidům ve hře a vrátí relativní polohu N nejbližších z nich k vesmírné lodi.



Obrázek 3.1: Ukázka senzoru "N nejbližších asteroidů od vesmírné lodi"pro N=3

3.3 Akční plán

Druhou zmíněnou abstrakcí jsou akční plány, jejich smyslem je, podobně jako u senzorů, namísto elementárních akcí nízké úrovně volit akční plány vyšší úrovně. Akční plán představuje posloupnost množin elementárních akcí. Akční plány mají vždy nějaký cíl a záleží o jaký typ akčního plánu jde. Hlavní myšlenkou akčních plánů je, že využitím posloupnosti akcí, které aplikujeme během následujících krocích hry, můžeme dosáhnout komplexnějších důsledků, než kterých dosáhneme jednorázovým provedením libovolné elementární akce.

Podobně jako u senzorických metod i zde hledání většiny akčních plánů probíhá simulací hry. Každý akční plán se snaží dosáhnout konkrétního cíle. V rámci simulace se

3.3.1 Jednotliv plány

- Útočný plán - V této simulace se hledá, jak se má vesmírná loď otočit a jakou střelu vystřelit, aby rozstřelila asteroid, kterým může zasáhnout nepřátelskou loď. Simulace postupně prochází všechny možné otočení a následné střely. Postupuje se inkrementálně. Vesmírná loď se v simulaci jednou otočí, vystřelí střelu a pokud střela zasáhne nějaký střední, nebo veliký asteroid. Nad tímto asteroidem se následně zkouší, jestli vzniklé asteroidy rozstřelením zasáhnou nepřátelskou loď. Rozstřelení asteroidu se zkouší pro oba typy střel.

  V případě, že vystřelená střela nezasáhla žádný z asteroidů požadované velikosti, nebo rostřelené asteroidy nezasáhy cíl, se v simulaci vesmírná loď pokusí provést další rotaci a celý pokus o střelu opakovat. Pokud v simulaci nastalo úspěšné sestřelení, tak se vrací útočný akční plán, který obsahuje přílušný počet rotací následovaný střelou.

- Obranný plán - Pro hledání obraného plánu je nejprve potřeba vědět, před kterým asteroidem se chceme bránit. A tomu právě využijeme připravené senzory. Obraný plán je vlastně složením dvou částí. První částí je rotace vesmírné lodi tak, aby mířila na asteroid, před kterým se chce bránit. A druhá část je samotná střelba, ta už není zcela součástí akčního plánu. Vyhodnocení, zda vystřelením střely sestřelíme konkrétní asteroid je opět záležitost jednoduché senzorické metody.

  Rotace vesmírné lodi, aby byla orientována směrem k danému asteroidu, není potřeba provádět simulací, stačí nám k tomu statický výpočet. Nejprve musíme zjistit, kde má ke srážce dojít. V případě, že vesmírná loď, anebo asteroid stojí staticky na místě, tak ke srážce dojde na současné poloze vesmírné lodi. Pokud jsou ale oba objekty v pohybu, tak ke srážce dojde na zcela jiném místě a toto musíme vzít v potaz. Pokud by se vesmírná loď otočila pouze vzhledem k současné poloze asteroidu, tak je možné, že by střelou mohla letící asteroid minout. Proto se namísto současné polohy asteroidu míří na cílovou polohu. Cílová poloha pro nás bude bod, který leží na přímce spojující tyto dva body a to ve vzdálenosti 15 procent jejich vzdálenosti blíže k polože asteroidu. Tímto způsobem bude střela letět do směru letu asteroidu. Empiricky bylo vyzkoušeno, že toto funguje velmi

dobře. Poté co určíme cílovou polohu, tak jen spočítáme rozdíl současného úhlu vesmírné lodi, od úhlu k cílové polože. Akční plán pak bude obsahovat potřebný počet rotací.

- Úhybný plán - Jedná se také o defenzivní plán, ale místo přímé sestřelené nebezpečného asteroidu, se tento akční plán snaží asteroidu vyhnouot. Podobně jako u obranného plánu, potřebujeme vědět, jakému asteroidu se chceme vyhnout. Simulace postupně prochází všechny možné otočení a následnou akceleraci. Simulace akcelerace zkouší také inkrementálně počet potřebných akcelerací. Nejprve se vyzkouší provést akci akcelerace jednou a následně se simuluje pohyb vesmírné lodi a asteroidu. Pokud došlo ke srážce, tak se vyzkouší provést akci akcelerace dvakrát a opět se simuluje, zda se objekty srazí. V případě úspěšného vyhnutí se vrátí úhybný plán obsahující akce rotace a následně potřebný počet akcí akcelerace.

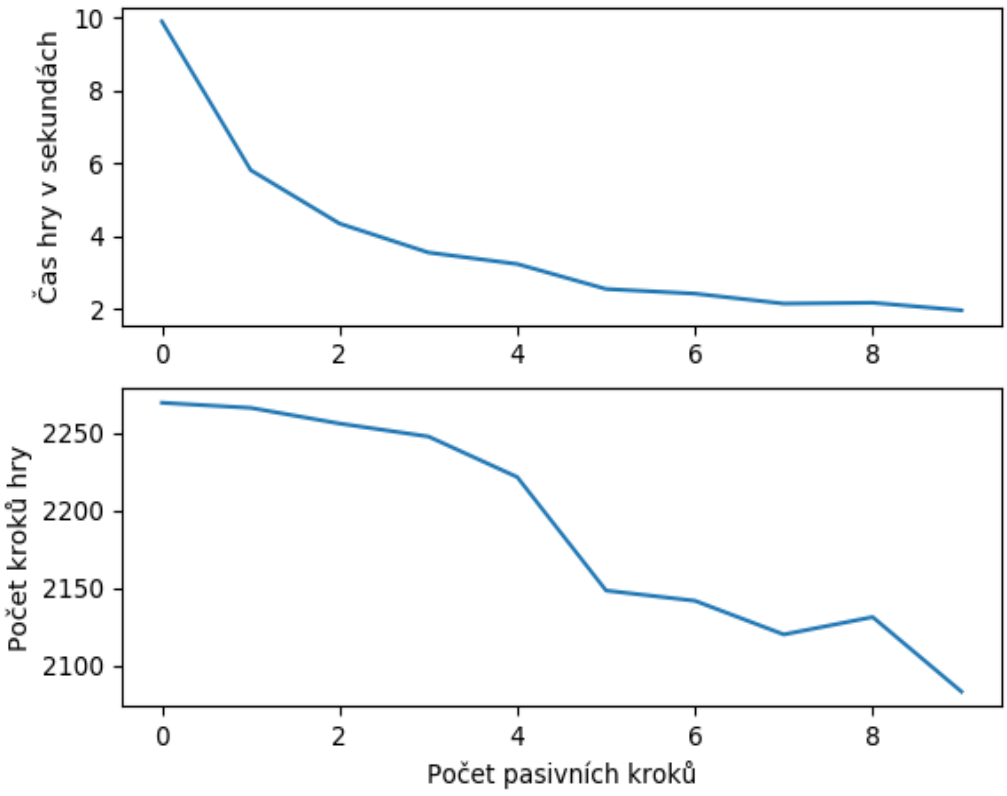
- Zastavovací plán - Tento plán jsem vytvořil na základě sledování jak se chová úhybný plán. Úhybný plán vždy vrací nejkratší plán, který stačí na vyhnutí se srážce, proto většinou obsahuje minimálně počet rotací, který je dostačující. To má za následek, že agent, který se řídí pouze úhybními plány používá akceleraci mnohem častěji než rotace a lítá tak obrovskou rychlostí napříč prostorem. Proto mě napadlo, že by se mohl hodit akční plán, který vesmírnou loď uvede do klidu.

  Zastavovací plán má přímočarou myšlenku. Nejdříve se vesmírná loď zrotuje, aby byla nasměrována proti směru pohybu a následně provede potřebný počet akcelerací, aby zpomalovala až do úplného zastavení. Zastavovací plán ve výsledku obsahuje jistý počet akcí rotace následovaný potřebným počtem akcí akcelerace.

  Tento akční plán mi připadal takový více lidský. V kombinaci s úhybným plánem se pak agent chová více přirozeně. Namísto zběsilého letu napříč prostorem se vesmírná loď po vyhnutí asteroidu zastaví.

3.3.2 Přepočítávání akčních plánů

Získávání akčních plánů je výpočetně náročné, proto by bylo vhodné omezit jejich přepočítávání. Akční plány nám vracejí posloupnost akcí na více kroků dopředu a proto není potřeba je přepočítávat v každém kroku. Změny mezi dvěmi po sobě jdoucími stavy hry není veliká, ale může být dostatečná na to, aby se přepočítaný plán lišil od předchozího. V každém kroku hry se rozhoduje, zda se bude plán přepočítávat. Když je jednou plán vypočítaný, tak v dalších krocích stačí pouze vzít další akce z plánu a není potřeba žádného jiného výpočtu. Plán se přepočítává, pokud uplynul daný počet pasivních kroků, ve kterých jsme pokračovali v již vytvořeném plánu, a nebo byl předchozí plán dokončen. Počet neaktivních kroků se rovná konstaně *INACTIVE_STEPS_LIMIT*. S vyšším počtem pasivních kroků se velmi výrazně snižuje celkový čas odehrátí hry, ale počet kroků během hry se sníží jen minimálně (viz obr03). Je zde vidět, že kvalita hráčů se mírně snižuje, ale v poměru kolik ušetříme času, je tato ztráta kvality zanedbatelná. V příštích kapitolách se nám bude pro učení hodit odehrát velké množství her, proto si dovolíme nastavit vyšší počet pasivních kroků.



Obrázek 3.2: Srovnání počtu neaktivních kroků k průměrné délce hry. Čísla byla získána průměrováním 40 her, kde proti sobě hráli agenti využívající pouze obranných plánů.

3.3.3 Použití akčních plánů

Metody, které vypočítávají akční plán vracejí kromě plánů samotných i jejich délku. V případě, že akční plán není nalezen, tak se místo jeho délky vrací konstanta *NOT_FOUND_STEPS_COUNT* vysoké hodnoty, reprezentující, že akční plán neexistuje. Díky tomu lze tyto metody použí také jako senzorické metody.

4. Genetické programování

4.1 Základní princip

Genetické programování je evoluční technika, která vytváří počítačové programy. Cílem genetického programování je vyřešit co nejlépe zadaný problém. Nespecifikujeme jak je potřeba ho vyřešit, ale víme, co je potřeba vyřešit.

Každý program budeme nazývat jedincem a množině jedinců budeme říkat populace. Algoritmus pak běží iterativně v generacích. V každé iteraci se provede výběr nějakých jedinců z populace, ti se pomocí genetických operátorů upraví a nakonec se rozhoduje, kteří z nich přežijí do další generace. Populaci na začátku iterace nazýváme rodiče a populaci, která bylo nakonci iterace pro další generaci, nazýváme potomky. Každý program představuje jedno řešení daného problému. Kvalitu tohoto řešení ohodnocujeme takzvanou fitness funkcí. Čím vyšší hodnotu této funkce jedinec získá, tím lépe řeší daný problém.

4.1.1 Reprezentace

Program je obvykle reprezentovaný syntaktickým stromem. Stromy ve vnitřních uzlech obsahují funkce (neterminály) a v listech terminály. Vyhodnocení stromu probíhá od listů ke kořeni a výsledek programu je pak hodnota kořenu.

4.1.2 Inicializace populace

Na začátku evoluce potřebujeme inicializovat populaci. Na počátku jsou jedinci vytvořeni náhodně. Obvykle se k vytvoření jedinců používají dvě metody. První z nich je vytváření jedinců s pevně danou hloubkou stromu, kde v listech jsou vždy už jen terminály. Druhou metodou je stavět strom náhodně z předem daného počtu neterminálů. Po vyčerpání počtu neterminálů se již opět přidají terminály jako listy. Tato metoda vytváří jedince různých velikostí a tvarů. Častou praxí je prvotní populaci vytvořit tak, že každá z metod vytvoří polovinu jedinců.

4.1.3 Selekce

V každé iteraci chceme vybrat několik jedinců, nad kterými budeme provádět různé genetické operátory. Snahou je vybírat jedince s vyšší hodnotou fitness funkce. Asi nejpoužívanější metodou selekce je turnajová selekce. Náhodně se zvolí daný počet jedinců, ti se porovnají mezi sebou na základě jejich hodnoty fitness funkce a nejlepší z nich je pak zvolen do výběru. Dalším z mnoha metod selekce je ruletová selekce. Zde je pravděpodobnost zvolení jedince do výběru přímo úměrná jeho hodnotě fitness funkce. Pravděpodobnost výběru i-tého jedince je $p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$.
Ruletová selekce má jednoduchou implementaci a zároveň je rychlá na výpočet. Jejím problémem je ale předčasná konvergence k lokálnímu optimu.

Genetické operátory mohou občas upravit nejlepšího jedince tak, že se jeho hodnota fitness funkce výrazně zhorší. Z tohoto důvodu se často používá technika zvaná elitismus, která automaticky do další generace vybere několik nejlepších

současných jedinců. Tímto způsobem máme zajištěno, že neztratíme nejlepší jedince.

https://www.researchgate.net/publication/259461147_Selection_Methods_for_Genetic_Algorithms

4.1.4 Genetické operátory

Genetické operátory jsou dvojího druhu, křížení a mutace. Myšlenkou křížení je vzít dva jedince, nazývejme je rodiče, a pomocí křížení informací každého z nich vytvořit nového potomka. V genetickém programování pracujeme s jedinci reprezentovanými stromy, proto křížení jedinců představuje křížení jejich stromů. V každém z rodičů se zvolí jeden uzel. Výsledný potomek vypadá jako první rodič, jen na původním místě vybraného uzlu bude nyní podstrom, který je zavěšený pod vybraným uzlem druhého rodiče.

Druhým genetickým operátorem je mutace, ta již nepotřebuje mít dva rodiče, ale úprava se provede nad jedincem samotným. Mutovat můžeme v jedinci buď jediný bod, nebo celý nějaký podstrom. V případě mutace podstromu se namísto vybraného podstromu vygeneruje zcela nový podstrom. Toto v zásadě představuje křížení s novým náhodně vytvořeným jedincem.
Mutace jediného bodu změní náhodně jediný uzel ve stromě. V případě terminálu se může vybrat libovolný jiný terminál. A v případě vnitřního uzlu může být vybraný libovolný jiný neterminál, který je stejné arity.

4.1.5 Silně a volně typované genetické programování

Ve volně typovaném genetickém programování nezadáváme typy funkcí ani terminálů. Jediné co musíme u funkcí určit je jejich arita a na základě toho se generují a mutují jedinci korektně. Obvykle se nám více bude hodit silně typované genetické programování, zde určujeme u všech funkcí nejen jejich aritu, ale také typ každého z argumentů dané funkce a také typ návratové hodnoty. Podobně musíme určit i hodnotové typy terminálů. Díky tomu máme zaručeno, že pokud máme například funkci, která vrací typ boolean, tak tato hodnota nebude vynásobena hodnotou typu float. Pak jen určíme hodnotové typy celého problému a algoritmus už se postará o to, aby byly typové podmínky dodrženy.
https://deap.readthedocs.io/en/master/tutorials/advanced/gp.html

4.2 Využití

Genetické programování je využitelné ve všech problémech, kde jsme schopní vymyslet způsob jak reprezentovat jedince představujícího řešení problému a fitness funkci, díky které jsme schopni dané řešení ohodnotit. To tedy znamená, že možnosti využití jsou téměř nekonečné. Zmiňme ale pár konkrétních příkladů.

4.2.1 Obecné využití

volně přeloženo a zkráceno z původního textu str. 126 Obecně se genetické programování ukazuje být vhodné ve všech problémech, které splňují některou, z následujících podmínek:

- Vzájemné vztahy zkoumaných proměnných nejsou dobře známy, nebo je podezření, že jejich současné porozumění může být mylné.
- Nalezení velikosti a tvaru hledaného řešení je částí řešeného problému.
- Existují simulátory pro testování vhodnosti zadaného řešení, ale neexistují metody pro přímé získání dobrých řešení.
- Obvyklé metody matematické analýzy nedávají, nebo ani nemohou být použity pro získání analytického řešení.
- Přibližné řešení je zcela postačující.

4.2.2 Symbolická regrese

V mnoha problémech je naším cílem nalézt funkci, jejíž hodnota splňuje nějakou požadovanou vlastnost. Toto známe pod názvem symbolická regrese. Obyčejná regrese má obvykle za cíl nalézt koeficienty předem zadané funkce, tak aby co nejlépe odpovídala daným datům. Zde je problém, že pokud potřebná funkce nemá stejnou strukturu, jako zadaná funkce, tak dobré koeficienty nenalezneme nikdy a musíme zkusit hledat funkci jiné struktury. Tento problém může vyřešit právě symbolická regrese. Ta hledá vhodnou funkci aniž by na začátku měla očekávání o její struktuře. Uvedeme triviální příklad. Řekněme, že hledáme výraz, jehož hodnoty odpovídají polynomu $x^2 + x + 1$ na intervalu $[-1,1]$. Budeme hledat funkci jedné proměnné x, proto x přidáme jako terminál. Dále přidáme jako terminály číselné konstanty (například -1,1,2,5,10), které budou sloužit pro hledání koeficientů. Pro náš případ bude stačit, když si jako aritmetické funkce přidáme ty základní, tedy sčítání, odečítání, násobení a dělení. Fitness funkci můžeme zvolit jako součet absolutních hodnot rozdílů daného výrazu a hledaného výrazu x^2+x+1. Toto stačí pro spuštění evolučního algoritmu. V takto triviálním případě se hledané řešení nalezne pravděpodobně vezmi brzo a bude mít jednu z podobu stromu reprezentujícího daný výraz (viz strom výrazu 4.1)

4.2.3 Další příklady použití

Symbolická regrese je jen jedním z mnoha problémů, které lze řešit pomocí genetického programování. Podobě jako ve field guide to genetic programming vyjmenovat konkrétní výsledky s odkazy na stejné články jako v knize? Nebo zmínit jen oblasti bez vysvětlování konkrétních problémů? - (Bioinformatika, počítačové hry, Ekonomické modely, Umění)

4.3 Aplikace

4.3.1 Úvod

Genetické programování lze využít i v mém problému hledání inteligentního agenta. Nezbytným požadavkem pro využití genetického programování je existence reprezentace jedince a fitness funkce, která mu přiřadí jeho hodnotu. Jedinec bude představovat rozhodovací funkci, která se na základě vstupních argumentů rozhodne, který akční plán bude vybrán.

V našem případě máme hru, kde spolu dva hráči soupeří a hra končí výhrou jednoho z hráčů. Přesně tohoto můžeme ve fitness funkci jedince využít. Pokud chceme využít výsledku hry, jak jedinec ve hře dopadl, tak musíme ale nejprve zvolit proti jakému hráči bude jedinec, který je předmětem našeho zájmu, hrát.

Genetické programování jsem využil ve dvou experimentech. Reprezentace jedince je pro oba experimenty stejná, rozdíl je ale v přístupu k fitness funkcím, ty popíšu v každém experimentu separátně.

Pro experimentování s genetickým programováním jsem zvolil knihovnu deap pro python. Zde lze jednoduše konfigurovat evoluční algoritmus na konkrétní řešený problém. Stačí popsat jak reprezentovat jedince a jaká je jeho fitness funkce a zbytek knihovna vyřeší za nás.

4.3.2 Reprezentace jedince

V předchozí kapitole jsme si vybudovali abstrakce v podobě senzorů a akčních plánů a těch zde budeme chtít využít. Jedinec, podobně jako u symbolické regrese, je funkce, tedy může být reprezentován stromem.

- Terminály:
  - Vstupní argumenty rozhodovací funkce
  - Celočíselné konstanty -1, 1, 3, 5, 10, 100
  - Nulární funkce vracející výčtové hodnoty reprezentující zvolený akční plán

  Jako argumenty funkce jsem si zvolil následující hodnoty: délky všech čtyř akčních plánů a počet kroků před srážkou vesmírné lodi s asteroidem. Délky akčních plánů se pohybují v intervalu (1,100), proto jsou číselné konstanty zvoleny tak, aby se jejich sčítáním a násobením lehce dosáhlo dalších hodnot z tohoto intervalu.

- Neterminály:
  - Aritmetické operace sčítání a násobení
  - Funkce *compare*
  - Funkce *if_then_else*

  Z aritmetický operací nám stačí sčítání a násobení. Operaci odčítání získáme pomocí sčítání a násobení konstantou -1. Hodnoty z intervalu (1,100) jednoduše získáme také pomocí sčítání a násobení potřebných konstant, proto pro operaci dělení není důvod. Všechny aritmetické operace jsou typu $([int,int], int)$. Funkce *compare* je typu $([int,int], Bool)$, vrací zda první argument je větší než druhý argument. Poslední použitá funkce *if_then_else* je typu $([Bool, ActionPlanEnum, ActionPlanEnum], ActionPlanEnum)$. Tato funkce dostává jako argumenty výraz typu bool a následně dvě hodnoty reprezentující akční plány. Na základě pravdivosti výrazu vrací funkce první nebo druhou z hodnot akčních plánů.

4.3.3 Experiment 1: Soupeření s obranným agentem

Cílem tohoto experimentu bylo vyvinout agenta, který bude lepší než agent, který se řídí čistě obranným akčním plánem. Výpočet fitness funkce zahrnuje zahrání 6 her současného jedince s obranným agentem a výsledek je průměr z hodnot každé z her. Hra je pokaždé velmi náhodná, tedy zahrání jedné hry by mělo nízkou vypovídající hodnotu. Proto jsem pro přesnější informaci zvolit zahrání 6 her. Hodnota zahrané hry se skládá z více částí.

- Počet kroků trvání hry
  Myšlenkou je zde, obzvláště v počátku evoluce, upřednostňovat takové jedince, kteří dokážou vydržet ve hře co nejdéle, tedy nejsou ve hře okamžitě poraženi. Délka hry se pohybuje pro představu v intervalu (900,2900) kroků.

- Penalizace za nevyužití některého z plánů
  Během hry se udržuje historie, kolikrát se agent rozhodl pro každý z akčních plánů. Za každý ze čtyř akčních plánů, který agent ani jednou během hry nezvolil získá penalizaci -500. Cílem těchto penalizací je upřednostňovat takové jedince, kteří používají všechny akční plány. Toto jsem se rozhodl udělat pro větší diverzifikaci jedinců.

- Extra bonus za útočný a obranný plán Zkontrolovat zda je toto spravne
  Přestože chci od jedinců aby používali všechny akční plány, tak očekávám, že hledaný agent bude převážně používat útočný a obranný plán. K výsledné hodnotě se přičte počet kroků, ve kterém agent zvolil útočný nebo obranný akční plán.

- Bonus/penalizace za výhru/prohru
  Toto je asi nejdůležitější část. Pro zdůraznění rozdílu mezi vyhranými a prohranými hrami se v případě výhry přičtou k výsledku 2000 a v případě prohry se 2000 odečtou. Motivací mohou být následující dvě situace. V jedné hře se podařilo jedinci dlouho bránit, řekněme, že vydržel 2500 kroků hry a poté prohrál. V druhé hře porazil soupeře v rychlých 1200 krocích. Bez bonusu za vyhranou hru, by prohraná hra získala jedinci daleko vyšší hodnotu, než hra, kterou vyhrál.

Algoritmus byl spuštěn s následujícími parametry:

- Velikost populace: 30
- Pravděpodobnost křížení: 60%
- Pravděpodobnost mutace: 20%
- Počet generací: 100
- Metoda selekce: turnajová selekce

Výsledný nejlepší jedinec bohužel nesplnil naše očekávání. Zde předkládám data získané ze zahrání 20 her proti obrannému agentovi.

doplnit reálná čísla Náš nalezený agent nejen není lepší než obranný agent, ale ani nemá moc odlišný přístup k volbě akčních plánů. V 97% volil také obranný akční plán a ve zbylých procentech párkrát volil všechny ostatní akční plány, tohoto bylo zřejmě docíleno kvůli penalizaci nepoužívání některých z akčních plánů.

4.3.4 Experiment 2: Postupné zaměňování úspěšnějšího jedince

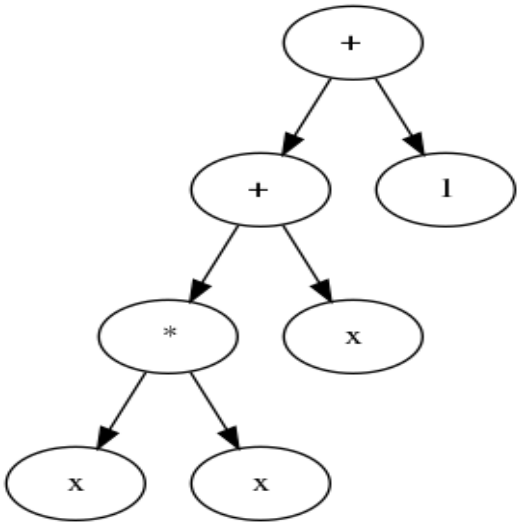
V tomto experimentu nebylo cílem porazit konkrétního, stálého agenta jako v předchozím případě. Cílem v tomto experimentu bylo postupně vybudovat nejzdatnějšího agenta. Stejně jako v předchozím případě i zde fitness funkce bude zahrnovat zahrání 6 her. Avšak zde nebudeme počítat žádné speciální hodnoty, jak která z her dopadla, ale spokojíme se s jednoduchou informací, který z agentů danou hru vyhrál.

Po celou dobu evoluce si budeme pamatovat současného nejlepšího jedince. Na začátku zinicializujeme zcela náhodného jedince a označíme ho jako současného nejlepšího jedince. Pak vytvoříme populaci dalších jedinců a započneme evoluci. Fitness funkce jedince bude reprezentovat poměr, kolik ze 6 zahraných her jedinec vyhrál v souboji se současně nejlepším nalezeným řešením. Evoluce hledá řešení, která budou proti současnému nejlepšímu co nejlepší. Každou 3. generaci se kontroluje, zda již náhodou nebyl v populaci nalezen jedinec, který současně nejlepšího jedince porazil alespoň v 5 ze 6ti her. Pokud ano, tak takový jedinec bude nově zvolen jako nejlepší a celý proces bude pokračovat stejným způsobem.

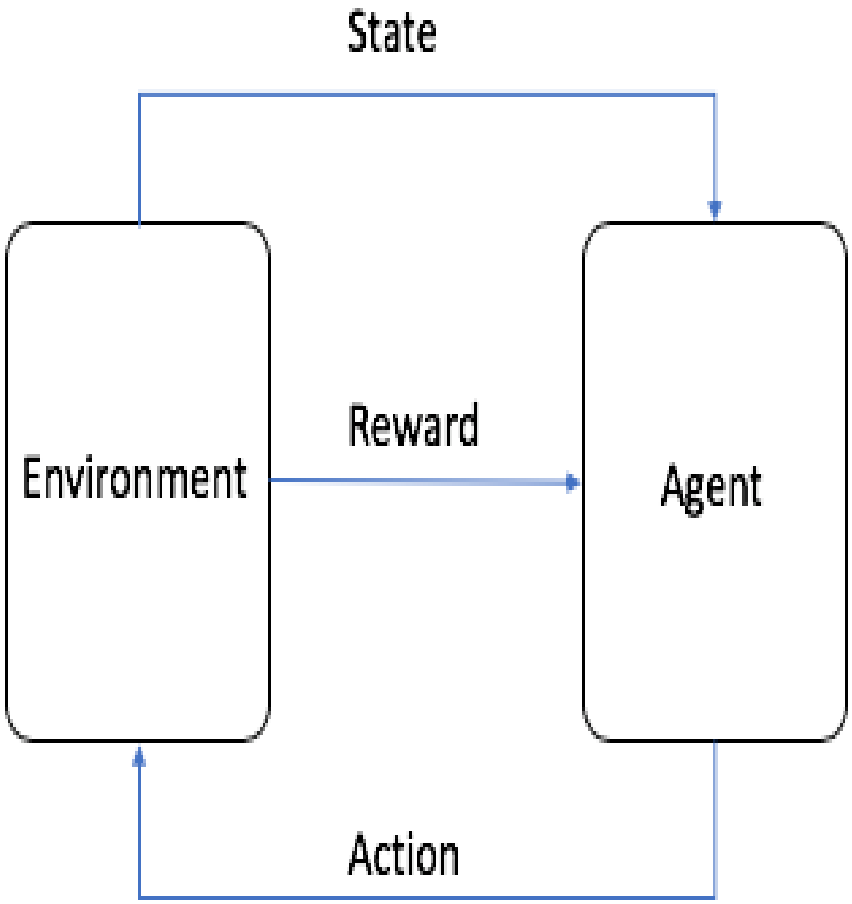
Po výměně nejlepšího jedince musíme nově přepočítat fitness funkci všech stávajících jedinců v populaci, protože jejich současná hodnota se vztahovala k původním jedinci. Rovněž musíme, ze stejného důvodu, smazat všechny jedince ze síně slávy (ang. Hall of fame), kde se průběžně ukládají nejlepší jedinci spolu s hodnotou jejich fitness funkce.

Všechny tyto, výše zmíněné, změny už nelze nakonfigurovat přímočarým způsobem jako v předchozím experimentu. Ale bylo zapotřebí upravit samotnou kostru evolučního algoritmus.

2.) Zde postupné volení lepšího a lepšího jedince vzájemnými souboji. Na začátku zvolit náhodného jedince jakožto nejlepšího. Nechat ho zahrát s ostastními a ten, kdo ho nejvíckrát porazil, tak ho nahradí. Ten bude nahrazen až se najde protivník s 5 ze 6 výher proti němu. Zde bylo třeba mírně upravit kostru algoritmu. Smazání Hall of fame po smazání. každé 3 generace kontrola, zda už nebyl naleen nějaký lepší soupeř. Celkově jsem zahrál 500 generací. Výsledkem je dobrý jedinec, který poráží Stable Deffensive Agenta, ale pouze v nějakých random seedech. Při vysokém inactive steps začne prohrávat. Výsledný poměr vobly akčních plánů je 2/3 obrana, 1/3 útok



Obrázek 4.1: Strom výrazu



Obrázek 4.2: Herní cyklus

5. Hluboké Q-učení

5.1 Základní princip

5.2 Využití

Kde se používá v praxi.

5.3 Aplikace

Jak jsem to použil já a jakých výsledků jsem dosáhl.

6. NEAT

6.1 Základní princip

6.2 Využití

Kde se používá v praxi.

6.3 Aplikace

Jak jsem to použil já a jakých výsledků jsem dosáhl.

Seznam obrázků