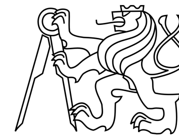
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	U ení automatických hrá online her ve webovém prohlíže i
Student:	Bc. Jakub Melezínek
Vedoucí:	Ing. Jaroslav Kucha
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Na webu existuje ada problém , pro které je spole ným pojítkem strojové u ení jako nástroj pro jejich ešení. Cílem práce je u ení automatických hrá (bots) hraní her p ímo ve webovém prohlíže i s využitím možností jazyka JavaScript.
- Nastudujte problematiku strojového u ení se zam ením na evolu ní techniky a u ení hrá .
- Prove te rešerši dostupných knihoven v jazyce JavaScript.
- Navrhn te, implementujte a otestuje vlastní knihovnu v jazyce JavaScript. Zam te se na neuronové sít a rozhodovací stromy. Popište výhody, nevýhody a p ípadn omezení jazyka JavaScript pro tento typ úloh.
- Implementujte webovou aplikaci s GUI využívající tuto knihovnu a techniku rozší eného prohlížení (augmented browsing), tak aby umož ovala u ení hrá a jejich použití p ímo ve webovém prohlíže i.
- Funk nost demonstrujte na dostupné online h e Agar.io. Prove te experiment a vyhodno te kvalitu nau ených hrá . Vyhodno te vlastnosti prohlíže (nap . JavaScript engine) a jejich dopad na u ení hrá .

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

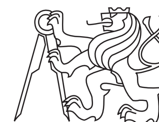
Ing. Michal Valenta, Ph.D. vedoucí katedry	prof. Ing. Pavel Tvrdík, CSc. d kan

V Praze dne 17. ledna 2016

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství



Diplomová práce

Učení automatických hráčů online her ve webovém prohlížeči

Bc. Jakub Melezínek

Vedoucí práce: Ing. Jaroslav Kuchař

17. února 2017

Poděkování

Především bych chtěl poděkovat vedoucímu mé práce, panu Ing. Jaroslavu Kuchařovi, za vstřícný přístup, milé jednání a trpělivost. Dále děkuji všem, kteří mě při studiu i psaní práce podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 17. února 2017 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Jakub Melezínek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Melezínek, Jakub. *Učení automatických hráčů online her ve webovém prohlížeči.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Strojové učení je používáno v mnoha oblastech včetně webu. V této práci je použito při implementaci učenlivých automatických hráčů hrajících webové hry přímo v prohlížeči. Jsou diskutovány rozhodovací systémy pomocí umělých neuronových sítí a rozhodovacích stromů. Nakonec je k učení využito neuroevoluce NEAT algoritmem. Výsledkem je knihovna napsaná v jazyce TypeScript a GUI Angular aplikace, která umožňuje ovládání evolučního cyklu a vizualizaci šlechtěných jedinců. Knihovnu využívají dvě implementace botů. První řeší problém vyvážení převráceného kyvadla. Druhý se pokouší hrát hru Slither.io podobnou legendárni hře Snake. Skripty jsou vkládány do hry na straně klienta a lehce upravují původní kód hry. Provedené experimenty ukazují na funkčnost knihovny.

Klíčová slova web, prohlížeč, JavaScript, TypeScript, rozšířené prohlížení, vkládání skriptů, umělá inteligence, AI, strojové učení, simulovaná evoluce, rozhodovací stromy, umělá neuronová síť, NEAT, automatický hráč, HTML5 hra, problém vyvážení kyvadla, agar.io, slither.io

Abstract

Machine learning is used in many areas, including the Web. In the thesis it is used to implement self-learning bots that play web games right in a bowser. For decision making system I discuss options such as artificial neural networks or decision trees. At the end neuroevolution through augmenting topologies (NEAT) is chosen. As result of the thesis I created library written in TypeScript and Angular GUI application on top of it. It is possible to control evolution cycle through this application and see processed individuals. Two bots import the library. The first one plays game based on the pole-cart problem. The second one tries to play Slither.io, game similar to legendary Snake. Skripts are injected to game on client side and slightly changes the original game code. Results of the experiments suggest that application is working properly.

Keywords web, browser, JavaScript, TypeScript, script injection, augmented browsing, artificial intelligence, AI, machine learning, simulated evolution, artificial neural network, NEAT, bot, HTML game, pole balancing problem, agar.io, slither.io

Obsah

Úvod **1**
Motivace 1
Struktura práce 4

1 Cíl práce **5**
1.1 Popis problému 5
1.1.1 Hra Agar.io 5
1.1.2 Automatický hráč 6
1.2 Specifikace cílů 7

2 Rešerše **9**

3 Analýza **11**
3.1 Strojové učení 11
3.1.1 Učení s učitelem a bez učitele 12
3.2 Simulovaná evoluce 13
3.2.1 Názvosloví 13
3.2.2 Schéma cyklu 14
3.2.3 Konvergence versus diverzita 15
3.3 Rozhodovací stromy 15
3.3.1 Model Agar.io 15
3.3.2 Hráči Agar.io 16
3.4 Umělé neuronové sítě 18
3.4.1 Umělý neuron 19
3.4.2 Učení ANN 19
3.5 NEAT 20
3.5.1 Kódování fenotypu 20
3.5.2 Mutace 21
3.5.3 Křížení 22
3.5.4 Druhy 23

3.5.5 Počáteční populace 25
3.5.6 Výsledky/závěry práce 25
3.6 JavaScript a TypeScript 26
3.7 Rozšířené prohlížení 26

4 Návrh a implementace **29**
4.1 Model 32
4.1.1 BaseNeatIndividual 32
4.1.2 NodeGene 34
4.1.3 ConnectGene 34
4.1.4 Species 35
4.2 EvoCycle 36
4.3 EvoCycleControls 38
4.4 Boti 40
4.4.1 SlitherioController 41
4.4.2 SlitherioBot 42
4.5 Sestavení a distribuce 44
4.6 Uživatelská příručka 45

5 Experimenty **47**
5.1 Pole Balancing Game jako benchmark prostředí 47
5.1.1 Popis problému 47
5.2 BalancingBot 49
5.2.1 Experiment č. 1 50
5.2.2 Experiment č. 2 51
5.2.3 Experiment č. 3 51
5.2.4 Závěr 54
5.3 SlitherioBot 55
5.3.1 Slither.io versus Agar.io 55
5.3.2 Senzory 55
5.3.3 Experiment č. 1 57
5.3.4 Experiment č. 2 57
5.3.5 Experiment č. 3 59
5.3.6 Závěr 59

Závěr **61**
Zhodnocení 61
Možnosti pokračování 62

Literatura **63**

A Seznam použitých zkratek **67**

B Obsah přiloženého CD **69**

Seznam obrázků

0.1 IBM Deep Blue . . . 2
0.2 Sethbling MARI/O . . . 3

1.1 Snímek obrazovky agar.io . . . 5

3.1 Graf chyby při učení s učitelem . . . 12
3.2 Schéma evolučního cyklu . . . 14
3.3 Snímek obrazovky modelu hry Agar.io v NetLogo . . . 16
3.4 Graf rozhodovacího stromu . . . 17
3.5 Schéma neuronu ANN . . . 19
3.6 NEAT kódování . . . 21
3.7 NEAT mutace . . . 22
3.8 NEAT permutační problém . . . 23
3.9 NEAT křížení . . . 24

4.1 Diagram balíčků . . . 29
4.2 Celkový pohled na třídy . . . 31
4.3 Diagram třídy `BaseNeatIndividual` . . . 32
4.4 Diagram třídy `NodeGene` . . . 34
4.5 Diagram třídy `ConnectGene` . . . 34
4.6 Diagram třídy `Species` . . . 35
4.7 Diagram třídy `EvoCycle` . . . 36
4.8 Diagram fungování cyklu . . . 37
4.9 Snímek obrazovky s angular aplikací `EvoCycleControls` . . . 39
4.10 Diagram třídy `SlitherioBot` . . . 40
4.11 Diagram typů a rozhraní ze třídy `SlitherioController` . . . 42
4.12 Diagram komunikace `SlitherioBot` . . . 43

5.1 Snímek obrazovky Pole Balancing Game . . . 48
5.2 Grafy experimentu č. 1 s `BalancingBot` . . . 50
5.3 Grafy experimentu č. 2 s `BalancingBot` . . . 52

5.4 Grafy experimentu č. 3 s `BalancingBot` 53
5.5 Snímek obrazovky SlitherioBot 56
5.6 Grafy experimentu č. 2 se `SlitherioBot` 58

Seznam tabulek

5.1 Proměnné systému 49

Úvod

Motivace

Strojové učení je podoblastí umělé inteligence (AI) a v dnešní době získává čím dál více na popularitě a také důležitosti. Za tím stojí potřeba automaticky zpracovávat stále větší množství dat. K čemuž jsou vyvíjeny nové a zdokonolovány již známé metody. To vše je umožněno díky stále rostoucí výpočetní síle.

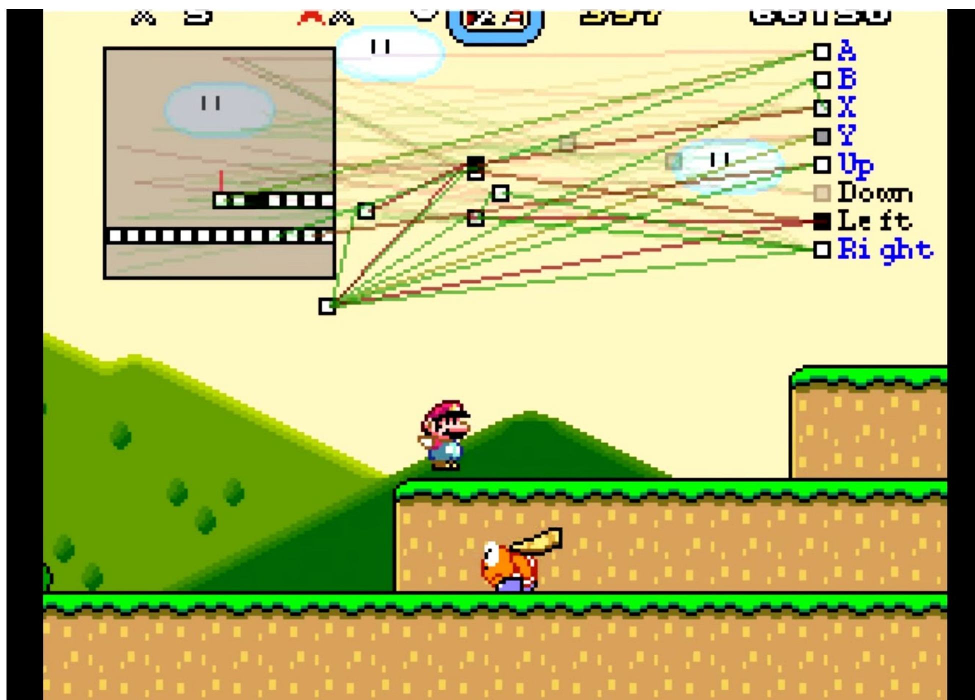
Mnoho oblastí již běžně strojového učení využívá, například filtrování spamu, doporučovací systémy, detekce útoků po síti, odhalování podvodů, počítačové vidění, rozpoznávání jazyka, online reklama, hraní her, pohyb robotů, lékařské diagnózy a mnoho dalších.

Ještě dávno předtím, než jsem začal s programováním, v mých 7 letech (květen 1997), se odehrál médii zpopularizovaný souboj mezi člověkem a strojem, kterému jsem i já přihlížel s očekáváním. Garri Kasparov, tou dobou úřadující mistr šachu, se utkal se superpočítačem Deep Blue firmy IBM a prohrál (celkově 2 výhry pro Deep Blue, 1 pro Kasparova, 3 remízy). [1] Deep Blue ukázaný ná obrázku 0.1 [2] byl schopen vypočítat 200 milionů tahů za sekundu, využíval minimax algoritmu a znalosti předchozích her Kasparova. [3]

V březnu 2016 se udála podobná událost ve hře Go, kdy byl poražen nejlepší hráč Go na světě Lee Sedol (AlphaGo 4 výhry, Lee Sedol 1). Tuto zásluhu si připsal program AlphaGo od Google DeepMind. [4] V této hře je možných kombinací více než atomů ve vesmíru, a proto není možné využít tradičních brute-force přístupů. Místo toho program AlphaGo využívá dvou obrovských (miliony neuronů) umělých neuronových sítí, strojového učení a stromového vyhledávání, k čemuž potřebuje obrovskou výpočetní sílu, kterou mu dodává Google Cloud Platform. [5]

Mě k nápadu také zkusit implementovat některý algoritmus strojového učení přivedl youtuber známý jako SethBling, který v jednom ze svých videí

Obrázek 0.1: IBM Deep Blue superpočítač z roku 1997; volný překlad štítku: Deep Blue byl založen na superpočítači IBM RS/6000 SP2, skládal se z 30 procesorů ve dvou věžích, přičemž na obrázku je zobrazena jedna z nich. Klíčovým pro výkonnost celého systému bylo 480 na zakázku vyrobených čipů pro šachové výpočty.



Obrázek 0.2: Ukázka z videa prezentující program MarI/O. Hráč uprostřed je ovládaný neuronovou sítí zobrazenou v horní polovině - vlevo vstupy z obrazovky, vpravo výstupy kláves, uprostřed skrytá vrstva a spoje neuronové sítě.

[6] ukazuje, jak program MarI/O, viz. screenshot 0.2, který sám napsal, je schopný se naučit známou „skákačku“ Super Mario World od Nintendo.

Fascinovaly mě dvě věci. Zaprvé, že celý program (respektive plugin do simulátoru) byl schopný napsat sám a má pouhých 1224 [7] řádků a využil k tomu poznatků z víceméně jen jedné práce [8]. A zadruhé jak učenlivý tento automatický hráč byl. Na počátku nevěděl, ani jak se pohnout z místa, a po 32. generacích a 24 hodinách učení byl schopný dokončit první level. Vše se naučil pomocí neuroevoluce, specificky NEAT algoritmu. [6]

Jak bylo zmíněno, celá řada využití AI je aplikovatelná na oblast **webu**. Web již není jenom platformou pro text a obrázky, ale vznikají zde plnohodnotné aplikace. Takovými jsou i HTML5 a JavaScript hry. Pokud se na hráče podíváme jako na lidmi ovládané roboty, kteří nemají žádnou fyzickou reprezentaci, pouze tu abstraktní v prohlížeči, můžeme na ně aplikovat algoritmy strojového učení.

Struktura práce

Kapitola 1 Cíl práce V této kapitole je popsán problém automatického hráče pro hry stylu Agar.io a jsou specifikovány cíle, kterých má být v práci dosaženo.

Kapitola 2 Rešerše V rešerši jsou diskutována dostupná řešení a vybrán vlastní postup.

Kapitola 3 Analýza Analýza se zabývá problematikou učení automatických hráčů aplikovatelnou ve webovém prostředí a také vkládáním vlastního kódu do existující webové aplikace.

Kapitola 4 Návrh a implementace Knihovna je podrobně navrhnuta a implementována zvoleným jazykem. Stejně tak vzniknou i konkrétní aplikace vuyžívající této knihovny pro učení hráčů ve zvolených hrách.

Kapitola 5 Experimenty Experimentování je důležitým krokem pro otestování funkčnosti knihovny. Výsledky podrobeny diskusi.

Kapitola 1

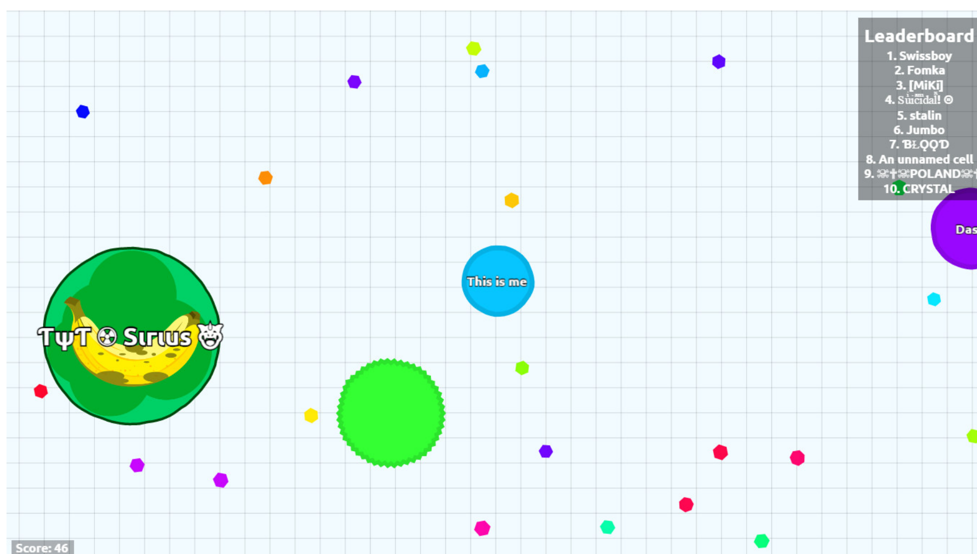
Cíl práce

1.1 Popis problému

1.1.1 Hra Agar.io

Agar.io je jednoduchá webová hra, kde hrajete za buňku pohybující se v Petriho misce pojídající malé barevné kuličky (potrava) a ostatní hráče. Pohyb se ovládá myší a dvěma klávesami. Cílem hry je stát se největši buňkou na mapě. Pro lepší představu je přiložena ukázka 1.1.

Hra se řídí zdánlivě jednoduchým „jez nebo budeš snězen“, ale na hratelnosti přidává několik pravidel. Pouze větší buňka může sníst menší, ale menší buňky se pohybují rychleji a mohou dobře unikat větším predátorům. Od ur-



Obrázek 1.1: Na screenshotu ze hry agar.io je vidět modře buňka hrajícího hráče, jednobarevně zeleně virus, malé tečky jsou potrava a velké ostatní hráči.

čité velikosti může hráč provést „výpad“, čímž se rozdělí na dvě poloviny, které ovládá dohromady. Při „výpadu“ je jedna polovina vymrštěna směrem pohybu a díky tomu může dohnat a pohltit i rychlejší buňku nepřátelského hráče. Pro získání rychlosti se lze vzdát malého množství hmotnosti, které buňka vystřelí v podobě potravy. Buňka ztrácí svou hmotnost a tím velikost s časem. Čím masivnější buňka je, tím rychleji o hmotnost přichází a velké buňky se tedy musí činit, aby si udrželi svůj vzrůst. Neposledním zádrhelem hry jsou viry, které jsou po mapě rozmístěny náhodně. Málé buňky se v nich můžou schovat a jsou chráněny před velkými. Ty totiž, když se viru dotknou, jsou rozdrobněny na několik malých buněk a stávají se tak velice zranitelnými. Hráči soupeří online v reálním čase a tak mohou taktizovat a podporovat se navzájem. [9]

Hru uvedl 28. dubna 2015 Matheus Valadares a stala se rychle populární. [9] V listopad 2015, kdy její obliba, zdá se, dosahuje vrcholu, ji hraje v jednu chvíli i přes 100 tisíc hráčů na různých serverech po celém světě. A za měsíc má kolem 150 milionů návštěv. V listopadu 2016 má již „jen“ 30 milionů návštěv za měsíc. [10]

1.1.2 Automatický hráč

Pro člověka jednoduchá hra na pochopení i ovládání. Není třeba příliš tréninku a hráč je schopný dostat se do nejlepších deseti na dané mapě. I naprogramování bota (hráč ovládaný počítačovým programem), který by dokázal hru hrát, je celkem přímočaré. Je jasný cíl hry a i způsob jak ho dosáhnout. Je třeba navrhnout algoritmus takový, aby se buňka pohybovala prostředím a nejlepším možným způsobem reagovala na nastalé situace. Nejlepším v tomto případě znamená podle statického rozhodování, které určil programátor a natvrdo vepsal do programu.

Obtížná část přijde, pokud je třeba bota nechat, aby se sám naučil jak správně hrát. Zde přichází na řadu strojové učení. Přístupů je více. Například je možné navrhnout a implementovat různé strategie hry (defenzivní, agresivní, vyčkávací) a nechat bota, aby se naučil, při jaké situaci je která strategie vhodná. K tomu se dobře hodí evoluce rozhodovacích stromů. Nebo lze volit obecnější postup, kdy předmětem učení bude i samotné ovládání a jak dosáhnout cíle. Není tedy dostupné žádné programátorem definované chování. To ve své podstatě může vést k nalezení nové cesty k cíli, ovšem vede to k delší době učení. To je příklad učení neuronových sítí, které lze provádět i evolučním algoritmem.

Možné by byly i jiné přístupy, ale tyto dva mají jeden společný rys, tím je šlechtění řešení/botů evolučním algoritmem skládajícím se ze stejných kroků - selekce, křížení, mutace - opakujících se v iteraci. Stejný přístup k učení, různá implementace rozhodovací - šlechtěné - části bota.

1.2 Specifikace cílů

Cílem této diplomové práce je vytvořit aplikaci, která umožňuje automatickým hráčům, takzvaným botům, naučit se hrát různé hry **v prostředí webového prohlížeče**. Aplikace se bude skládat ze **znovupoužitelného** a **rozšiřitelného** jádra (knihovny) napsané v TypeScriptu, které bude obstarávat řízení evolučního cyklu, definovat rozhraní pro implementaci jedinců a obsahovat samotnou implementaci pro neuronové sítě (ANN) a/nebo rozhodovací stromy (DT).

Nad touto knihovnou bude postavena JS/TS aplikace, která technikou **rozšířeného prohlížení** bude získávat data ze hry pro vstupy automatického hráče a také bude vykonávat jeho rozhodnutí, tak jako kdyby hrál skutečný člověk. Tento automatický hráč bude určovaty své herní tahy na základě rozhodovací systému (ANN, DT). Podle dosaženého výsledku bude podroben selekci a případně křížení a mutaci v jádru aplikace.

V rešeršní a analytické části je vhodné se zaměřit na prozkoumání podobných knihoven. Zda již neexistuje řešení pro js/ts, které by se dalo po vhodném rozšíření využít. Díky obrovské popularitě Agar.io budou existovat pluginy pro tuto hru, které by se daly při vývoji použít.

Při návrhu a implementaci klást důraz na znovupoužitelnost a obecnost. Používat přístup objektově orientovaného programování, využívat standartních návrhových vzorů a správně používat frameworky a knihovny. Návrh vytvořit přehledně pomocí diagramů. Testování je nutnou součástí vývoje každého softwaru a nemělo by být opomenuto ani při vývoji této aplikace.

Kapitola 2

Rešerše

Umělá inteligence i strojové učení je do hloubky prozkoumávaný problém a tak není obtížné najít knihovny nebo programy, které se těmto tématům věnují. Specifika této práce však tyto knihovny dost omezují. Půjde o aplikování strojového učení v prohlížeči na již existujícím kódu. K tomuto účelu se téměř výhradně dá použít pouze JavaScriptu, takže nemá cenu se zde zabývat knihovnami, které jsou psané například pro Javu.

Dalším specifikem je úkol, na který chceme učení použít. Hraní her není jako ohodnocování datasetů, kde pro část dat známe požadovaný výstup. Na takových datech algoritmus naučíme a aplikujeme jen na zbylá neohodnocená data, abychom je ohodnotili. V případě hraní her pouze získáme skóre na konci každé hry, avšak ne každého tahu/rozhodnutí. Pro rozhodovací systém (ať už je implementován jakkoliv), si to můžeme představit tak, že hráč během hraní opakovaně provádí tahy podle výstupů tohoto systému, které se počítají ze vstupních proměnných. Avšak po každém jednom rozhodnutí není známo jaké rozhodnutí by bylo správné a systém se tedy nemůže ihned přeučit. Až na úplném konci se dozvídáme, zda celá hra byla úspěšná nebo nikoliv podle dosaženého skóre hráče.

Takže metody učení s učitelem jako je back propagation pro ANN a podobné lze také zavrhnout. Pro učení rozhodovacích systémů se zdá vhodná metoda simulované evoluce, která je použita i v motivační ukázce z úvodní kapitoly .

Hledání se zaměřuje na JavaScriptové knihovny pro strojové učení, které implementují simulovanou evoluci. Jak se lze dočíst například v článku [11] svět JavaScriptu nezůstává s knihovnamy pro strojové učení pozadu. Knihovny často obsahují nějakou implementaci neuronových sítí nebo rozhodovacích stromů, naleznou se i takové, které implementují algoritmy genetického programování (mezi které patří simulovaná evoluce). Nenašel jsem však jedinou, která by přímo implementovala, nebo se mi zdála vhodná k použití pro evoluci neuronových sítí pomocí rozšiřování topologie a šlechtění vah. Navíc vždy šlo pouze o Javascriptové knihovny, proto jsem se rozhodl implementovat vlastní

řešení od začátku v TypeScriptu.

Z přehledu [12] je vidět, že existuje mnoho uživatelských skriptů, přičemž některé z nich jsou takzvané hacky, které dávají hráčům nějakou výhodu, ale naleznou se i boti, jako například Agar.io-bot od GitHub uživatele Apostolique [13], který hraje tuto hru s velkým úspěchem [14]. Nejedná se však o učenlivého bota. Jeho chování je předem naprogramované. Předpokládám, že takovéto skripty budou vhodné k prostudování při implementaci vlastního automatického hráče.

Kapitola 3

Analýza

3.1 Strojové učení

Mezi hlavní cíle v rámci výzkumu umělé inteligence patří řešení problémů, uvažování, plánování, učení, reprezentace znalostí, zpracovávání přirozeného jazyka (komunikace), vnímání prostředí a další. Strojové učení (angl. machine learning) je rozsáhlá podoblast umělé inteligence.

Strojové učení se zabývá technikami, díky kterým se může počítačový program učit řešit problém, pro který nemá explicitně naprogramovaný řešič. Učením se rozumí taková změna vnitřního stavu, která zefektivní schopnost řešení dané úlohy. Jde o algoritmy, které se dokáží učit z dat a poté na nově předložených datech dělat různé předpovědi a rozhodnutí.

Tento obor má silnou vazbu na statistiku a vytěžování z dat (data mining), ale také na optimalizaci. Z toho vycházejí nejběžnější použítí:

regrese a predikce – odhadování nové hodnoty na základě již známých (např. předpovídat hustotu dopravy na základě dat z minulých let)

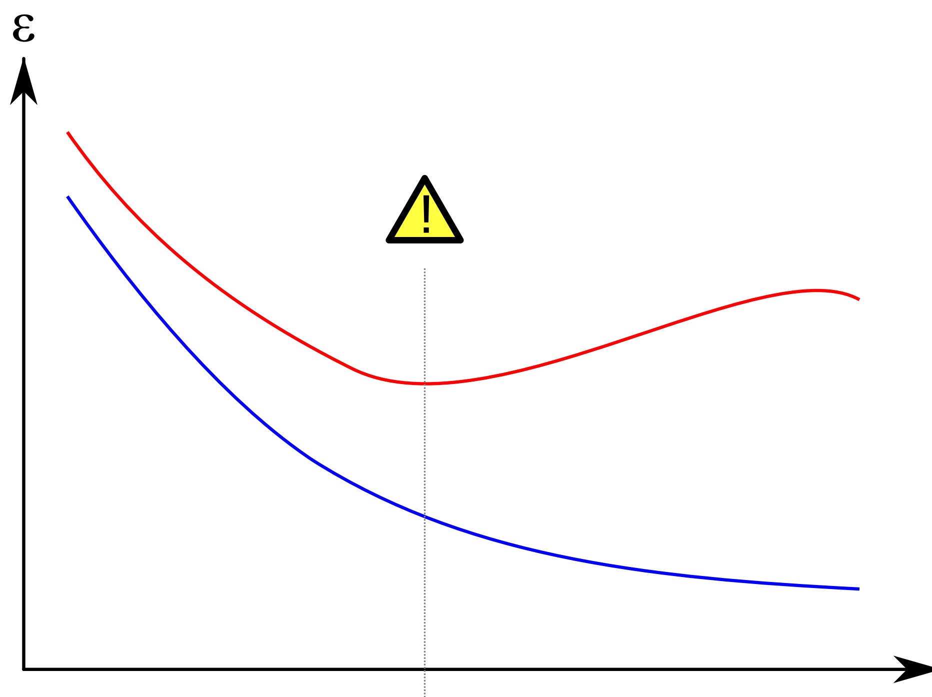
klasifikace – naučení modelu na datech, která jsou rozdělena do několika skupin, tento model poté dokáže rozdělit nová data do těchto kategorií (např. rozpoznávání spamu)

shlukování (clustering) – podobné klasifikaci, ale modelu je poskytnut pouze počet skupin, do kterých má data rozdělit podle jejich podobnosti, jde o typický příklad učení bez učitele (např. hledání podobností v neohodnocených datasetech)

rozhodování – vyhodnocování podnětů z prostředí a provádění akcí vedoucích k dosažení tíženého cíle (např. ovládání aut nebo *počítačových hráčů*)

dále i prohledávání stavového prostoru sloužící k optimalizaci, hledání asociačních pravidel, deep learning atd.

Uplatnění se nalezne při dobývání znalostí z dat, filtrování spamu, lékařské diagnóze, rozpoznávání řeči a obrazu, odhalování podvodů a průniků, učení



Obrázek 3.1: Graf zobrazující závislost velikosti chyby na čase učení pro dva datesety, modře chyba na trénovacích datech, červeně chyba na validačních datech. Označena hranice kdy dochází k přeučení.

systémů řešit problémy, plánovat a řídit, dále při automatickém ohodnocování datasetů, jako doporučovací systémy a další. [15, 16, 17]

3.1.1 Učení s učitelem a bez učitele

Při učení s učitelem (angl. supervised learning) má program k dispozici pro své učení ohodnocená data. Data jsou ve strukturované formě například jako vektor, tabulka, výstupy senzorů apod. To že jsou ohodnocená znamená, že expert pro všechna vstupní data určil správný výstup, jaký od programu očekává. Běžně se tímto způsobem řeší klasifikace a predikce.

Model se učí na dvou sadách dat - trénovacích a testovacích (validačních). Trénovací data se použijí, aby program naučil model. Model vyřeší problém a dostane zpětnou vazbu v podobě chyby, podle ní se pak může přeučit. Aby nedocházelo k tzv. přeučení (overfitting), kdy je model příliš přizpůsoben trénovacím datům, ale nedokáže správně určovat jiná, používá se i validační sada dat. Model naučený na trénovacích datech je spuštěn na validačních datech a sleduje se, kdy je chyba nejmenší na těchto validačních datech, ne na trénovacích, viz graf 3.1 [18].

Když model dokončí učící fázi na malé sadě dat, tedy je naučen, tak aby jeho chybovost byla minimální, nebo menší než daný práh, tak poté je používán na obvykle větší neohodnocené sadě dat (vybavovací fáze), kde se předpokládá že jeho chybovost je opět malá.

Zvláštním případem je zpětnovazební učení, také učení posilováním (reinforcement learning), kdy jedinou zpětnou vazbou modelu je přidělení konečného skóre. Toho je nejčastěji využíváno v agentních systémech. Agent zná akce, které může v prostředí provést. Jeho cílem je zvyšovat skóre. Tento přístup přesně odpovídá učení *automatických hráčů*. Bot odehraje hru, je mu přiděleno skóre a pokusí se přeučit tak, aby v dalším kole dosáhl skóre lepšího.

Na druhé straně stojí učení bez učitele. Během učení nedostává systém žádnou zpětnou vazbu, protože není znám správný výsledek. Jde o hledání podobností a struktury v neohodnocených datech. Obvyklou úlohou je shlukování a hledání asociačních pravidel.

Protože získávat ohodnocené datasety je náročné na čas i pracovní sílu, ale učení s učitelem produkuje lepší modely, lze používat kombinace učení s učitelem i bez něj. To je zajímavé především v dnešní době a do budoucna, protože dat přibývá a není je možné ohodnocovat ručně. Buďto se používají data, kde jich je ohodnocena pouze část, nebo se systém dotáže experta, aby pro něj ohodnotil data, která programu v učení pomůžou nejvíce. [15, 16, 17]

3.2 Simulovaná evoluce

Simulovaná evoluce je pokročilou učící metodou, která se inspiruje v přírodě biologickou evolucí. Jde o populační iterativní optimalizaci fungující na principu „šlechtění“ kandidujících řešení. Nepracuje pouze s jediným řešením, ale s celou populací, kterou prohledává stavový prostor naráz. Princip šlechtění vychází z výběru kvalitních jedinců k reprodukci. [19]

3.2.1 Názvosloví

Několik nutných pojmů:

genotyp (genotype, chromosome) - reprezentace řešení, běžně vektory (binární řetězec, DNA řetězec), stromy, neuronové sítě, gramatiky, apod.

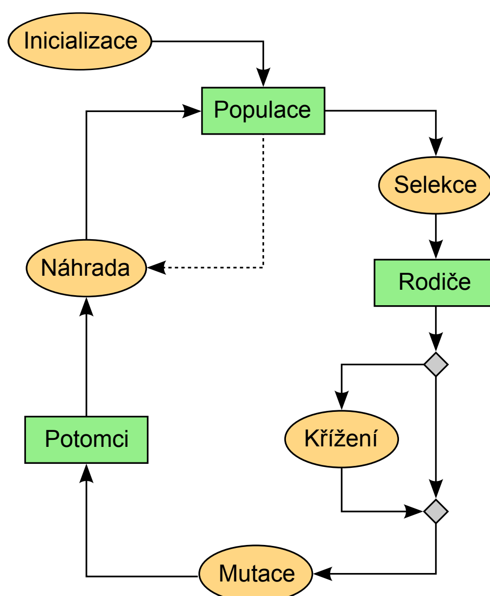
fenotyp (phenotype) - sémantika (význam) řešení, co genotyp kóduje,

zdatnost (fitness) - kriteriální funkce optimalizačního problému, jak dobré dané řešení je,

jedinec (individual) - kandidující řešení,

populace (population) - množina jedinců,

generace - populace po n krocích cyklu,



Obrázek 3.2: Obecné schéma evolučního cyklu.

reprodukce (křížení) - tvorba nových jedinců, většinou křížením nebo mutací,

potomci (offspring) - množina jedinců vzniklá reprodukcí, [19]

3.2.2 Schéma cyklu

Řešení je zakódováno v podobě binárního řetězce, stromu, neuronové sítě, apod. a nad takto zakodovaným řešením se opakovaně provádí selekce, křížení, mutace a konečně náhrada. Tyto kroky jsou obecné, jejich konkrétní implementace záleží na zvoleném postupu i kódování, ale co je důležité, **nejsou problémově závislé**. Mnoho různých problémů lze zakódovat stejně, například jako binární řetězec, a poté na ně lze použít stejný řešič. Pro případ této práce to znamená že je třeba implementovat jednu verzi evolučního algoritmy, která bude umět šlechtit DT nebo ANN, a všechny problémy (rozhodování botů), kódovat do této stejné struktury, kterou budu moci jednotně řešit.

Schéma cyklu je zobrazeno na 3.2 [19] a skládá se z těchto kroků:

selekce - výběr jedinců z populace na základě zdatnosti k jejich reprodukci,

křížení - vzájemná výměna informace mezi jedinci, konstrukce nového jedince,

mutace - drobná změna genotypu,

náhrada - náhrada jedinců v původní populaci jedinci novými, populace má obvykle stálou velikost. [19, 20]

3.2.3 Konvergence versus diverzita

Evoluční algoritmus má dvě složky. 1. konvergující k lokálnímu řešení (exploatace) ovládána selekčním tlakem (preference nejlepších řešení) a 2. složkou je explorace, která je zajišťována mírou mutace a diverzitou jedinců (globální složka) Poměr mezi oběma složkami je zásadní a je vhodné, aby se s časem měnil, tak aby mutace byla na počátku větší a dovolovala prohledávání, ale ke konci malá aby nepoškozovala zdatné jedince. Stejně tak selekční tlak, by měl být na počátku menší, aby dovoloval i horším řešením přežívat a na konci vetší, aby se zdatní jedinci maximálně zlepšili a našli lokální optima. [19, 20]

3.3 Rozhodovací stromy

Jak již bylo zmíněno možným přístupem při řízení automatických hráčů je využití *rozhodovacích stromů*. Rozhodovací strom je orientovaný kořenový strom a platí pro něj, že vnitřní uzly jsou rozhodovací funkcí, listy terminálem (třídou v rozhodovacího problému). Je komplexnější než lineární genotyp (např. binární řetězec) a také flexibilnější - i struktura řešení je předmětem evoluce. Výhodou rozhodovacích stromů je, že se rychle učí a jsou snadno pochopitelné pro člověka viz 3.4. Na druhou stranu je nutné toho více explicitně programovat. [21] Pro stomové struktury existují operátory mutace (různé výměny uzlu) i křížení (prohození podstromů) a lze je tedy šlechtit evolučním algoritmem. [19]

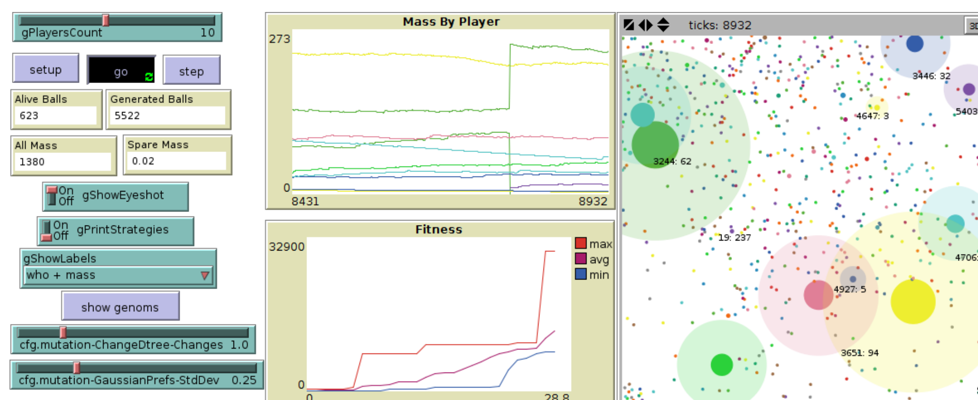
Tento přístup byl vyzkoušen v multiagentním programu NetLogo a je zde zmíněn spíše jako cesta, kterou se z důvodu neobecnosti nebylo vhodné vydat. Konkrétnější popis a výsledky lze najít v následujících podkapitolách.

3.3.1 Model Agar.io

V rámci analýzy byl vytvořen model hry Agar.io v aplikaci NetLogo, ukázku je možné vidět na obrázku 3.3. Byl zamýšlen jako „pískoviště“ pro zkoušení různých přístupů k řešení problému. Ale ukázalo se jako složité vytvořit přesný model, protože chování hry není nikde oficiálně (ani neoficiálně) dobře popsáno. Existují obecné poučky stylu „větší buňky rychleji ztrácí hmotnost“ nebo „menší buňku se pohybují rychleji“, [22] ale nejsou známy přesné funkce pro výpočet. Proto implementace modelu obsahuje konfigurační soubor, kde lze tyto vlastnosti nastavovat.

3.3.1.1 NetLogo

NetLogo je multiplatformní open-source prostředí pro programování v zabudovaném programovacím jazyku. Tento jazyk využívá předpřipravených agentů - turtles (pohybliví agenti - želvy), patches (nepohybliví agenti - políčka), links (spoje) a observer (pozorovatel) - aby modeloval přírodní i společenské



Obrázek 3.3: Výřez screenshotu z aplikace NetLogo, ve které běží model Agar.io. V levé části ovládací prvky, uprostřed graf hmotnosti hráčů a graf fitness populace, vpravo vizualizace světa. Na vizulizaci jsou vidět hráči jako barevná kolečka a jejich dohled jako průhledné barevné okolí. Popisek připojený v pravém dolním rohu každého hráče je ve tvaru <id>:<hmotnost>. Hráč 3244 je pronásledován hráčem 19, který je zároveň největším na mapě. Přestože hráč 4647 vznikl teprve nedávno, má nevhodně vyšlechtěný rozhodovací strom a jen stojí na místě a čeká na zánik.

systémy a jevy. Existují již implementované modely z oblasti fyziky, biologie, matematiky, informatiky, ekonomie, sociální psychologie a jiných.

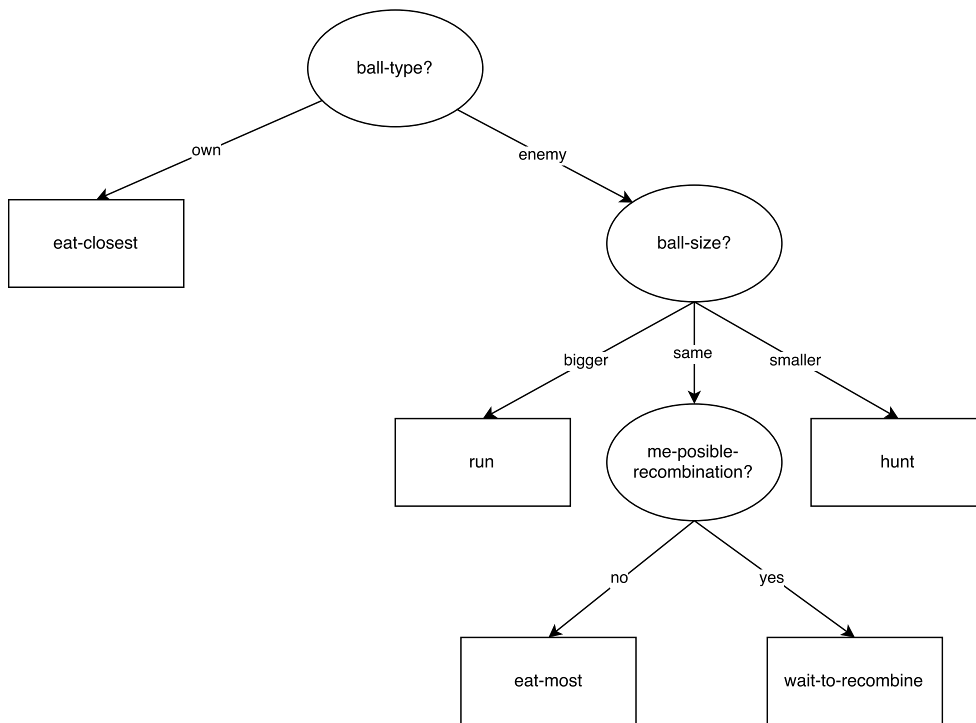
Každý agent provádí svou vlastní činnost ve společném světě, ve kterém navzájem všichni agenti iteragují. Řídí se instrukcemi, ty jsou napsány pomocí příkazů a reportérů (vrací hodnotu). V jazyku NetLogo existuje celá řada primitiv, zabudovaných příkazů a reportérů, díky nim lze modely snadno a rychle vytvářet.

V grafickém UI je možné vytvářet tlačítka, přepínače, posuvníky apod., kterými může pozorovatel zasahovat do simulace v reálném čase. Lze vytvářet i grafy a monitory agentů, kterými simulaci může sledovat ještě podrobněji, než jen na samotném zobrazení světa. [23]

3.3.2 Hráči Agar.io

V modelu existují hráči, kteří se řídí podle rozhodovacích stromů, například takového jako je na 3.4. Pro každou buňku v dohledu je učiněno rozhodnutí jaká strategii hraní na ni má být použita. Implementovaná rozhodovací pravidla jsou:

1. ball-size - rozhodnutí zda je buňka větší, menší nebo srovnatelně velká
2. ball-type - jde o vlastní buňku nebo nepřátelskou
3. ball-distance - je buňka na dostřel pro „výpad“



Obrázek 3.4: Příklad vyšlechtěného rozhodovacího stromu pro agenta v simulovaném prostředí NetLogo. Vnitřní uzly jsou zobrazeny jako elipsy a obsahují rozhodovací funkci. Listy stromu jsou zobrazeny jako obdélníky a určují strategii hry.

4. can-ball-split - může buňka zaútočit „výpadem“
5. me-single-celled - je hráč rozdroben na více buněk nebo ovládá pouze jednu
6. me-posible-recombination - uběhl dostatečný čas na to, aby se rozdrobené buňky mohly spojit opět v jednu

A možné strategie, které se většinou vztahují ke konkrétní buňce, pouze označené hvězdičkou (*) jsou obecné:

1. split-to-attack - zaútoč „výpadem“
2. hunt - lov (sleduj) buňku
3. run - utíkej před buňkou
4. split-to-escape - použij „výpad“ pro únik
5. wait-to-recombine - počkej na spojení buněk
6. eat-closest* - pohyb směrem k nejbližší potravě
7. eat-most* - pobyb směrem, kde je nejvíce potravy

Cyklus genetického algoritmu neprobíhá po generacích, ale k selekci, křížení a mutaci dochází po každé prohře (úmrtí) hráče. Pokud hráč zemře, zjistí se jeho zdatnost (skóre) a může být zařazen mezi mrtvé hráče. Udržuje se jen několik nejlepších mrtvých hráčů. V tuto chvíli na hracím poli chybí jeden hráč. Vytvoří se křížením dvou mrtvých hráčů vybraných pomocí ruletového výběru. Nakonec je hráč podroben mutaci a umístěn na hrací pole.

Šlechění podléhá jak samotný rozhodovací strom, tak vektor preferencí strategií, který slouží pro konečně rozhodnutí jaká strategie proti které buňce bude vykonána (nejpreferovanější a hmotnostně nejvýhodnější).

- Rozhodovací strom se kříží prohozením podstromů rodičů a mutuje náhodnou změnou uzlu (strategie za strategii, pravidlo za pravidlo se stejnou aritou).
- Vektor preferencí se kříží pomocí průměru z preferencí rodičů a mutuje pomocí odchylky z normálního rozdělení.

Většina hráčů na počátku není schopna se hnout z místa nebo růst. To je způsobeno především tím, že se snaží uplatnit strategii proti své vlastní jediné buňce a zůstanou stát na místě. Nebo proto, že jejich nejpreferovanější strategie není vhodná pro růst. Ale jak hráči umírají a šlechtí se noví, dochází k znatelnému pokroku. Roste maximální i průměrná fitness. A je vidět, že se šlechtí smysluplnější rozhodovací stromy, které například dokáží utíkat před predátory, nebo honí kořist, ale pokud není kořist/nepřítel v dohledu, snaží se sbírat jídlo.

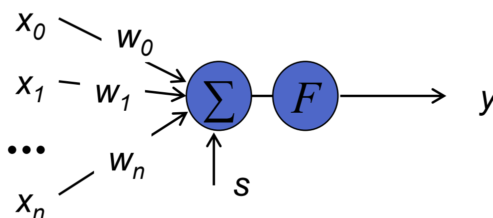
Samozřejmě ne každý nový hráč je lepší než předchozí. Pokud křížení nedopadne nejlépe, může se stát, že i potomek dvou dobrých rodičů má rozhodovací strom, který je horší než ty rodičů.

Výsledky se zdají být v pořádku, ale takto vygenerovaní boti by byly jen těžko schopni hrát proti lidským hráčům. To je z části pravděpodobně proto, že celé rozhodování je velmi jednoduché a z části proto, že učení probíhalo jen cca hodinu. Tento přístup by se také dal vylepšit o znalost minulého rozhodnutí, rozšířit rozhodovací pravidla i strategie.

Závěrem lze říci, že bylo dobré tento přístup vyzkoušet a zjistit jak pracný je. Nejen modelování hry v NetLogo, ale i samotná implementace hráčských strategií a rozhodovacích uzlů. Představa, že pro každý nový případ, na který má být knihovna použita, je nutné jej nejdříve zanalyzovat a poté implementovat konkrétní rozhodovací uzly a strategie, je odstrašující.

3.4 Umělé neuronové sítě

Druhou zmiňovanou možností, jak řídit automatické hráče je pomocí *umělých neuronových sítí* (ANN). Jde o model inspirovaný přírodou založený na propojení velkého množství malých výpočetních jednotek simulujicí mozek a jeho



Obrázek 3.5: Schéma neuronu ANN. Výstupy x_i jiných neuronů jsou vstupy tohoto neuronu. Ty jsou váženy váhami w_i. Před vstupem do aktivační funkce F jsou vážené vstupy sečteny včetně vychýlení s.

neurony. Neuronová síť je schopna přizpůsobovat spoje mezi neurony, tím se učí a zlepšuje své chování z hlediska dosahování cílů.

Druhů umělých neuronových sítí existuje poměrně velké množstí a lze je použít k různým účelům od klasifikace a shlukování přes rozhodování, řízení, rozpoznávání až po deep learning. Jsou vhodné pro složité nebo rozsáhlé úlohy, které by bylo obtížně explicitně naprogramovat. Pro svou obecnost je lze učit téměř cokoliv a často mohou objevit i řešení, které nebylo známé.

3.4.1 Umělý neuron

Umělý neuron, viz obrázek 3.5 [17], je základem každé ANN a chová se podobně jako lidský neuron. Přijímá výstupy několika jiných neuronů. Každému takovému vstupu je přiřazena váha. Neuron je zpracovává svou aktivační funkcí a vrací **jediný** výstup, který je přiveden i na několik neuronů. Tato aktivační funkce je ještě ovlivněna vychýlením (bias), někdy se také mluví o prahu (threshold), kdy je neuron aktivní. Neuron tedy reaguje na aktuální informace přívāděné na jeho vstupy z jiných neuronů podle naučených vah.

Výstup neuronu je vypočítán následovně: $y = F(\sum_{i=0}^{n}(x_i \cdot w_i) + s)$. Aktivační funkce F je jednoduchý výpočet, například sigmoid nebo jednotkový skok. Váhy w_i určují jak je daný vstup důležitý a bias je použit k vychýlení celé křivky (posun po ose x). Tyto hodnoty jsou předmětem adaptace při učení.

3.4.2 Učení ANN

Nejjedodušší sítí je jeden neuron se skokovou aktivační funkcí, také nazýván perceptron. Taková síť je schopna nanejvýš binární klasifikace dvou tříd - jeden řez rovinou. Problém XOR nebo klasifikace více tříd je nutné řešit pomocí více neuronů. Ty se většinou spojují po vrstvách jako je tomu například u vícevrstvého perceptronu (MLP).

Pro různé sítě existují konkrétní učící algoritmy. Pro perceptron to je Rosenblattův učící algoritmus, pro MLP zpětné šíření chyby. Umělé neuronové sítě lze ale učit i pomocí šlechtění, tzv. neuroevolucí. Jedincem je umělá neuronová síť o pevné struktuře s náhodně vygenerovanými vahami. Tyto

váhy jsou pak šlechtěny. Topology and Weight Evolving Artificial Neural Networks (TWEANNs) jdou ještě dále a evoluci podléhá i samotná struktura sítě. [17, 19]

3.5 NEAT

Tato kapitola pojednává o „neuroevoluci rozšiřováním topologie" (NeuroEvolution of Augmenting Topologies - NEAT) a čerpá především z práce K. O. Stanley a R. Miikkulainen [8] nazvané Evolving Neural Networks through Augmenting Topologies. V ní se věnují problematice konkrétního případu TWEANN, tedy evoluci neuronových sítí, při níž je podrobena evoluci váha přechodů i samotná struktura sítě.

Neuroevoluce s pevnou topologií šlechtí pouze váhy a struktura sítě je pevně dána od počátku experimentu - například jedna skrytá vrstva neuronů plně propojena se vstupy a výstupy. Evoluce v takovém případě prohledává prostor vah spojení neuronů. Avšak nejenom váhy spojení v NN ovlivňují její chování. Také na struktuře záleží její funkcionalita.

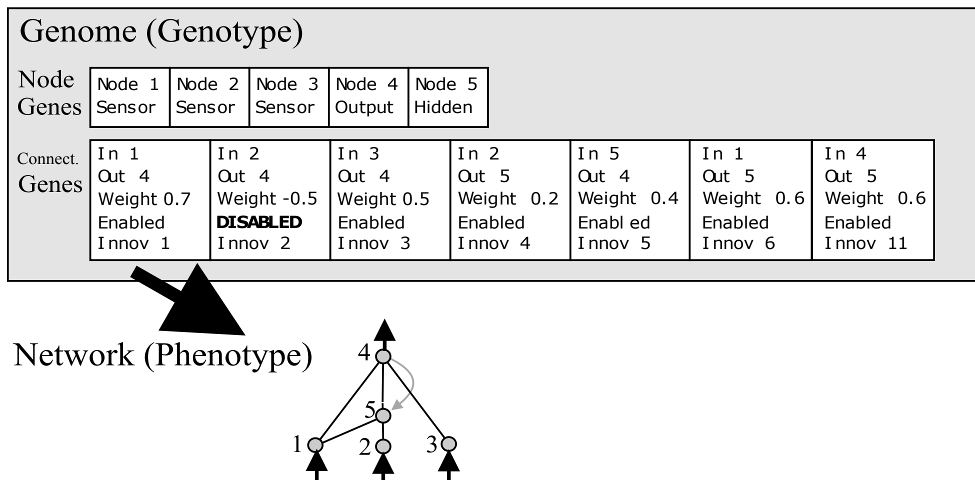
Otázkou je, zda má smysl tuto strukturu také šlechtit, když plně propojená síť může aproximovat jakoukoliv spojitou funkci. Jak výsledky studie o NEAT ukázaly pokud je neuroevoluce obojího, tedy vah i topologie, udělána správně může výrazně přispět k výkonu celého systému. Protože v NEAT algoritmu síť roste postupně z pouze vstupních a výstupních neuronů, tak má tendenci být minimalní po celou dobu evoluce, což zjednodušuje prostor vah, ve kterém je nutné hledat optimum. Také odpadá potřeba zásahu člověka, kdy musí vymyslet strukturu sítě, případně restartování stroje s novým a novým počtem skrytých uzlů.

NEAT naráží na 3 problémy, které úspěšně řeší:

1. existuje reprezentace NN taková, že snadno umožní různým topologiím smysluplné křížení,
2. jak ochránit nové struktury, které potřebují několik generací k optimalizaci, aby z populace nevymizely předčasně,
3. jak udržovat topologie minimální v průběhu evoluce bez nutnosti speciálních úprav.

3.5.1 Kódování fenotypu

Jak je znázorněno na obrázku 3.6 [8] každý genom obsahuje seznam uzlů/neuronů (node genes), které mohou vytvářet spoje, a jejich typ - vstupní, skrytý, výstupní. Dále obsahuje seznam hran/spojů (connection genes), přičemž každý z těchto spojů odkazuje na dva neurony, které spojuje. Hrany specifikují vstupní a výstupní neuron, váhu spoje, historické (inovační) číslo a



Obrázek 3.6: Příklad kódování fenotypu (neuronové sítě) v NEAT. Uvedeny jsou 3 vstupní neurony, jeden skrytý a jeden výstupní. Dále 7 spojů, přičemž jeden je zpětný (mezi uzlem 4 a 5) a jeden neaktivní (mezi uzlem 2 a 4), takže se ve fenotypu neobjevuje.

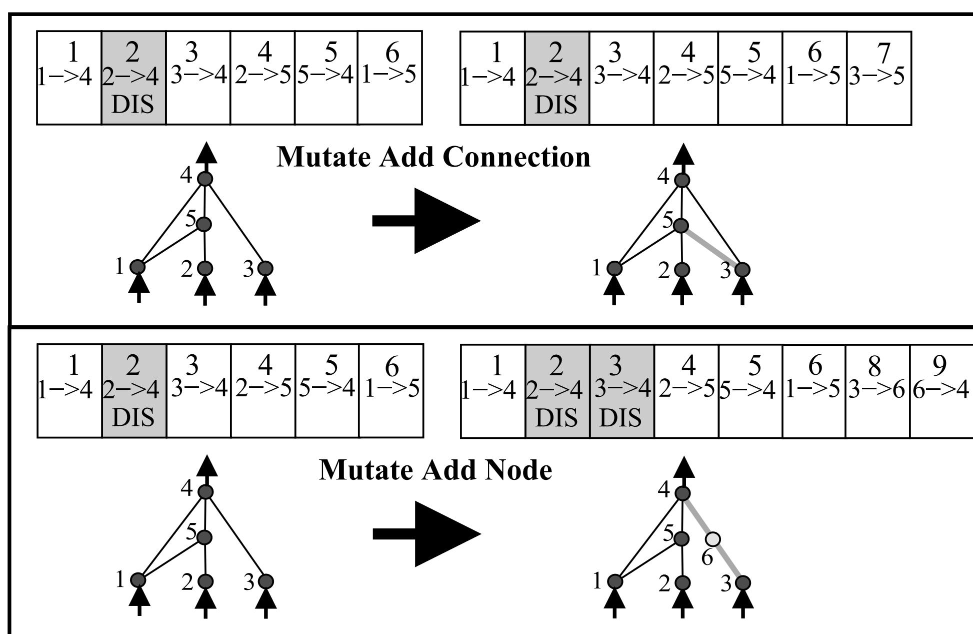
zda je spoj aktivní. Inovační číslo je konstrukt, který NEAT zavádí, aby označil historický původ jednotlivých genů. Novým spojům je uděleno unikátní vzrůstající číslo. Toto číslo nalezne využití při křížení i mutaci.

3.5.2 Mutace

Mutace mění váhy spojů neuronové sítě, tak jak je to běžné u neuroevoluce. Každý spoj každého jedince s určitou pravděpodobností mírně změní svoji váhu. NEAT také představuje dva druhy mutací struktury sítě, skrze které síť postupně roste. Jsou vyobrazeny na obrázku 3.7 [8].

Mutace přidáním spojení přidává jeden nový gen - spoj s náhodnou váhou mezi dva dříve nespojené neurony. Mutace přidáním uzlu přídává 3 nové geny (2 spoje, 1 uzel) a původní spoj je deaktivován. Novému spoji vedoucímu do nového uzlu je udělená váha 1, druhému spoji vedoucímu z nového uzlu je udělena váha původního spoje. Tímto stylem má přidání nového uzlu pouze minimálním efekt na funkčnost neuronové sítě, ale geny mají možnost se optimalizovat a zdokonalit jedince.

Všimněme si také inovačního čísla spojů. Spoji přidánému v první mutaci je přiřazeno nové vyšší inovační číslo 7. Dva spoje přidané v druhé ukázce mají inovační čísla 8 a 9. Kdykoliv v budoucnu, pokud se budou tyto jedinci křížit, jejich potomci dostanou si ponesou tyto inovační čísla dále - inovační číslo genů se nikdy nemění. Tím je zajištěno, že historický původ je znám po celou dobu evoluce.

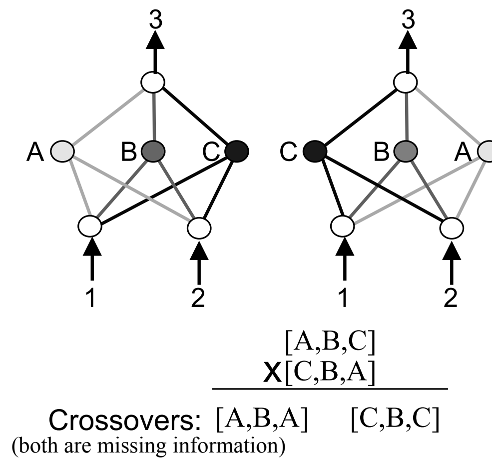


Obrázek 3.7: Příklady dvou druhů mutace struktury NEAT neuronové sítě zobrazené pomocí seznamu spojení a fenotypu, který kódují. Spoj má přiřazené inovační číslo, vstupní a výstupní uzel a informaci, zda je aktivní. (1) Při mutaci přidáním spoje je tento spoj jednoduše přidán do seznamu spojů. (2) V případě mutace přidáním uzlu je spojení přerušeno (deaktivováno) a nahrazeno dvěma novými spojeními a novým uzlem.

3.5.3 Křížění

Competing Conventions Problem neboli Permutations Problem je problém vznikající z faktu, že obecně pro TWEANN existuje více než jeden způsob, jak vyjádřit stejné řešení. Pokud se kříží dva jedinci s genomy reprezentující podobné řešení, ale jejich genomy nejsou kompatibilní, jejich potomek bude pravděpodobně méně schopný. Představme si situaci z obrázku 3.8 [8], kde je vidět, že z křížení vznikl potomek kterému chybí $\frac{1}{3}$ informace. V reálu nastávají složitější situace, protože NN sítě mohou reprezentovat podobná řešení pomocí úplně odlišných topologií, různých velikostí.

NEAT vychází z toho, že chromozomy mohou mít různou velikost, geny na stejném místě reprezentovat jinou vlastnost nebo geny na různých místech reprezentovat vlastnost stejnou. Z toho vychází potřeba umět zarovnat geny k sobě, tak aby se spolu křížily sobě si odpovídající neurony. Toho je těžké dosáhnout pouhou analýzou struktury, proto NEAT představuje “historický původ” (historical origin, innovation number) neuronu. Díky čemuž dokáže sledovat, které geny (neurony) jsou kompatibilní mezi jakýmikoliv jedinci v populaci.



Obrázek 3.8: Příklad permutačního problému. Dvě plně propojené neuronové sítě, které jsou reprezentovány jiným chromozomem (neurony v jejich skryté vrstvě se vyskytují v jiném pořadí), vrací stejný výsledek. Dvěma z šesti možných křížení (permutací) však chybí vnitřní uzel A nebo C.

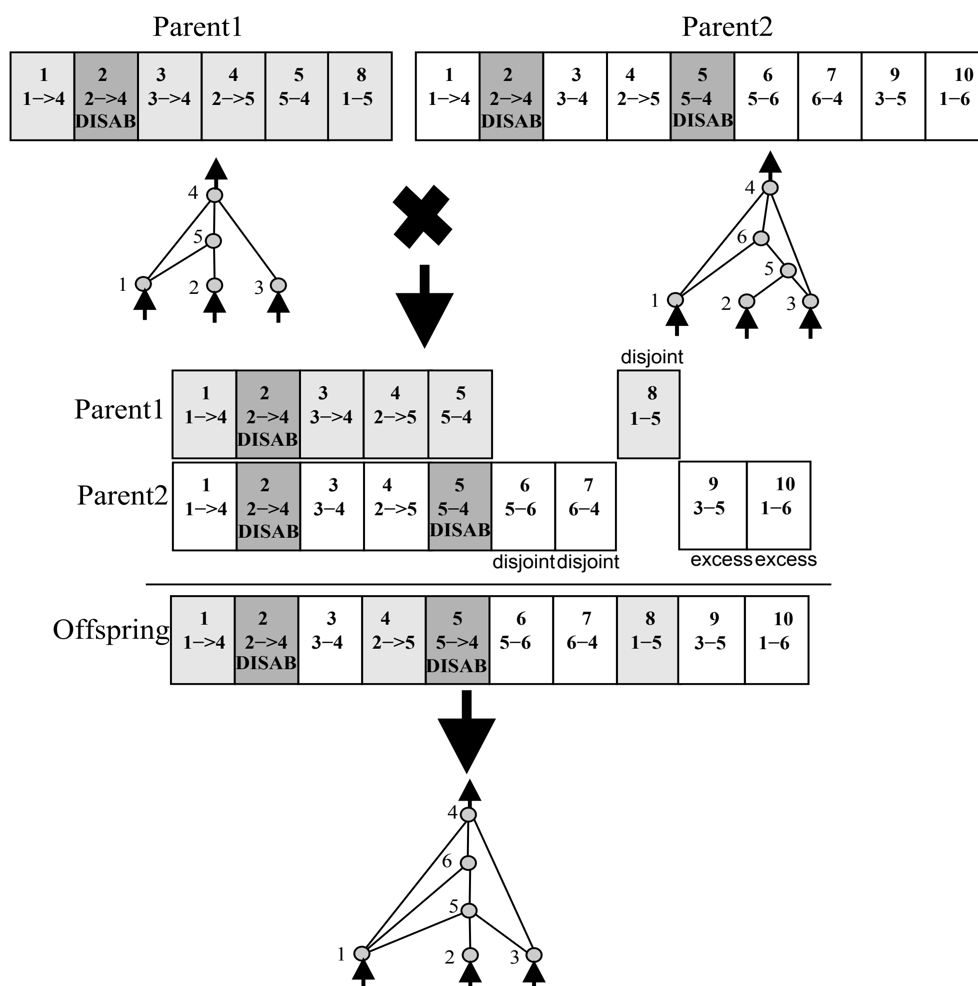
Jak je vidět na obrázku 3.9 [8], při křížení jsou geny seřazeny a vzniknou tři skupiny. Geny které si inovačním číslem odpovídají v obou rodičích (matching genes), geny disjunktní (disjoint genes) s hodnotou inovačního čísla menší než hodnota jiného inovačního čísla druhého potomka, nakonec zbývající geny (excess genes) z potomka s mladšími (vyšší inovační číslo) geny. Tento způsob umožňuje křížit potomky kódované lineárním genomem (viz. Encoding) bez nutnosti složité analýzy topologie. Potomek zdědí náhodně geny ze spárovaných genů, geny, které si neodpovídají, zdědí ze zdatnějšího rodiče.

3.5.4 Druhy

Inovace vznikají mutací, kdy je do struktury přidán nový neuron nebo spoj. Tyto změny většinou vedou v počátku ke snížení fitness zmutovaného jedince. Je třeba, aby váhy spojení měly čas se optimalizovat, což vyžaduje několik generací. Je však nepravděpodobné, že taková inovace tyto generace přežije, právě kvůli zhoršení fitness na počátku.

NEAT zavádí ochranu inovací pomocí izolace druhů. Této technice se říká speciation nebo niching a je k ní potřeba funkce kompatibility chromozomů, která určí, kteří jedinci spolu patří do jednoho druhu. Nalézt takovou funkci obecně pro TWEANN je složité, ale konkrétně pro NEAT, který zavádí původ neuronu, toho lze docílit. Dále je použito sdílení fitness (fitness sharing) pro stejný druh. Výsledkem toho je, že v populaci existují druhy, které seskupují jedince s podobnou strukturou NN, ale žádný druh nemůže přerůst a převzít vládu nad celou generací.

Rozdělování jedinců do druhů (podle topologické podobnosti) zajišťuje, že



Obrázek 3.9: Příklad párování genů a křížení jedinců s využitím inovačního čísla. Rodiče vypadají rozdílně, avšak díky inovačním číslům jednotlivých genů víme, které geny lze párovat s geny z druhého rodiče. Odpovídající si geny jsou zděděny náhodně, disjunktní a přebývající geny potom ze zdatnějšího rodiče. V tomto případě předpokládáme stejnou fitness obou rodičů a tak i neodpovídající geny jsou zděděny.

jedinci ze stejné oblasti prohledávaného prostoru (niche) soupeří pouze spolu, raději než s celou populací. Tak je zaručeno, že topologické inovace přežívají ve společném druhu, kde soupeří mezi sebou a mají čas se optimalizovat. Jedinci jsou shlukování přes topologickou podobnost na základě historického čísla. Počet neodpovídajících si genů (disjoint a excess) je vhodným měřítkem podobnosti. NEAT používá lineární kombinaci disjunktních (D) a zbylých genů (E) a také průměrné odchylky vah shodných genů ($\overline{W}$). Koeficienty $c1, c2, c3$ určují váhu jednotlivých složek a N je počet genů v mohutnějším z jedinců.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$

V každé generaci se jedinci postupně řadí do druhů, tak že jedinec je zařazen pod první druh, se kterým je kompatibilní (podle předepsané vzdálenostní funkce). Druh je reprezentován náhodným jedincem z předchozí generace. Pokud není jedinec kompatibilní s žádným druhem, je vytvořen nový druh. Druhy se nikdy nepřekrývají.

Aby jeden druh nepřerostl, jedinci z něj sdílí fitness mezi sebou. Takže ani druh, ve kterém si všichni vedou dobře, se nemůže stát příliš velkým, protože s každým novým jedincem se sdílená fitness sníží. Čitatel výpočtu upravené fitness pro jedince je výsledek klasické funkce zdatnosti, jmenovatel pak počet jedinců v jeho druhu.

$$f_i' = \frac{f_i}{|specie_i|}$$

Počet potomků v druhu je proporcionální ke sdílené fitness. Noví jedinci nahrazují nejméně zdatné z celé populace. Jedinci, kteří zbydou jsou považování za novou generaci. V případě, že se fitness celé generace po mnoho generací nezlepší, je možné nechat reprodukovat jen 2 nejlepší druhy, což vede k zaměření hledání ve slibné oblasti prohledávaného prostoru.

3.5.5 Počáteční populace

V genetických algoritmech je většinou počáteční populace náhodně vygenerována, v případě TWEANN jedinci s náhodnou strukturou neuronů a vah spojení. Avšak NEAT se snaží vyvinout minimální řešení a tak je pro tento algoritmus vhodné začínat z minimální sítě pouze vstupů a výstupů - žádné skryté neurony. Struktura roste z minimální sítě evolucí v generacích a systém tak hledá řešení v prostoru vah s nejmenší možnou dimenzionalitou (prohledávany prostor je co nejmenší) po celou dobu evoluce. To odpovídá řešení třetí otázky z úvodu podkapitoly.

3.5.6 Výsledky/závěry práce

Jak vyznívá z experimentů a analýzy popsaných v kapitolách 4 a 5 práce K. O. Stanleyho a R. [8], NEAT umí úspěšně rozvíjet topologie sítě, pokud to

problém vyžaduje. Navíc se na benchmarkovacím problému vyvážení převráceného kyvadla (více v o problému v kapitole 5.1), ukázalo že je skutečně výkonný, výkonnější než porovnávané přístupy.

NEAT šlechtí mnoho rozdílných struktur naráz v různých druzích, každý reprezentující jinou část prohledávaného prostoru. Takže v jednu chvíli se snaží řešit problém hned několika rozdílnými přístupy. To pomáhá, aby nedošlo k předčasné konvergenci, ale také, aby dokázal řešit složité rozhodovací úlohy. To se ukázalo nejen na experimentech v [8], ale i při aplikaci NEAT algoritmu pro učení hry Mario viz 2. Proto se zdá být neuroevoluce umělých neuronových sítí podle NEAT vhodná pro účel této práce. A z tohoto důvodu bude použita pro **vlastní řešení** zadaného problému.

3.6 JavaScript a TypeScript

Hra Agar.io běží ve webovém prohlížeči a klient je napsaný v JavaScriptu. JavaScript (js) je multiplatformní interpretovaný jazyk nejvíce známý jako skriptovací jazyk pro web, ale používá se i mimo internetové prohlížeče. Jde o objektově orientovaný, dynamicky typovaný jazyk. Postrádá třídy, místo nich je využíváno prototypování, čímž lze docílit stejných výsledků (zapouzdření, dědění, atd). S funkcemi se zachází jako objekty (first class functions) - mají vlastní atributy, lze je předávat jako parametr a vracet z funkcí. Napsaný kód se vykonává na straně klienta, a proto z bezpečnostních důvodů nemůže přistupovat k souborovému systému na disku ani k systémovým objektům. Přesto jde o plnohodnotný programovací jazyk, nejen o nástroj pro design stránky. [24]

TypeScript (TS) je nadmnožinou JavaScriptu a je to jazyk pro psaní rozsáhlých JS aplikací. Kompiluje se do čistého js, který je poté standartním způsobem distribuován a spouštěn. Obohacuje JS kromě jiného především o volitelné typy, třídy (rozhraní, dědičnost, atd.) a moduly, čímž práci s JS přibližuje třídním jazykům jako je Java. [25]

To že je Agar.io napsaný v JavaScriptu znamená, že zdrojový kód je poslán do prohlížeče a na něm interpretován. Tento kód lze rozšířit dopsáním vlastního kódu, například o zvýrazňování nepřátel na hrací ploše nebo i automatického hráče. A proto JS/TS bude použít jako programovací jazyk pro vývoj výsledné aplikace.

3.7 Rozšířené prohlížení

Rozšířené prohlížení (augmented browsing) je klíčovou technologií pro napsání zamýšlené aplikace. Jde o upravování a vylepšování webu (poskytovaných informací, vzhledu) na straně klienta, tak aby uživateli web poskytl lepší zážitek z prohlížení. [26] Různé ad-block doplňky do prohlížečů pro blokování reklamy jsou jedním možným příkladem. Dalším například je automatické překládání

stránek přímo v prohlížeči. Pro vyvíjenou aplikaci je to vložení vlastního JS kódu, který bude za člověka ovládat hru.

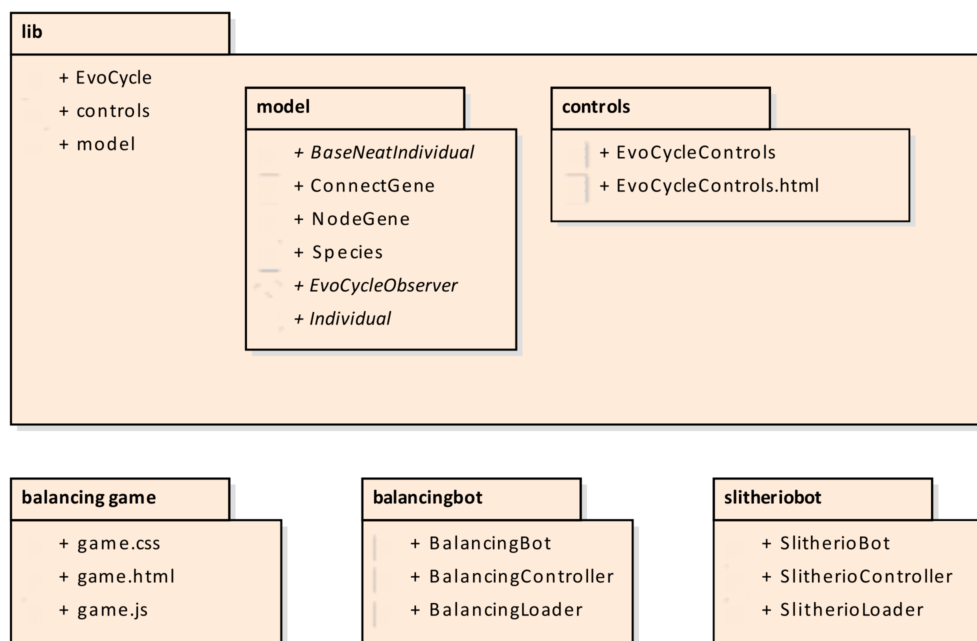
Vyvíjet aplikaci jako samostatný addon do prohlížeče se zdá zbytečné. Je třeba pouze vkládat vlastní JS, HTML a CSS kód do existující stránky. V minimalistické formě si lze vystačit pouze s developerskou konzolí, která je dostupná pro IE, Chrome i FF. V ní lze napsat a spouštět jakýkoliv JS kód. Pro složitější JS aplikace je lepším řešením využít script manageru a takové addony již existují - Tampermonkey pro Chrome, Greasemonkey pro Firefox, a další.

Kapitola 4

Návrh a implementace

Jak je rozvedeno již v kapitole 3.3.1, jako první v rámci analýzy vznikl model hry Agar.io v multiagentním systému NetLogo. Automatičtí hráči činili rozhodnutí na základě rozhodovacích stromů a zlepšovali se pomocí evolučního algoritmu. Ve své podstatě toto řešení vedlo k funkčním učenlivým agentům, ale nebylo příliš obecné a aplikovatelné na další případy aplikací a her. Pro tento přístup je nutné ručně psát možné uzly rozhodovacích stromů pro každý konkrétní případ.

Pro konečné řešení bylo vybráno, jak již může být patrné z kapitoly 3.5, řešení pomocí neuronových sítí. Po zbytek kapitoly se tedy zabývám návrhem



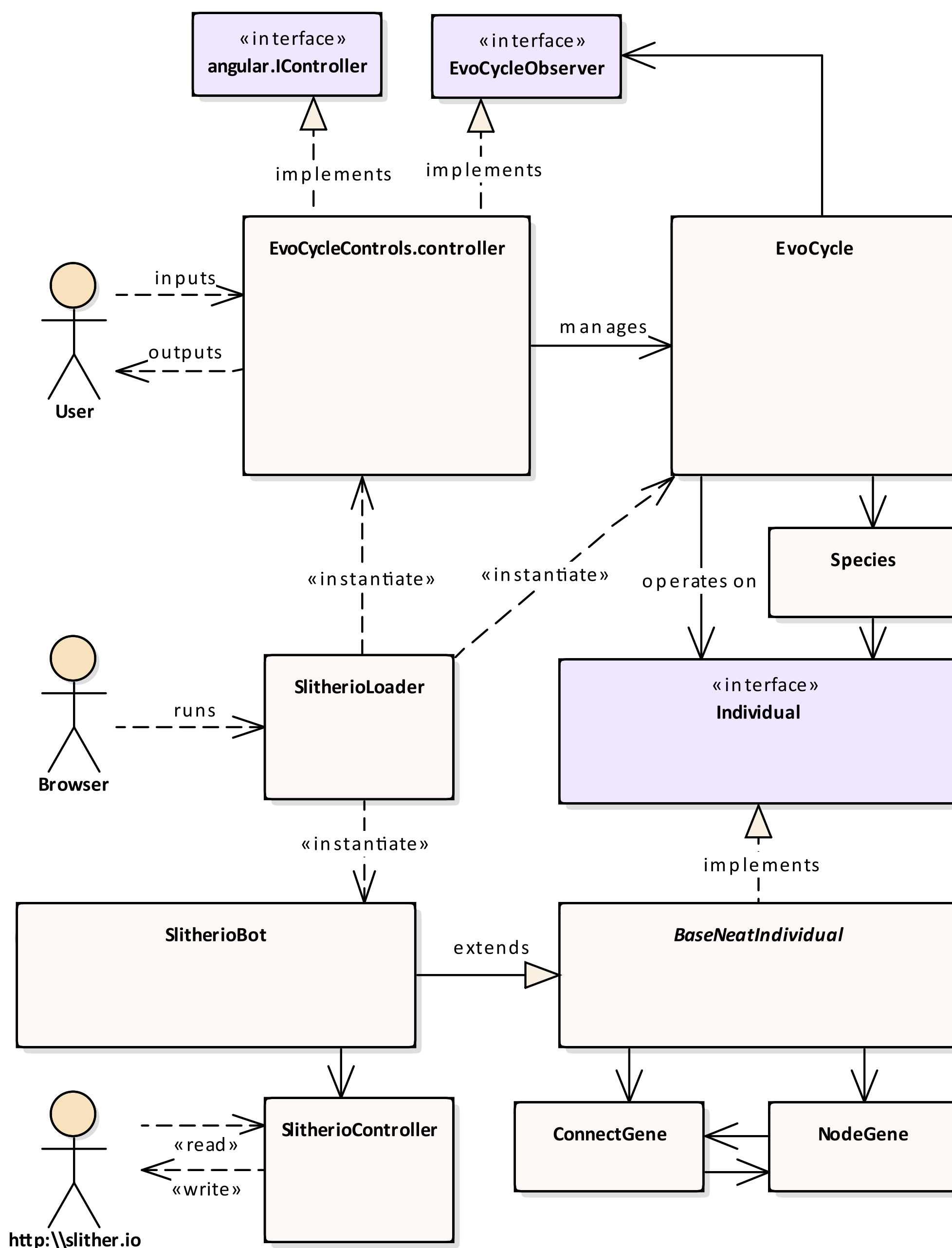
Obrázek 4.1: Diagram rozdělení tříd do balíčků

a implementací NEAT algoritmu aplikovaného pro učení botů hraní her ve webovém prohlížeči.

Protože jde o JavaScriptovou aplikaci, musím to brát při návrhu a implementaci v úvahu. JavaScript má svá specifika. TypeScript pomáhá s typy, třídami, rozhraními, děděním apod. Ale jedná se stále o interpretovaný jazyk v prohlížeči, který má k dispozici jediné vlákno které nemůže blokovat. Proto se využívá asynchroních volání, callbacků, promise a timerů.

Nejdříve implementuji třídy entit, které navrhuji podle kapitoly 3.5. Tyto třídy slouží jako model, nad kterým bude pracovat třída zajišťující evoluci jedinců. Nakonec naimplementuji automatické hráče jako konkrétní implementace jedinců. Rozdělení do balíčků je vidět na diagramu 4.1. Celkový pohled na zamýšlené fungování je v minimalistické podobě shrnut na diagramu 4.2. V dalších kapitolách uvádím podrobný návrh a popis jednotlivých tříd. Využívám **tříd** a jejich **dědění**, **implementací interfaců** a dalších vlastností **objektově orientovaného** návrhu.

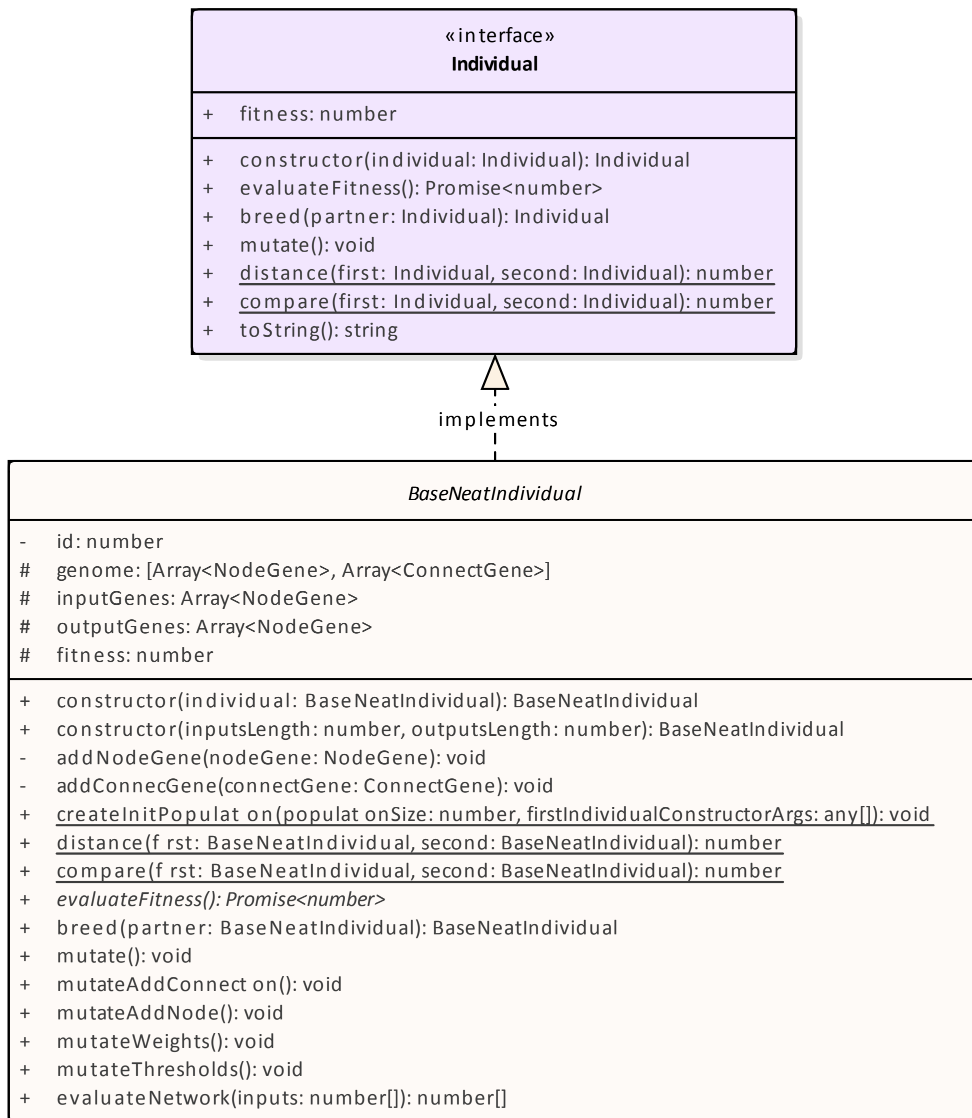
Při návrhu používám program Sparx Systems Enterprise Architect [27] a webovou aplikaci draw.io [28]. Protože je implementačním jazykem TypeScritp, ve všech diagramech je použita jeho notace. Kód píši v IDE JetBrains IntelliJ IDEA [29]. Testuji a experimenty spouštím v prohlížečí Google Chrome [30].



Obrázek 4.2: Celkový pohled na propojení a fungování tříd.

4.1 Model

4.1.1 BaseNeatIndividual



Obrázek 4.3: Diagram třídy `BaseNeatIndividual` a závislostí.

Začnu odprostřed a to entitou jedince, která je z balíčku `model` nejdůležitější. Vytvořil jsem rozhraní, které každý jedinec musí splňovat, aby ho třída EvoCycle byla schopná používat. Grafické znázornění na diagramu 4.3

`fitness` - vypočtené fitness pro výpis v `EvoCycleControls` a také pro rozhodnutí, zda je nutné fitness počítat nebo je již známé z předchozí generace.

`evaluateFitness` - výpočet fitness. Návratovou hodnotou je **`Promise`**, jejímž vyřešením je číslo. Je to slib, že někdy v budoucnu bude tato hodnota dostupná. Jde o šikovné JavaScriptové řešení asynchronních dotazů, bez nutnosti předávání callback funkce nebo implementování Observer patternu. Předpokládám, že zavoláním této funkce jedinec spouští řadu událostí v prohlížeči, které jsou asynchronní (např. `setInterval` pro učinění rozhodnutí, jak se chovat každých 100ms). A až ve chvíli, kdy jedinec dokončí svůj úkol, je možné spočítat zdatnost a vrátit ji. V tuto chvíli může volající pokračovat ve svých výpočtech s touto hodnotou. Ale není blokován samotným výpočtem fitness.

`constructor` - copy construktor pro vytváření mutací jedince.

`breed` - funkce křížení vracející nového potomka.

`mutate` - funkce mutace nad tímto jedincem.

`distance` - statická funkce počítající rozdílnost jedinců pro zařazení do druhů.

`compare` - statická funkce porovnávající zdatnost jedinců.

`toString` - přetížená metoda toString pro výpis v `EvoCycleControls`.

Jedinou implementací rozhraní je abstraktní třída `BaseNeatIndividual`, viz 4.3. Třída implementuje celou logiku NEAT, pouze metoda `evaluateFitness` je ponechána abstraktní, protože je problémově závislá a musí jí každý bot implementovat sám.

`constructor` - dva konstruktory. Jeden klasický konstruktor, který vytvoří náhodného jedince podle zadaného počtu vstupních a výstupních uzlů. Půjde o plně propojený graf s náhodnou hodnotou vah hran a hodnot prahu aktivační funkce, viz dále. Druhý je copy construktor, který je hojně využíván při mutacích a křížení jedinců.

`createInitPopulation` - pro NEAT je třeba první generaci vytvořit tak, aby všichni jedinci měli stejná inovační čísla na vstupních a výstupních uzlech a na hranách mezi mezi nimi.

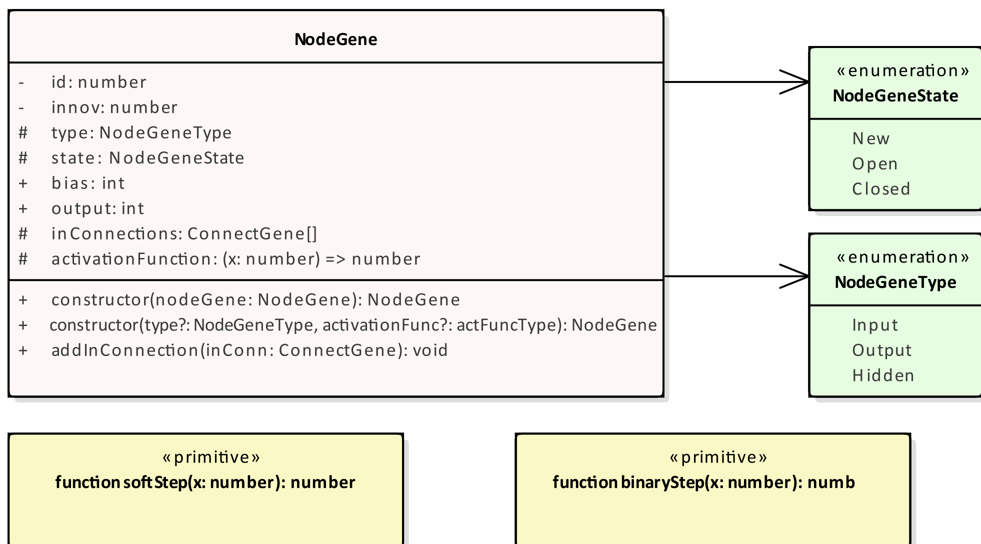
`evaluateFitness` - abstraktní metoda, která však předpokládá volání funkce `evaluateNetwork` pro činění rozhodnutí.

`evaluateNetwork` - grafový problém vyhodnocení výstupů. Při výpočtu využívá polí `inputGenes`, které vyhodnotí jako první (konečné podmínky) a rekurzivního volání `evaluateOutput` nad polem uzlů `outputGenes`.

`breed` - implementace křížení podle kapitoly 3.5

`mutate*` - implementace mutací podle kapitoly 3.5 a mutací vah.

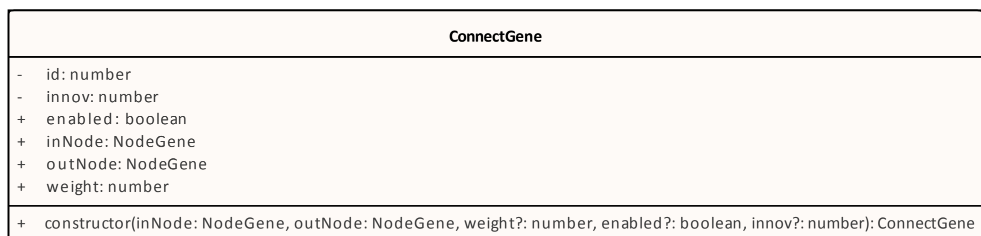
4.1.2 NodeGene



Obrázek 4.4: Diagram třídy `NodeGene` a závislostí.

Třída `NodeGene`, viz diagram 4.4, představuje základní kámen neuronové sítě – uzel grafu. Kromě id a inovačního čísla nese dále informaci o typu uzlu (v jaké vrstvě se nachází), aktivační funkci a hodnotu prahu při které neuron takzvaně pálí. Pro výpočet hodnoty výstupu je déle třeba znát hodnoty vstupních hran. Metoda `evaluateOutput` je volána rekurzivně na uzly na začátku těchto hran. Proto je zaveden stav uzlu, který přeruší případný cyklus v grafu a vrátí hodnotu vypočítanou v minulém volání `evaluateNetwork` viz. 4.1.1. Copy construktor je využíván ze třídy `BaseNeatIndividual`. Sám o sobě nekopíruje hrany na tento uzel. O to se stará volající metoda.

4.1.3 ConnectGene



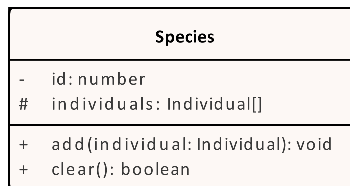
Obrázek 4.5: Diagram třídy `ConnectGene`

Třída `ConnectGene`, viz diagram 4.5, představuje hranu uzlu. Stejně jako `NodeGene` má id a inovační číslo. Přičemž inovační číslo je zde nutné pro kří-

žení, u uzlu je pouze z praktických důvodů při ladění a výpisech. Hrana musí vědět, které dva uzly spojuje a jakou má váhu, navíc může být neaktivní. Hrany se z jedinců nikdy nemažou, pouze se vypnout a mají v budoucnu šanci být opět zapnuty při křížení nebo mutaci.

Nemá copy konstruktor, protože není třeba. Když se klonuje jedinec, nejdříve se naklonují uzly pomocí copy konstruktoru, ale poté je potřeba vytvořit spoje mezi těmito novými uzly. Takže jsou vytvářeny úplně nové spoje mezi těmito novými uzly, které jsou obrazy spojů v původním jedinci. Za povšimnutí stojí, že v constructoru je i nepovinný parametr `innov`. Nepovinný je v případě, že vzniká nová hrana mutací, povinný když děláme obraz spoje v novém jedinci. Takové spoje musí mít stejné inovační číslo.

4.1.4 Species

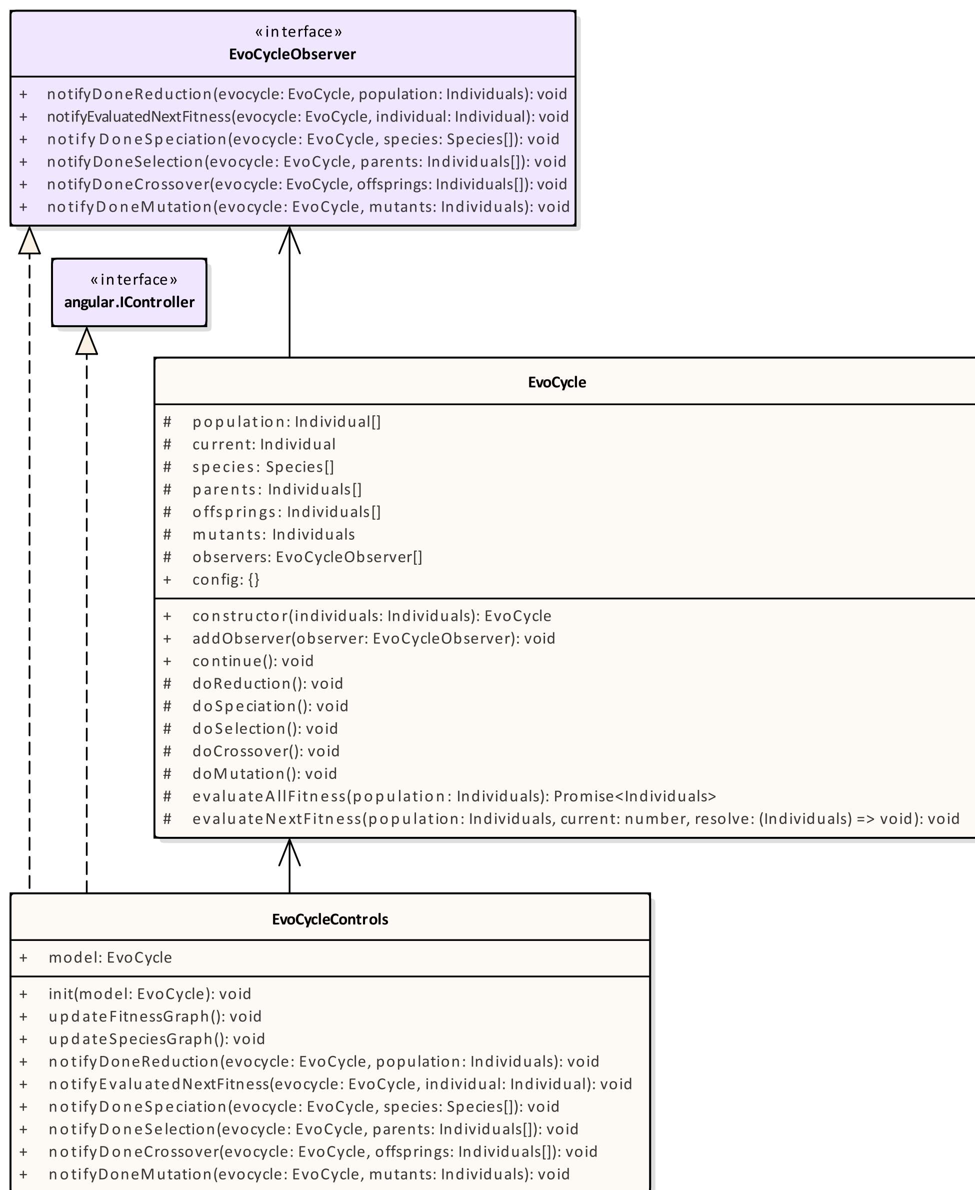


Obrázek 4.6: Diagram třídy `Species`

Jak je vidět z diagramu 4.6, toto je velmi jednoduchá třída sloužící jako kontejner pro jedince, kteří patří do stejného druhu. Zajímavá k vysvětlení je pouze metoda

`clear` - nechá v druhu pouze jednoho jedince jako representanta, podle kterého se v další generaci bude určovat vzdálenost a tím příslušnost do daného druhu.

4.2 EvoCycle



Obrázek 4.7: Diagram třídy `EvoCycle` a závislostí.

Když je navrhnutý model, je možné pokračovat s jádrem knihovny. To bude implementováno třídou `EvoCycle`, viz dia:EvoCycle. Ta implementuje celý evoluční cyklus, tak jak jsem jej popsal v kapitole 3.2.

`population` - kontejner pro zpracovávané jedince.

species - druhy do kterých jedinci patří.

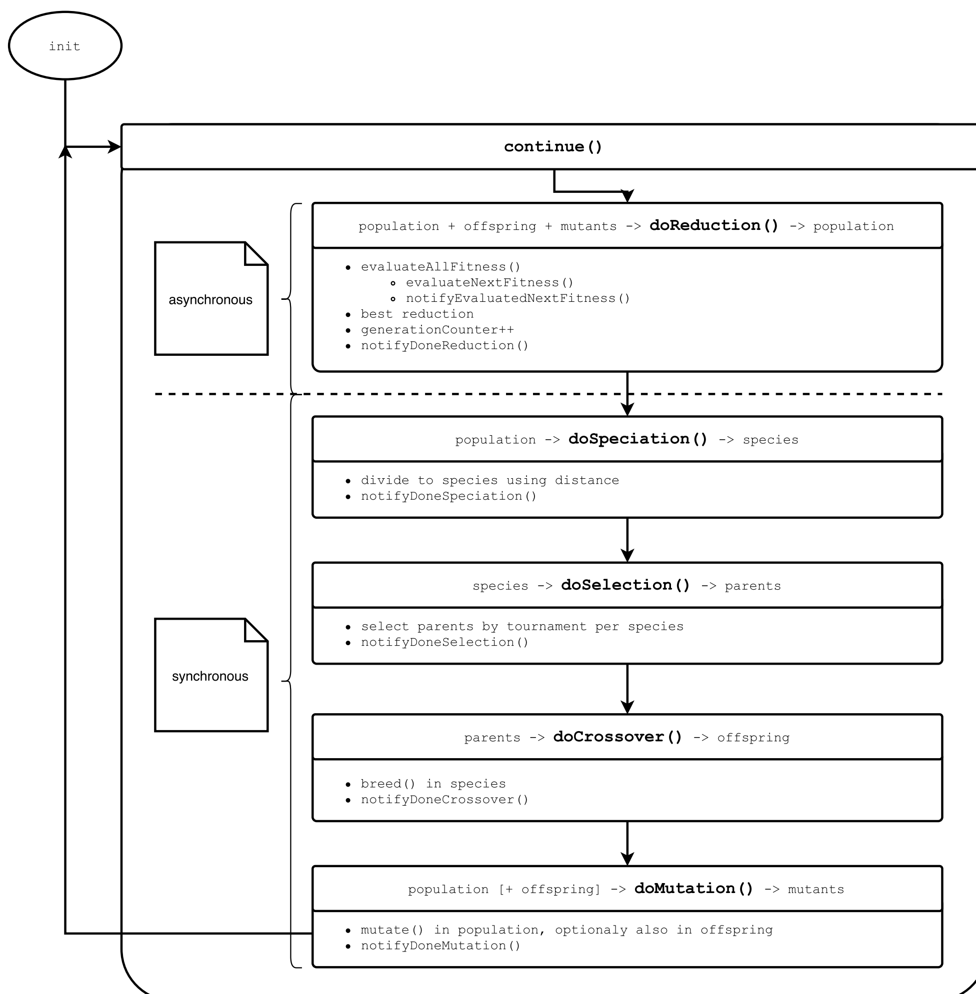
parents - reference na jedince, kteří během selekce vyhráli, pole pro každý druh.

offspring - nově vytvoření jedinci z rodičů, pole pro každý druh.

mutants - reference na jedince, kteří podstoupil mutaci.

observers - lze registrovat observer, který bude notifikován po každém kroku cyklu

config - objekt udržující konfigurační proměnné. Především pravděpodobnosti mutací, parametry distance funkce a podobně.



Obrázek 4.8: Diagram fungování cyklu.

Volání funkcí cyklu je nejlépe vidět na diagramu 4.8. Metoda `continue` spouští vyhodnocení další generace. Zopakují se kroky redukce, rozdělení do

druhů, selekce v druzích, křížení v druzích a mutace. První krok, který vyhodnocuje fitness jedinců je asynchronní a využívá promise. V této fázi jednotliví boti hrají hru v reálném čase. Proto je tato část zdaleka nejnáročnější na čas.

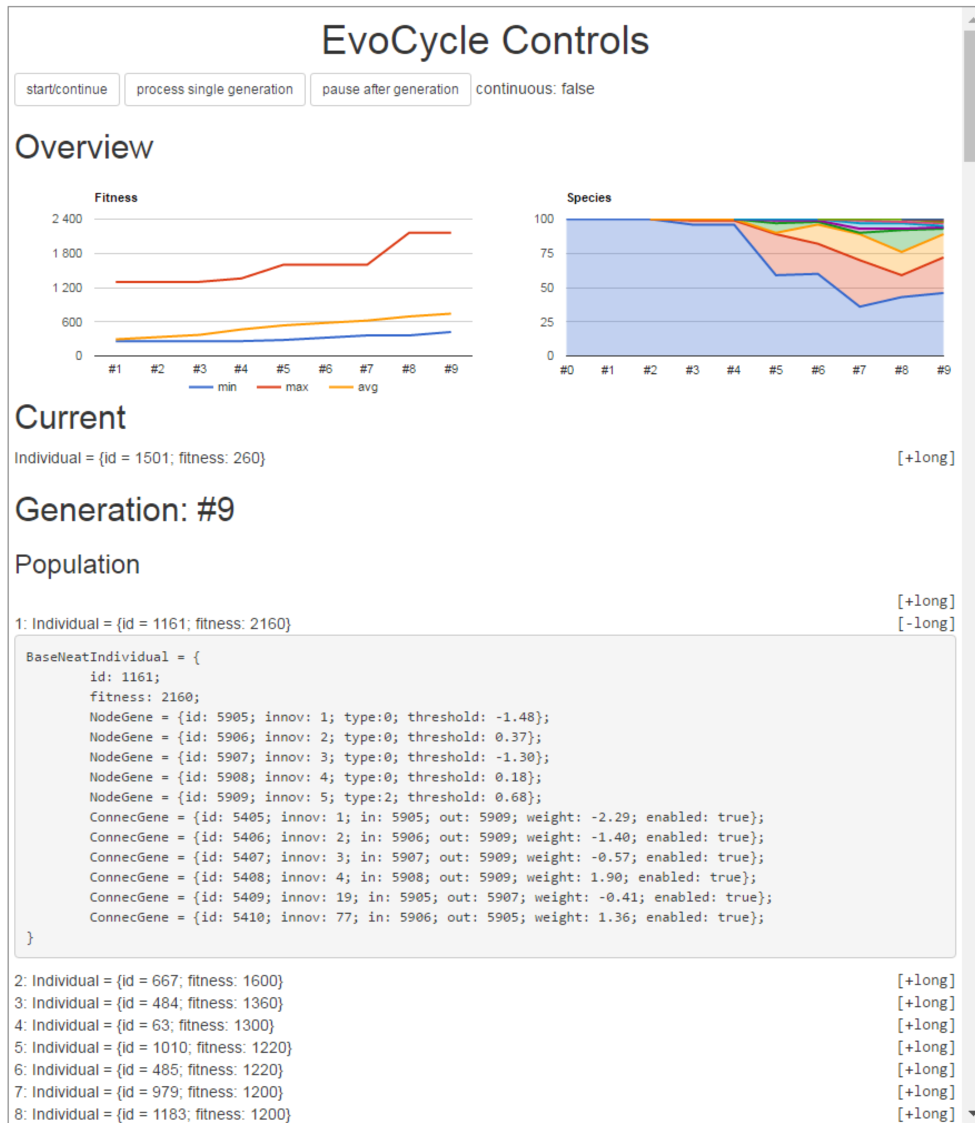
4.3 EvoCycleControls

Pole `parents`, `offspring`, `mutants` jsou ve třídě `EvoCycle` především kvůli jednostránkové aplikaci napsané v AngularJS, která má za úkol sledovat a zobrazovat model a dát uživateli možnost měnit konfigurační proměnné, což lze i za běhu simulace. Skládá se z angular controller `EvoCycleControls`, také zobrazena na diagramu dia:EvoCycle, a jedné html stránky s Angular prvky. Výsledná aplikace je vidět na obrázku 4.9. Angular umožnuje velice jednoduchý double-binding, kterého zde využívám. Třída navíc implementuje rozhraní `EvoCycleObserver` a registruje se jako **observer** třídy `EvoCycle`.

`model` - reference na `EvoCycle` pro možnost okamžitého zobrazování a editaci dat

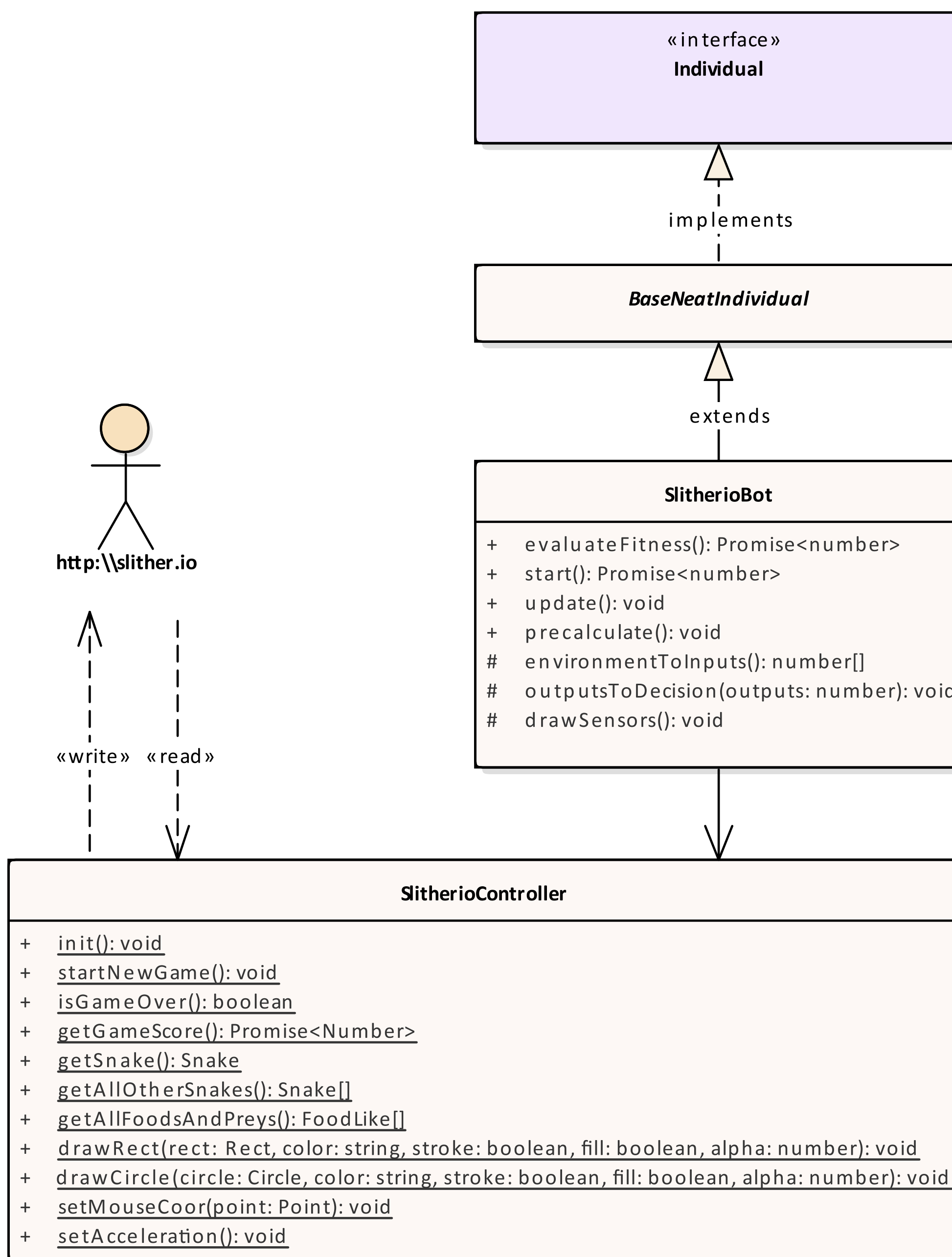
`init` - je velmi důležitá funkce, která vytvoří a nastartuje bootstrap aplikaci.

`update*Graph` - vykreslují změny v grafech.



Obrázek 4.9: Na tomto snímku obrazovky je vidět html div, který je vložen do stránky a angular `EvoCycleControls` aplikací jsou v něm rozumnou formou zobrazena data z `EvoCycle`. V horní části je několik tlačítek na ovládání cyklu. Následuje přehled s dvěma grafy. Jeden zobrazuje fitness, druhý rozložení jedinců do druhů v průběhu všech generací. Dále jsou vypsání jedinci. Nejdříve aktuální, který hraje hru, dále celá populace, rodiče vybraní k páření, potomci a nakonec zmutovaní jedinci. První jedinec v populaci má rozbalený dlouhý výpis, kde jsou vidět jeho geny. Na konci je několik vstupních polí, které zobrazují aktuální nastavení evolučního cyklu a tyto hodnoty zde lze měnit.

4.4 Boti



Obrázek 4.10: Diagram třídy `SlitherioBot` a závislostí.

Dalšími balíčky jsou jednotlivé implementace botů. Toto je **jediná** část programu, která je problémově závislá a je jí nutné vyvinout konkrétní pro každou hru. Já jsem napsal automatické hráče pro dvě hry. V obou případech se má

řešení skládají ze tříd `*Controller` a `*Bot`, které implementují fungování automatického hráče. A třídy `*Loader` s jedinou metodou startující celý proces vytvořením populace, instance `EvoCycle` a inicializací `EvoCycleControls`. Samotné spuštění cyklu je možné udělat také programově, ale v mém případě čekám na vstup od uživatele.

Nevytvářel jsem pro tuto část žádná rozhraní tříd, protože implementace může být provedena různě. Jediné, co je nutné splnit, je rozhraní jedince popsané na začátku této kapitoly v 4.1.1. Teoreticky není ani nutné využívat `EvoCycleControls`.

4.4.1 SlitherioController

Třída `SlitherioController` tvoří rozhraní mezi samotnou hrou a automatickým hráčem. Disponuje operacemi čtoucí stav hry, vykonávající pohyb hráče nebo vykreslující dodatečné informace na obrazovku. Popis metod uvedených na diagramu 4.10

`init` - přepne hru do mobile render mode což snižuje výpočetní náročnost. Dále přepisuje některé původní funkce jako například **`redraw`**, **`onmousemove/down`**, **`onkeydown`**, čímž hru obohacuje o vlastní funkcionalitu, ale stále zavolá i původní funkce.

`startGame` - zapne novou hru.

`isGameOver` - true pokud hráč umřel.

`getGameScore` - vrací získané skóre hráče.

`getSnake` - vrací objekt reprezentující hráče.

`getOtherSnakes` - vrací objekt reprezentující ostatní hráče.

`getFoodsAndPreys` - vrací objekt reprezentující potravu.

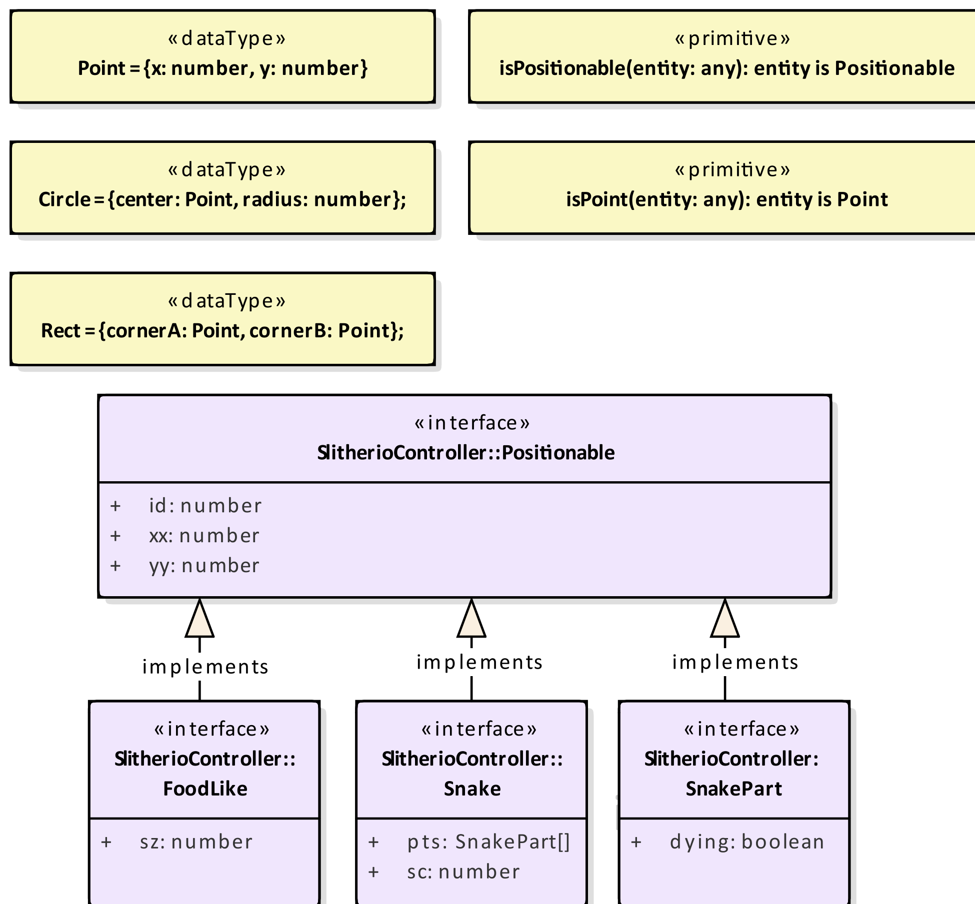
`setMouseCoor` - nastaví kam se má hráč pohybovat.

`setAcceleration` - použije speciální vlastnost hráče díky které se pohybuje rychle ale ztrácí hmotnost.

`draw*` - funkce pro vykreslování dodatečné grafiky na obrazovku.

Všimněte si také z diagramu 4.11, že jsem zavedl `vlastní typy` a `rozhraní`. Díky tomu mám **typovou kontrolu** i nad JS objekty, které získávám přímo ze stránky. To je v TypeScriptu běžné. Kromě vracení konkrétních typů místo nicneříkajícího `any`, může být výhodou například i takováto funkce:

```
/**
 * Converts Positionable to Point, keep Point untouched.
 */
public static positionableToPoint(entity: Positionable | Point):
    Point { // takes any of these types
```

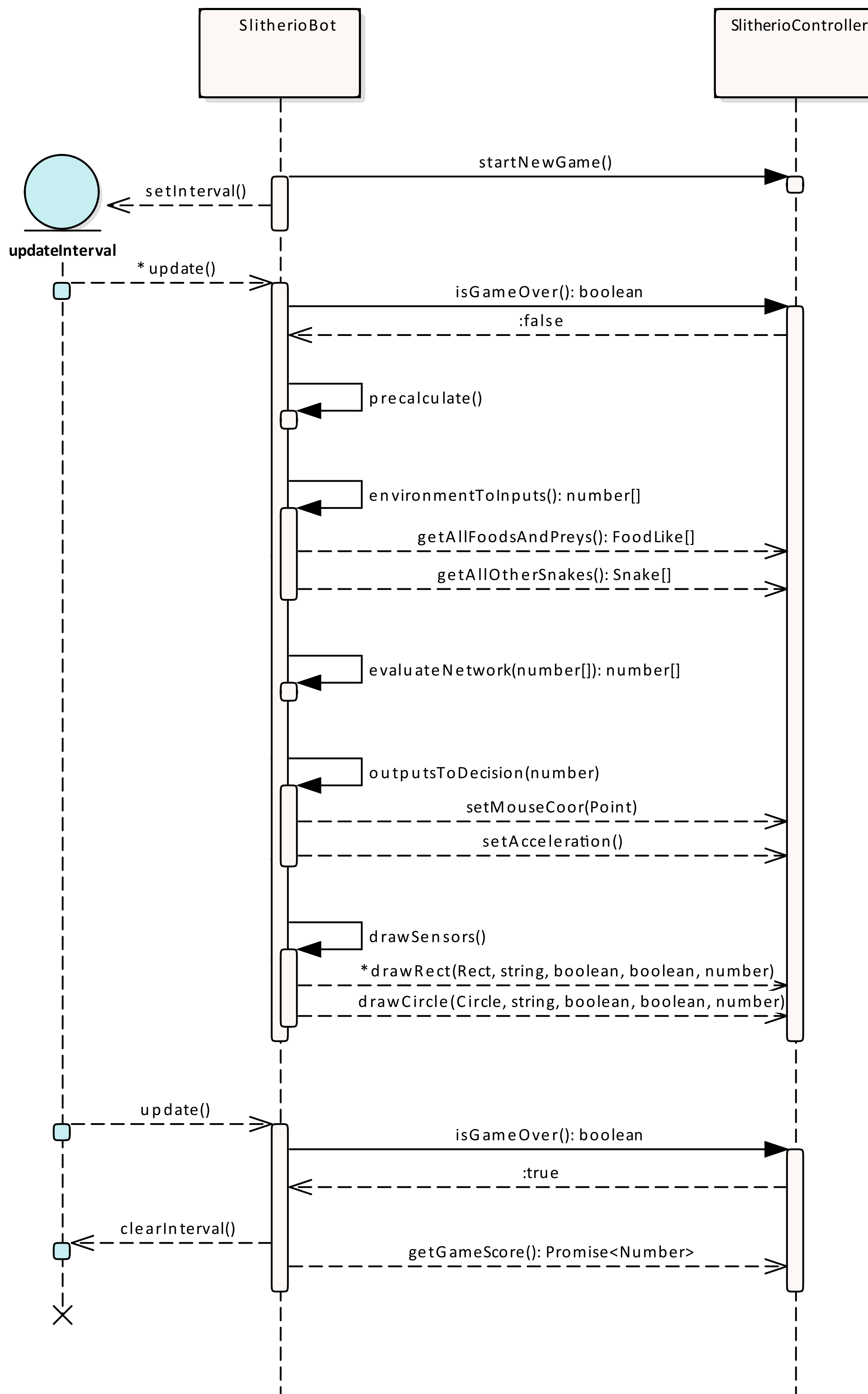


Obrázek 4.11: Diagram typů a rozhraní definovaných ve třídě `SlitherioController`

```
    // return {x: entity.xx, y: entity.yy}; // Error: Property 'xx'
        does not exist on type Point
        if(isPoint(entity)) { // type guard
            return entity; // \textbf{compiler knows it is Point in
                this block}
        } else {
            return {x: entity.xx, y: entity.yy}; // \textbf{compiler
                knows it is Positionable in this block}
        }
    }
```

4.4.2 SlitherioBot

`SlitherioBot` dědí ze třídi `BaseNeatIndividual`, která implementuje rozhraní `Individual`. Implementuje abstraktní metodu `evaluateFitness`, čímž



Obrázek 4.12: Diagram komunikace `SlitherioBot`

se stává konkrétní implementací, kterou lze použít v `EvoCycle`. Na diagramu 4.12.

`evaluateFitness` - implementace abstraktní metody rodiče. Startuje hráče. Když je `Promise` dokončena, uloží skóre do fitness a řetězí jej dále.

`start` - metoda vytvoří interval, který každých 100ms volá metodu `update` a startuje hru.

`update` - provede rozhodnutí hráče o následujícím tahu, vykreslí sensory a směr pohybu na obrazovku.

`precalculate` - ukázalo se, že metoda `update` je poměrně náročná, pokud je v okolí mnoho ostatních hráčů, proto se na jednou místě předpočítají hodnoty, které se využívají při rozhodování opakovaně.

`environmentToOutputs` - výpočetně nejnáročnější metoda. Pro každou část hada a potravu rozhoduje, zda neleží v jednom ze senzorů. Rychlé rozhodnutí, které kontroluje, zda vůbec prvek je dostatečně blízko, aby mohl v některém ze senzorů být, úspěšně snížilo náročnost výpočtu.

`evaluateNetwork` - metoda rodiče vyhodnocující neuronovou síť na základě vstupů.

`outputsToDecision` - výstupy neuronové sítě je potřeba interpretovat do pohybů hráče.

`drawSensors` - vykreslí sensory červeně, pokud se v nich nachází nepřítel, zeleně a tučně podle součtu hodnot potravy, která se nachází v senzoru. Vykresluje také směr pohybu jako bílou tečku na obrazovku.

4.5 Sestavení a distribuce

Pro build aplikace využívám task manageru `gulp.js`. Samotný gulp i jeho komponenty spravuji přes balíčkovací systém `npm`. `npm` využívám i pro správů `.d.ts` (deklarace API) souborů knihovnen, na kterých je aplikace závislá. Samotné knihovny jsou v projektu obsaženy, ale nejsou importovány do výsledné aplikace. Místo toho využívám možnosti Tampermonkey importovat scripty.

TypeSript soubory kompiluji do JavaScriptu pro distribuci na web pomocí balíčků a funkcí `browserify`, `tsify` a `bundle`. Získám tak jediný JS soubor. Compiluji do ECMAScript6, což je verze obsahující promise už v jádře a v Chromu je téměř plně podporována.

Posledním krokem je vytvoření Tampermonkey scriptů pro aplikaci botů. Jsou to JS soubory obohacené o anotace, například `@name`, `@author` pro popis souboru, `@match` pro automatické spouštění na daných stránkách nebo zmíněné `@require` pro importování skriptů. Přestože vygenerovaný JavaScript, který vkládám do Tampermonkey scriptu je validní, Tampermonkey jshint v něm nalezne chyby (je příliš striktní), tak proto jej vypínám.

Pro `EvoCycleControls` mám vlastní Tampermonkey script, který do DOM stránky vkládá nutné css a html. Všechny zmíněné kroky lze provést spuštěním defaultního úkolu gulp z příkazové řádky v kořenové složce projektu.

4.6 Uživatelská příručka

Na stránce `http://neat.melezinek.cz/` můžete vidět mnou vyvinutou benchmarkovací hru, na které můžete spustit `EvoCycleControls` včetně automatického hráče kliknutím tlačítka. Pro spuštění na hře `http://slither.io` je nutné nainstalovat Tampermonkey (plugin do Google Chrome) a vložit do něj scripty `lib.contorls.js` a `lib.slitherio.js` z přiloženého CD ze složky `dist`, viz příloha B Obsah CD.

Pro případný další vývoj je zajímavá složka `src` a soubor `gulpfile.js`, viz příloha B Obsah CD. Pro založení projektu:

- nainstalujte Node.js,
- instalujte npm balíčky příkazem: `npm install`,
- zkompilujte projekt příkazem: `gulp [game | controls | balancing | slitherio]`,

Přidat vlastního automatického hráče je jednoduché. Doporučuji jej implementovat v TypeScriptu i když to není nutné. Jako rodiče použijte `BaseNeatIndividual` a implementujte logiku hraní. Vytvořte třídu `*Loader` podobně jako `BalancingLoader` a novou gulp úlohu v `gulpfile.js`. Inspirujte se již vytvořeným taskem `balancing`.

Kapitola 5

Experimenty

5.1 Pole Balancing Game jako benchmark prostředí

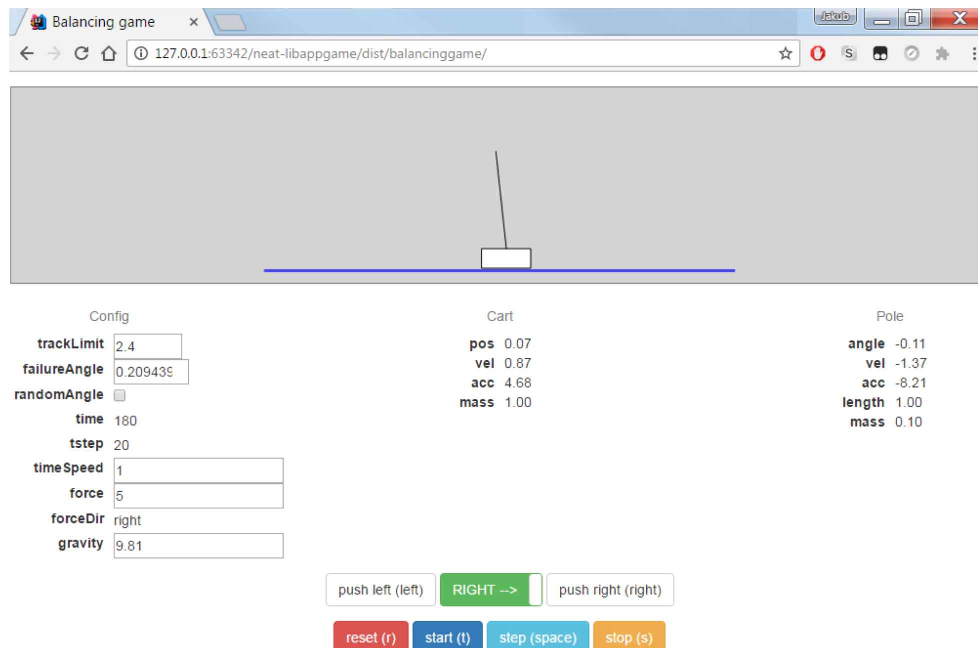
Pro účely této práce byla vytvořena webová hra postavená na základech níže popsaného problému. Jde o v javascriptu napsanou aplikaci dostupnou na `http://neat.melezinek.cz` a také přiloženou na CD. Ukázka je na obrázku 5.1. Využívá knihoven AngularJS, jQuery a jsql. Aplikace slouží jako testovací a zkušební prostředí pro praktickou část této diplomové práce.

5.1.1 Popis problému

Pole balancing problem, také nazýván pole-cart problem, broom/stick balancer, inverted pendulum problem, česky problém vyvážení převráceného kyvadla je základní úlohou v teorii řízení a zároveň často používanou benchmarkovací úlohou pro měření výkonu umělých neuronových sítí. I K. O. Stanley a R. Miikkulainen jej ve své práci [8] používají k porovnávání ostatních metod s metodou NEAT. Proto je problém převráceného kyvadla použit i v této práci jako kontrolní úloha.

Jak popisuje ve své práci J. Brownlee [31] máme vozík s tyčí (převráceným kyvadlem) pohybující se po vyznačené dráze. Tato tyč je k vozíku připevněna v jeho středu tak, že se může pohybovat pouze v jedné ose. V této stejné ose se pohybuje i vozík po trati. Stav systému je určen 4 hodnotami: úhlem tyče Θ, úhlovou rychlostí tyče Θ', pozicí vozíku vůči středu dráhy x a rychlostí vozíku x'. Výstupem ovládacího systému je rozhodnutí o směru působení síly na vozík, nikoliv však její velikosti. Tím je učiněno rozhodnutí o pohybu vozíku. Systém je typicky inicializován s vozíkem uprostřed tratě a tyčí ve vyrovnané poloze, případně s náhodným úhlem z normálního rozdělení s malou standardní odchylkou.

Řešení problému vyžaduje zpětnovazební kontrolní systém, který dokáže takovou tyč vybalancovat. Za neúspěšný řídící systém je považován takový,



Obrázek 5.1: Pohled na webovou hru problému převráceného kyvadla. V horní části je vizualizace systému v reálném čase, pod ní se opět v reálném čase vypisují veškeré proměnné systému - konfigurační, vozíku, tyče. Pod textovým výpisem nalezneme ovládací prvky, které využívá jak skutečný hráč (především přes klávesové zkratky), tak i automatický hráč (simulováním kliknutí/voláním odpovídající funkce). Na závěr krátký text, ve kterém je důležité tlačítko, které spouští simulovanou evoluci.

který neudrží tyč ve vzpřímené pozici, nebo vyjede s vozíkem mimo dráhu. Úspěšný je ten, který dokáže s tyčí balancovat po stanovenou dobu.

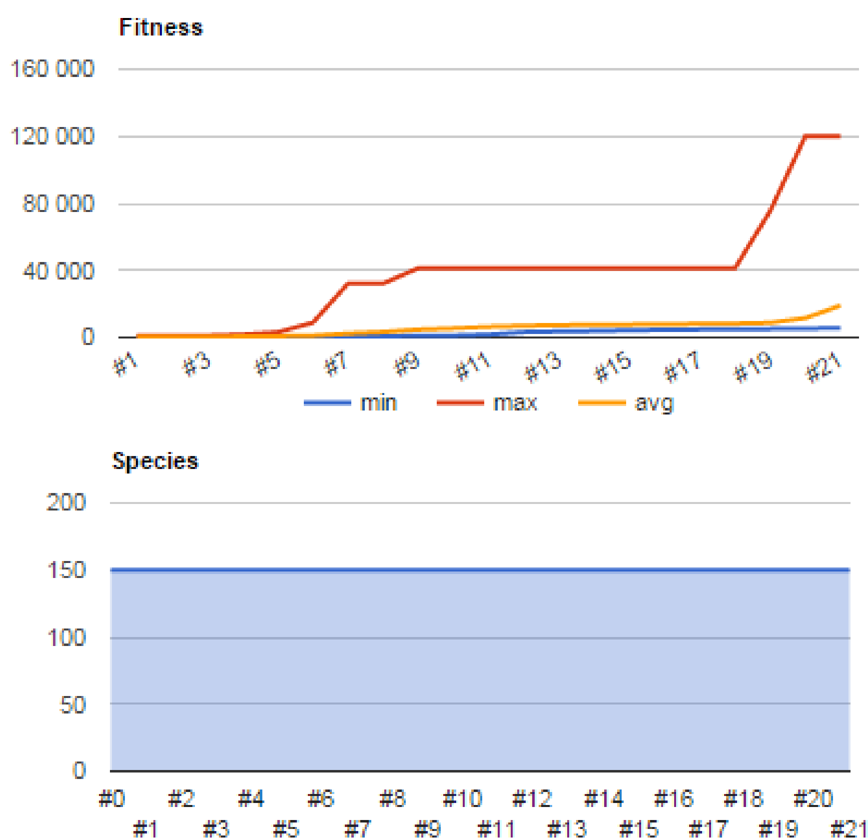
5.1.1.1 Proměnné systému

Symbol	Popis	hodnota [jednotka]
Θ	úhel tyče	[rad]
Θ'	úhlová rychlost tyče	[rad/s]
Θ''	zrychlení tyče	[rad/s^2]
x	pozice vozíku	[m]
x'	rychlost vozíku	[m/s]
x''	zrychlení vozíku	[m/s^2]
g	tíhové zrychlení	9,81 [m/s^2]
m_c	hmotnost vozíku	1,0 [kg]
m_p	hmotnost tyče	0,1 [kg]
l	vzdálenost mezi osou tyče a těžištěm	0,5 [m]
t	čas	[s]
F	síla	typicky ± 1 až ± 10 [N]
h	polovina délky drýhy	2.4 [m]
r	maximální přijatelné vychýlení tyče	$\pm 12°=$ 0.209 [rad] od vzpřímení pozice (0°)
τ	časový krok	[s]

Tabulka 5.1: Proměnné systému

5.2 BalancingBot

Při návrhu je důležité určit vstupy a výstupy neuronové sítě, která bude šlechtěna. A také jejich kódování. Vstupem a výstupem mohou být pouze čísla (v případě skokové funkce je výstup omezen na diskrétní hodnoty 0, 1). Stav problému převráceného kyvadla je definován proměnnými systému, které se mění s časem. Tedy úhel tyče, úhlová rychlost tyče, zrychlení tyče, pozice vozíku, rychlost vozíku a zrychlení vozíku. Existuje více verzí problému, kde nejtěžší z nich poskytuje jedincům pouze pozici vozíku a úhel tyče (případně tyčí). Hra tak jak jsem ji navrhl dává k dispozici všechny tyto proměnné, ale použiji jen první dvě zmíněné od každého, dohromady 4 číselné vstupy. Jediným výstupem je informace, zda tlačit vozík doprava nebo doleva. Mohl bych mít dva výstupní uzly, kdy každý bude říkat, na kterou stranu tlačit, ale je šance že by oba byly neaktivní nebo aktivní naráz, což není povolená kombinace. Proto bude výstupní pouze jeden. Poslední volbou je aktivační funkce neuronů – skoková nebo sigmoidní. Skoková funkce na neuronech bude znamenat, že výsledek dostanu v podobě „tlač vlevo, nebo tlač vpravo“. Sigmoidní aktivační funkce dovoluje říkat s jakou jistotou se má akce provést, případně



Obrázek 5.2: Grafy pro první experiment s `BalancingBot`. Osa x číslo generace, osa y počet ms pro Fitness graf, počet jedinců pro Species graf.

s jakou silou. Nemyslím si, že toto rozhodnutí má zásadní vliv na výsledek a volím sigmoidní funkci. Výstup je kódován tak, že pokud je na výstupu záporné číslo, vozík bude tlačen vlevo, v ostatních případech vpravo. Fitness funkce je přirozená a to počet milisekund, po které dokázal hráč splňovat podmínky hry.

5.2.1 Experiment č. 1

V prvním zaznamenávaném experimentu se snažím napodobit podmínky zmíněné v práci K. O. Stanley a R. Miikkulainen [8], ze které jsem čerpal při analýze a návrhu. Koeficienty funkce podobnosti jsou $c_1 = 1, c_2 = 1, c_3 = 0.4$ a $\delta = 3$. Dále vím, že pouze 25% jedinců mutuje topologii. Šance na mutaci přidáním uzlu jsou 3% a na přidání hrany 5%. Ostatní hodnoty jsem odhadnl. Šanci mutovat váhy nastavuji vyskokou a 80% jedinců zmutuje všechny váhy přičtením náhodné hodnoty ze standartního normálního rozdělení. Vygeneruje se stejný počet potomků jako je velikost populace a rodiče se vybírají

po druzích turnajovým výběrem o velikosti poloviny druhu. Mutace provádím naklonováním jedince, takže o původního nepřijdu, což je dobré v případě, že by mu mutace velmi snížila zdatnost. Mutuji i potomky vytvořené křížením. V dalších generacích nepřepočítávám fitness u již ohodnocených potomků, protože startovní podmínky jsou vždy stejné a není důvod předpokládat jiný výsledek.

Z grafu fitness 5.2 je vidět dlouhý zásek mezi 9. (méně než půl hodina) a 18. (1h 15min) generací. Přesto v tuto dobu bylo několik nejlepších jedinců schopno hrát velmi dobře a balancovat s tyčí na obě strany, jejich problém byl, že přejeli hrací plochu. Domnívám se, že v 18. generaci vznikla mutace, která tento problém odstranila, a proto skokově fitness stoupla ve dvou generacích na 120 tisíc ms, což byl nastavený strop hry. Dlouhou dobu si vysvětluji nízkou mutací topologie.

Na grafu druhů 5.2 je vidět že existoval pouze jediný druh celou dobu simulace. To není ideální situace, Druhy pomáhají hledat řešení více směry. Dále bych čekal, že graf průměrné zdatnosti bude vyšší. Takto se zdá, že jen malé procento jedinců je úspěšných, nebo že poměrně mnoho jedinců, kteří se do populace dostanou, skončí krátce po startu.

Zajímavostí bylo, že hráči měli tendenci balancovat na pravé straně hrací plochy velmi blízko kraji. Trvání generaci se ke konci velmi prodlužovalo, protože více a více jedinců bylo schopno hrát déle.

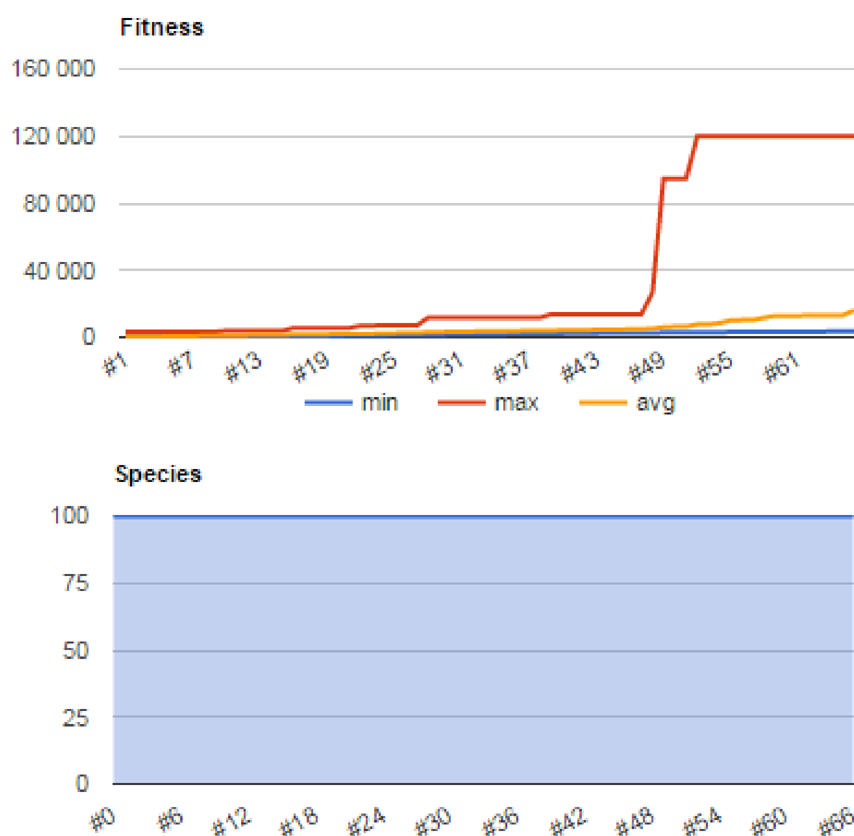
5.2.2 Experiment č. 2

Druhým experimentem se pokusím vylepšít ten první. Především zvýším šanci mutovat topologii. Každy jedinec mutuje přidáním nového genu s šancí 9% a nové hrany 15%. Dále snižuji počet jedinců na 100 a velikost vygenerovaných potomků na 75% velikosti populace. Snížil jsem šanci mutovat váhy, tak že jenom polovina vah bude zmutována. Těmito změnami bych chtěl docílit větší diverzity populace a rozdělení do více druhů a zároveň urychlit jednotlivé generace (tím celý proces) ale stále dosáhnout stejných výsledků.

Jak je vidět z grafů 5.3 dosáhl jsem pouze později zmíněného. Velký skok kolem 49. generace byl ve 40. minutě hraní. Pokud to porovnáme s prvním experimentem odpovídalo by to asi 15. generacím. Výsledkem byly opět jedinci schopni hrát 2 minuty. Experiment jsem nechal ještě chvíli pokračovat (celkově asi 1h), abych viděl jestli bude stoupat průměrná fitness, tedy že se více a více jedinců učí dobře hrát a nebyly to pouze náhody.

5.2.3 Experiment č. 3

V závěrečném experimentu jsem snížil práh distanční funkce na 0.75, aby jenom opravdu blízcí jedinci spadli do stejného druhu. Také jsem zvýšil maximální dobu hry na 10 minut. Graf druhů 5.4 konečně ukazuje i další druhy. Dokonce je jich řekl bych až příliš. Přesto simulace během hodiny a půl (po-

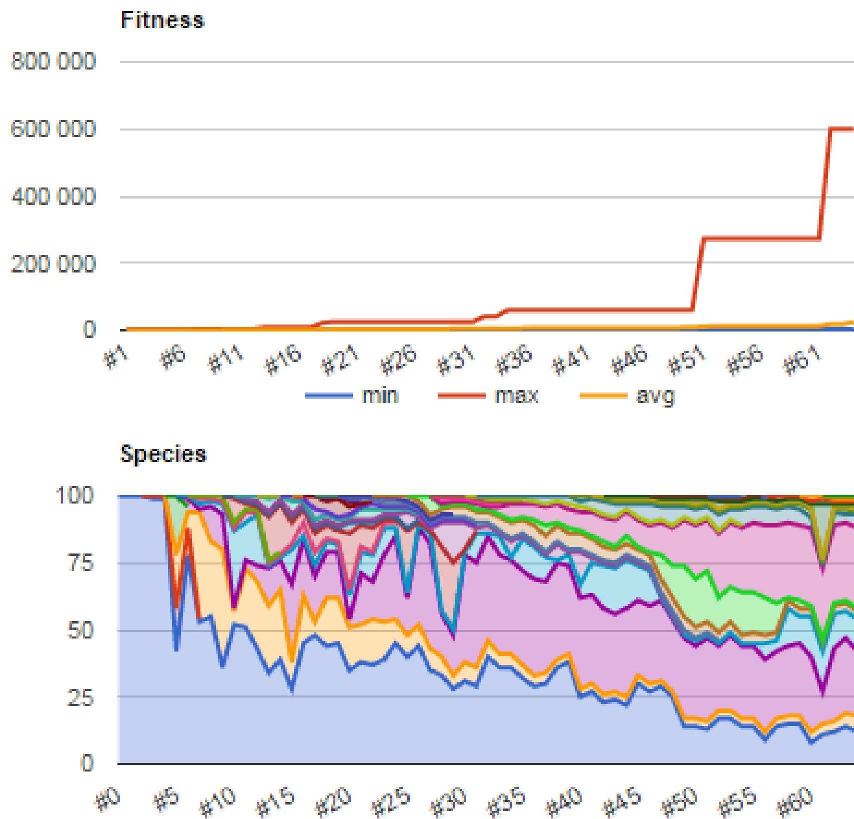


Obrázek 5.3: Grafy pro druhý experiment s `BalancingBot`. Osa x číslo generace, osa y počet ms pro Fitness graf, počet jedinců pro Species graf.

dobně dlouho jako 1. experiment) vyšlechtila jedince, kteří byly schopni hrát 10 minut.

Velmi zajímavé je podívat se na nejlepšího jedince. Ten má pouze jeden jediný `ConnectGene` navíc. Takže by se zdálo, že většinu práce zde odvedla mutace vah. Ale tento gen je velice důležitý. Tato hrana vede mezi vstupním uzlem, na který je přivedena hodnota ryclosti vozíku a dalším vstupním uzlem, na který je přivedena hodnota pozice vozíku. Když si uvědomíme, že pozice středu má hodnotu 0 a pozice od ní vpravo kladné hodnoty, pozice vlevo záporné hodnoty a to stejné platí pro rychlost vozíků vpravo a vlevo, je toto **gen, který zařídí, že vozík nepřejede z hrací plochy**.

Druhý jedinec je pouze mírnou mutací prvního a také dosáhl fitness 600 tisíc. Třetí nejlepší jedinec má 3 hrany a jeden uzel navíc a dosáhl pouze poloviční zdatnosti. To ukazuje, že **více genů nemusí nutně znamenat lepší výkon**, viz tito dva jedinci:



Obrázek 5.4: Grafy pro třetí experiment s `BalancingBot`. Osa x číslo generace, osa y počet ms pro Fitness graf, počet jedinců pro Species graf.

```
BaseNeatIndividual = {
  id: 10770;
  fitness: 600000; // 1.
  NodeGene = {id: 61598; innov: 1; type:0; threshold: -1.73};
  NodeGene = {id: 61599; innov: 2; type:0; threshold: 1.70};
  NodeGene = {id: 61600; innov: 3; type:0; threshold: -1.11};
  NodeGene = {id: 61601; innov: 4; type:0; threshold: -1.03};
  NodeGene = {id: 61602; innov: 5; type:2; threshold: -1.59};
  ConnecGene = {id: 72099; innov: 1; in: 61598; out: 61602;
    weight: 1.12; enabled: true};
  ConnecGene = {id: 72100; innov: 2; in: 61599; out: 61602;
    weight: 0.60; enabled: true};
  ConnecGene = {id: 72101; innov: 3; in: 61600; out: 61602;
    weight: 2.61; enabled: true};
  ConnecGene = {id: 72102; innov: 4; in: 61601; out: 61602;
```

```
    weight: 1.26; enabled: true};
  ConnecGene = {id: 72103; innov: 60; in: 61599; out: 61598;
    weight: 0.40; enabled: true};
}

BaseNeatIndividual = {
  id: 8713;
  fitness: 272540; // 3.
  NodeGene = {id: 49345; innov: 1; type:0; threshold: 6.62};
  NodeGene = {id: 49346; innov: 2; type:0; threshold: -0.05};
  NodeGene = {id: 49347; innov: 3; type:0; threshold: 0.07};
  NodeGene = {id: 49348; innov: 4; type:0; threshold: -1.71};
  NodeGene = {id: 49349; innov: 5; type:2; threshold: -4.31};
  NodeGene = {id: 49350; innov: 238; type:1; threshold: 2.21};
  ConnecGene = {id: 56934; innov: 1; in: 49345; out: 49349;
    weight: -2.05; enabled: true};
  ConnecGene = {id: 56935; innov: 2; in: 49346; out: 49349;
    weight: 2.56; enabled: true};
  ConnecGene = {id: 56936; innov: 3; in: 49347; out: 49349;
    weight: 7.64; enabled: true};
  ConnecGene = {id: 56937; innov: 4; in: 49348; out: 49349;
    weight: 8.03; enabled: true};
  ConnecGene = {id: 56938; innov: 28; in: 49348; out: 49346;
    weight: -0.70; enabled: true};
  ConnecGene = {id: 56939; innov: 875; in: 49348; out: 49350;
    weight: 1.68; enabled: true};
  ConnecGene = {id: 56940; innov: 876; in: 49350; out: 49346;
    weight: 1.39; enabled: true};
}
```

5.2.4 Závěr

Během těchto tří zaznamenaných experimentů a velkého množství nezaznamenávaných předtím, které sloužily především pro verifikaci kódu, se ukázalo, že nastavení cyklu má vliv na schopnost se učit. To není překvapující fakt. Bohužel najít nejvhodnější nastavení není snadné.

V posledním experimentu jsem demonstroval dvě vlastnosti jedinců. Zaprvé, více genů nemusí nutně znamenat lepší zdatnost, i když měl jedinec dlouhou dobu na to, aby optimalizoval váhy. A zadruhé, mutace, které přidávají geny jsou důležité a mutace vah slouží pouze k jejich zdokonalování.

Co je dobré neopomenout je fakt, že po celou dobu experimentu je nutné, aby okno prohlížeče s hrou bylo v popředí. Pokud nebude, tak prohlížeč omezí vykonávání všech javascript intervalů na jedno volání za sekundu, tím experiment víceméně zastaví.

Experimenty s touto hrou trvají kolem hodiny a půl a jsou tak vhodné pro zkoušení dalších variant. Například sigmoidní aktivační funkce versus skoková, jak který parametr cyklu ovlivňuje vývoj fitness atd.

5.3 SlitherioBot

5.3.1 Slither.io versus Agar.io

Prodleva mezi rešeršní a implementační částí této práce způsobila že Agar.io změnil svoje API a veškeří hráči, které jsem popisoval v kapitole 2 přestali fungovat. Po dlouhé snaze zprovoznit některého z nich, nalezení nově vytvořených fungujících s novou verzí hry i pokusech o napsání vlastního úplně od začátku, jsem od toho kvůli náročnosti upustil.

Místo toho využiji velmi podobné hry Slither.io. Dalo by se říct, že je to klon Agar.io s několika změnami. Hra staví na stejných základech. Tedy na ohraničené ploše se pohybuje váš hráč, v tomto případě had, kterým se snažíte posbírat co nejvíce jídla. Ostatní hráče můžete eliminovat tím, že narazí do ocasu jiného hada. Z takového hada vypadne velké množství potravy. Vlastní ocas může had bez omezení překračovat. Rozdílem oproti Agar.io je, že speciální vlastností hada není „výpad“, ale zrychlení, čímž může někomu zablokovat cestu, ale ztrácí tak rychle svou hmotnost. Navíc velikost nehraje roli přimo v tom koho lze sníst, ale nepřímo tím, že delší had vás může obmotat. Tuhle vlastnost nepředpokládám, že se dokáží moji hráči naučit.

Volba na Slither.io padla, protože je tak podobný, a protože jsem našel funkčního bota [32], ze kterého jsem mohl čerpat inspiraci. Dokonce si myslím, že je vhodnější pro učení hráčů než původně zamýšlený Agar.io.

5.3.2 Senzory

Oproti první hře zde musím více přemýšlet, jak kódovat vstupy a výstupy neuronové sítě. První nápad byl dávat na vstupy hodnoty nejbližší potravy nebo překážce. V této variantě je problém, že počet vstupů by byl proměnný a nejspíše by jich bylo velké množství, protože potravy, hlavně v případě, že právě umřel nějaký hráč, je v okolí velké množství. Nehledě na to, že tento přístup nijak nebere v potaz směr potravy, pouze vzdálenost.

Co se zdá jako mnohem logičtější řešení, je vytvoření senzorů, které budou scenovat část obrazovky a výsledná hodnota bude součtem hmotnosti jídla v tomto senzoru nebo -1 v případě, že se na ploše senzoru nachází překážka. Vizualizace je na obrázku 5.5. Takto oddělím plochy, kterým se hráč má vyhnout a plochy, které by měl preferovat a to dokonce s vahou.

Přemýšlel jsem nad ještě jedním vstupním parametrem a tím je velikost hráče. Nakonec se mi nezdála tato informace důležitá pro zamýšlený cíl, kterým je naučit bota pohybovat se tak, aby se vyhýbal překážkám a jedl u toho co nejvíce jídla.

Obrázek 5.5: Snímek obrazovky zachycující automatického hráče jak hraje slither.io. Pastelově růžový hráč uprostřed s popiskem „playing...“ je právě hrající hráč, který je ovládaný třídou `SlitherioBot`. Kolem hráče jsou jakoby v kruhu rozmístěny čtvercové senzory ve vrstvách. První vstrva má pouze 4 senzory. každá další 12. Senzory reagují na množství jídla, které v nich je tím. že se více zeleně zabarví. Pokud je uvnitř senzoru překážka (jiný had nebo konec mapy), senzor zčervená. Barvy odpovídají hodnotě, kterou senzor nabírá a neuronová síť hráče zpracovává. Malé šedé kolečko značí směr pohybu hráče. Normálně je hráč ovládán myší, v tomto případě má možnost se pohybovat směrem jednoho z 12 senzorů z druhé a vyšší vrstvy.

Jako nejjednodušší možnost výstupu se jeví x-ová a y-ová souřadnice, jenže nevidím způsob, jak by se je měla neuronová síť naučit počítat z daných vstupů. Proto výstupů jakým směrem se pohybovat bude 12, stejně jako senzorů v jedné vrstvě a budou namapovány na odpovídající senzor. Tento přístup bere v úvahu i směr potravy a překážky.

Posledním výstupem je příznak, zda se má hráč pohybovat rychle a ten je řešen jedním výstupním uzlem, který pokud nabývá hodnoty větší než nula znamená, že hráč chce akcelerovat.

Počet vrstev senzoru je jednoduše nastavitelný stejně tak, jako jeho základní šířka. Pro tyto experimenty používám šířku 50px a 4 vrstvy včetně první, která má pouze 4 senzory, dohromady 40 vstupních hodnot a 13 výstupních. Všechny uzly opět používají sigmoidni aktivační funkci. Jako zdatnost se bere hodnota, kterou hra zobrazuje na konci každého odehraného kola a je to délka hada při smrti. Jiné fitness funkce by mohly být také vhodnou metrikou, například skóre z leaderboard, které ale nevím, jak od hry získat nebo vypočítat. Nebo brát v potaz i dobu jakou had přežije.

5.3.3 Experiment č. 1

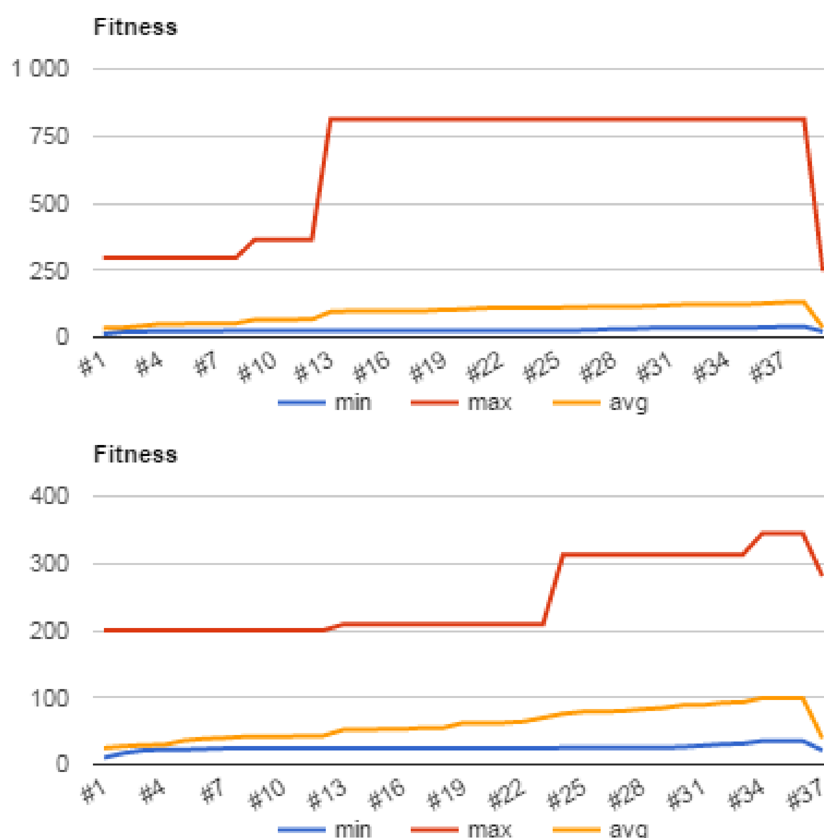
Nastavím cyklus, aby i jednou vyhodnocené jedince znovu nechal hrát. Slouží k tomu příznak `alwaysEvaluateFitness`. Dělám to z důvodu, že oproti první hře tato je generována náhodně a hrají v ní ostatni hráči online a tak výsledek dvou spuštění stejného hráče může být rozdílný. Počet jedinců bude 200, ať může být populace diverzifikovaná.

Využiji možnosti, že lze nastavení měnit za chodu a začnu s velkou mírou mutací topologie i vah, a budu doufat, že to velká populace zvládne a postupně tyto hodnoty snížím, abych nechal jedince se specializovat.

Bohužel jsem tento experiment ukončil již po půl hodině, když jsem zjistil, že svou první hru zatím odehrálo pouze 50 jedinců, tedy čtvrtina z celkového počtu. Tímto tempem by každá generace trvala 2 a více hodin. Výsledky nejsou nijak zajímavé, všichni hráči měli skóre kolem 10 - 30. Pouze jeden měl 156 a bohužel jsem ho neviděl hrát.

5.3.4 Experiment č. 2

Pro druhý experiment jsem musel zásadně změnit přístup, aby šlechtění nezabralo několik dní. Opakované počítání zdatnosti jsem zavrhl. Bylo by vhodné bota spouštět několikrát i v jedné generaci a použít třeba průměr výsledků, ale to by znamenalo enormně prodloužit dobu vyhodnocování zdatnosti. Smířím s tím, že někteří boti pravděpodobně budou mít vysoké fitness díky náhodě, že je nikdo nezabil. Jiní malé, protože se vygenerovali na špatném míste apod. V generaci bude jen 50 potomků, a přibližně třetina vznikne mutací topologine, třetina mutací vah a třetina křížením. Tak by se v každé generaci měl vyhodnotit stejný počet jedinců.



Obrázek 5.6: Grafy fitness pro druhý experiment se `SlitherioBot`. Osa x číslo generace, osa y počet ms.

V prvním pokusu jsem v 7. generaci nastavil špatně práh pro podobnost jedinců a vznikl druh pro každého jedince zvlášť. Protože jsem neimplementoval žádnou ochranu proti velkému počtu malých druhů, velmi dlouho v populaci přežívají. Tato chyba jistě ovlivní experiment, a tak jsem souběžně pustil ještě jeden stejný.

Spousta botů často zbytečně používá akceleraci. Vždy když dorostou dostatečné velikosti, akcelerují a všechnu nasbíranou potravu tak ztratí (při akceleraci had ztrácí hmotnost). Zdá se, že tato vlastnost z populace postupně mizí, což je znakem dobrého vývoje.

Zajímavé jak někdy vznikne jedinec, který reaguje na překážky v senzorech. Pozoroval jsem jednoho, který dlouhou dobu obcházel mapu po obvodu. Bohužel se do fitness započítává jenom konečná délka hada a tak tato vlastnost nejspíš nedostala příležitost se rozrůst na další jedince.

Grafy 5.6 dvou souběžných experimentů, které trvaly 10 a 8 hodin, odhalují, že experimenty jsou spíše zklamáním. V obou případech se zdá, že fitness

roste a tedy že se jedinci něco učí, skutečnost odhaluje poslední generace, kde jsem nechal všechny jedince znovu přepočítat svou zdatnost. Během toho jsem ty nejlepší sledoval a ukázalo se, že jejich předchozí vysoké skóre bylo spíše náhodné. Při opakované hře dosáhli většinou minimálních výsledků. Ukazuje se, že náhoda zde hraje příliš velkou roli.

Fitness v obou případech spadla na cca 300. To je poměrně zajímavá délka, ale lze ji snadno nasbírat i náhodně, pokud dostatečně dlouho nepotkáte žádného nepřítele. Tímto stylem si myslím, že vznikly i extrémní hodnoty v prvním grafu.

5.3.5 Experiment č. 3

V posledním neprovedeném experimentu bych navrhl trochu podvádět a počáteční populaci jedinců vyrobit ne jako úplně propojený graf, ale protože vstupní senzory pozicemi odpovídají s kódováním z výstupních uzlů, propojit jen ty uzly, které jsou ve stejném směru. Navíc udělat méně vrstev a větší senzory, aby síť byla jednodušší a tím pádem se lépe učila. Na výstup pro akceleraci nepřivádět žádný vstup, protože to byl častý problém hráčů z předchozího experimentu. Protože budou chybět hrany, je nutné zvýšit šanci jejich generování pomocí mutací.

5.3.6 Závěr

Zásadním problémem bylo, že jsem si neuvědomil časovou náročnost výpočtu fitness. Tím se experimenty stávají těžko proveditelnými, pokud na výsledek musíte čekat spoustu hodin. Další problematikou je fakt, že opakované spouštění stejného jedince dá jiný výsledek, protože mapy se generují náhodně a bot hraje s ostatními hráči online. Tato náhodnost hraje příliš velkou roli při získávání zdatnosti a dost stěžuje učení. Jedinec, který měl jednou štěstí a dosáhl vysokého skóre se často kříží přesto, že jeho geny nejsou nejlepší.

Myslím, že by šlo navrhnout lepší fitness měřítko, které by zohledňovalo nejen konečný stav, ale i průběh hry. Například zahrnout počet překážek, kterým se vyhnul, tempo růstu apod. Problémem může být i ohromné množství genů – 250 hran a 53 uzlů. Čím větší neuronová síť, tím složitější a delší je jekí učení.

Na druhou stranu je vidět, že někteří jedinci se naučili reagovat na prostředí, alespoň vyhýbáním se některým překážkám. Viz had, který objížděl hrací pole po obvodu. Poslední navrhnutý experiment by mohl dosažené výsledky zlepšit.

Závěr

Zhodnocení

Stanovil jsem si cíl práce, který odpovídá zadání. Během rešerše a analýzy jsem nastudoval možnosti, jak učit automatické hráče hraní her přímo ve webovém prohlížeči. Po vytvoření modelu připomínající hru Agar.io v NetLogo, kde hráči byly ovládání rozhodovacími stromy, jsem se rozhodl, že pro výslednou aplikaci raději zvolím jako rozhodovací systém umělou neuronovou síť. Odpadne tak nutnost implementace hráčských strategií a rozhodovacích uzlů. ANN je pro učení hráčů obecnější, za cenu větší náročnosti na učení.

Pro implementaci jsem zvolil neuroevoluci ANN algoritmem NEAT, který nešlechtí pouze váhy spojů neuronů, ale také topologii sítě. Mutacemi mohou vznikat nové uzly i hrany grafu. NEAT zavádí inovační číslo genu, díky kterému může geny různých potomků zarovnat k sobě tak, aby věděl, které si odpovídají. To umožňuje snadné křížení. Sítě začínají v minimální formě a rozrůstají se podle potřeby.

Provedl jsem návrh knihovny, botů i vkládání scriptů do stránky. Tohoto návrhu jsem se při implementaci držel a případné změny jsem zanášel zpět do vytvořených diagramů.

Knihovnu, její GUI i dvě webové aplikace botů, využívající této knihovny, jsem napsal v programovacím jazyku TypeScript. TS je nadmnožina JavaScriptu. Zavádí typy, třídy, rozhraní, moduly a další tak, aby se s ním pracovalo jako s třídně orientovaným OOP jazykem. TS jde dobře dohromady s balíčkovacím managerem npm a gulp.js task managerem, které jsem při implementaci využíval.

Rozhodování jak hrát hru je třeba neustále opakovat. Protože je JS interpretovaný jazyk, není možné to provádět v nekonečném cyklu, to by totiž blokovalo vykonávání ostatního kódu. K tomuto účelu využívám funkci `setInterval`, která ve zvoleném intervalu opakovaně volá definovanou funkci. Z takové funkce lze vrátit hodnota (vypočtená zdatnost) například použitím callbacku. Moderní způsob je použití třídy `Promise`, kterého využívám i já.

Testoval jsem pomocí experimentů, které potvrdili, že knihovna je funkční a dává dobré výslesdky na jednoduchém benchmarkovacím úkolu. Pro tento účel jsem vytvořil jednoduchou hru na principech problému vyvážení převráceného kyvadla. Experimenty na hře Slither.io však nedopadly příliš dobře. Vysvětlení může být několik od malé populace, špatného nastavení evolučního cyklu, nebo neideálního kódování vstupů a výstupů ANN. Proto jsem navrhl jeden další experiment, který by mohl dopadnout lépe. Bohužel opakovat experimenty je náročné, protože doba učení je enormní (až desítky hodin).

Přestože jsem se přesvědčil, že strojové učení je náročné téma, splnil jsem vytyčené cíle. Vypracoval jsem svojí doposud největší aplikaci v TS, naučil se základy AngularJS a používání task manageru gulp.js pro build a distribuci. Co stojí za povšimnutí je fakt, že vše funguje uvnitř webového prohlížeče a je tak dostupné komukoliv. Dále jsem se dozvěděl mnohem více o šlechtění neuronových sítí a co je důležité, alespoň částečně jsem si splnil svůj programátorský sen vytvořit učenlivý program, který dokáže „žít vlastním životem“.

Možnosti pokračování

Na práci lze navázat v několika směrech. Prvním je udělat další experimenty nad již existujícími automatickými hráči. Jeden pro Slither.io jsem dokonce navrhl. Knihovna umožňuje nastavovat mnoho parametrů a tak lze zkoumat jejich vliv na vývoj generací. Možností je i implementovat vlastní automatické hráče pro jiné hry a spouštět experimenty na nich.

Druhým směrem je pak vylepšování samotné knihovny. Například není implementována kontrola, zda v jedné generaci nevznikl mutací stejný gen. Takový gen by měl mít v obou potomcích stejné inovační číslo. Protože experimenty mohou trvat opravdu dlouho, bylo by vhodné implementovat funkci uložení a nahrání populace. Co by se také mohlo hodit je možnost nechat vybraného jedince opakovat výpočet fitness. Třeba z důvodů, kdy chcete opakovaně vidět jak hru odehrál.

Literatura

[1] IBM: Deep Blue. *IBM100* [online]. 2016, [cit. 2016-04-05]. Dostupné z: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

[2] leglessmoof: IBM Deep Blue [online]. Prosinec 2011, [cit. 2016-04-05]. Dostupné z: http://pictures.leglessmoof.net/v/sf2011/comphistory/P1140402.JPG.html?g2_imageViewsIndex=1

[3] IBM: FAQs Deep Blue. *IBM Research* [online]. Květen 1997, [cit. 2016-04-05]. Dostupné z: https://www.research.ibm.com/deepblue/meet/html/d.3.3a.html

[4] Guru, G. G.: DeepMind AlphaGo vs Lee Sedol. *Go Game Guru* [online]. 2016, [cit. 2016-04-05]. Dostupné z: https://www.research.ibm.com/deepblue/meet/html/d.3.3a.html

[5] David Silver, D. H.: AlphaGo: Mastering the ancient game of Go with Machine Learning. *Google Research Blog* [online]. Leden 2016, [cit. 2016-04-05]. Dostupné z: https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html

[6] SethBling: MarI/O - Machine Learning for Video Games. *YouTube* [online]. Červen 2015, [cit. 2016-04-05]. Dostupné z: https://www.youtube.com/watch?v=qv6UVOQ0F44

[7] SethBling: NEATEvolve.lua. *Pastebin.com* [online]. Červen 2015, [cit. 2016-04-05]. Dostupné z: http://pastebin.com/ZZmSNaHX

[8] Stanley, K. O.; Miikkulainen, R.: Evolving Neural Networks through Augmenting Topologies [online]. 2002, [cit. 2016-04-05]. Dostupné z: http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

[9] R3d193: Agar.io. *Agar.io Wikia* [online]. Květen 2016, [cit. 2016-05-17]. Dostupné z: http://agario.wikia.com/wiki/Agar.io

[10] Team, S.: Agar.io. *SimilarWeb* [online]. Listopad 2016, [cit. 2016-11-15]. Dostupné z: https://www.similarweb.com/website/agar.io

[11] Mayo, M.: Top Machine Learning Libraries for Javascript [online]. Červen 2015, [cit. 2015-10-15]. Dostupné z: http://www.kdnuggets.com/2016/06/top-machine-learning-libraries-javascript.html

[12] Fork, G.: User scripts for agar.io [online]. 2015, [cit. 2015-10-15]. Dostupné z: https://greasyfork.org/en/scripts/by-site/agar.io

[13] Moisan, J.-D.: GitHub - Apostolique/Agar.io-bot. 2015, [cit. 2015-10-15]. Dostupné z: https://github.com/Apostolique/Agar.io-bot

[14] Brunner, N.: Agar.io - a fascinating bot - YouTube. Červen 2015, [cit. 2015-10-15]. Dostupné z: https://www.youtube.com/watch?v=5F-2krk8EOc

[15] Brownlee, J.: A Tour of Machine Learning Algorithms. *Machine Learning Mastery* [online]. Listopad 2013, [cit. 2016-9-25]. Dostupné z: http://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/

[16] Sims, G.: What is machine learning? *Android Authority* [online]. Červenec 2015, [cit. 2016-9-25]. Dostupné z: http://www.androidauthority.com/what-is-machine-learning-621659/

[17] Pavel Kordík; Martin Šlapák: *Metody výpočetní inteligence (MI-MVI)* (nepublikované přednášky), praha, ČVUT, 2011.

[18] Gringer: overfitting [online]. Říjen 2007, [cit. 2017-01-05]. Dostupné z: https://commons.wikimedia.org/w/index.php?curid=2959742

[19] Řehořek, T.: *Základy umělé inteligence (BI-ZUM)* (nepublikované přednášky), praha, ČVUT, 2014.

[20] Schmidt, J.: *Problémy a algoritmy (MI-PAA)* (nepublikované přednášky), praha, ČVUT, 2011.

[21] Illich, M.: Strojové učení z rychlíku. *SlideShare* [online]. 2013, [cit. 2016-11-12]. Dostupné z: http://www.slideshare.net/michalillich/prez-develcz3

[22] İbrahim Sidat: RULES OF AGAR.IO GAME. *AGAR.IO GAME* [online]. Květen 2015, [cit. 2016-5-4]. Dostupné z: http://www.agariogame.com/rules-of-agar-io-game/

[23] Wilensky, U.: NetLogo User Manual [online]. 2016, [cit. 2016-5-4]. Dostupné z: https://ccl.northwestern.edu/netlogo/docs/

[24] jsx; galambalazs; Perre Neter; aj.: About JavaScript. *MDN* [online]. Květen 2016, [cit. 2016-5-21]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

[25] Mohamed Hegazy; Dan Quirk; Daniel Rosenwasser; aj.: TypeScript. *GitHub* [online]. Květen 2016, [cit. 2016-5-21]. Dostupné z: https://github.com/Microsoft/TypeScript

[26] Isadora, I.: Augmented Browsing with Firefox. *Teksocial* [online]. Květen 2013, [cit. 2016-10-11]. Dostupné z: http://teksocial.com/socialblog/2013/5/9/augmented-browsing-with-firefox.html

[27] Systems, S.: Enterprise Architect - UML Design Tools and UML CASE tools for software development. 2017, [cit. 2017-2-16]. Dostupné z: http://www.sparxsystems.com/products/ea/

[28] JGraph: Flowchart Maker and Online Diagram Software. 2017, [cit. 2017-2-16]. Dostupné z: https://www.draw.io/

[29] JetBrains: IntelliJ IDEA the Java IDE. 2017, [cit. 2017-2-16]. Dostupné z: https://www.jetbrains.com/idea/

[30] Google: Chrome for Desktop. 2017, [cit. 2017-2-16]. Dostupné z: https://www.google.com/chrome/

[31] Brownlee, J.: THE POLE BALANCING PROBLEM. A Benchmark Control Theory Problem [online]. 2005, [cit. 2016-04-05]. Dostupné z: https://pdfs.semanticscholar.org/3dd6/7d8565480ddb5f3c0b4ea6be7058e77b4172.pdf

[32] Miller, J.: Slither.io Bot Championship Edition [online]. Červenec 2016, [cit. 2017-01-15]. Dostupné z: https://github.com/j-c-m/Slither.io-bot

Příloha **A**

Seznam použitých zkratek

AI Artificial Inteligence

ANN Artificial Neural Network

API Application Programming Interface

DOM Document Object Model

DT Decision Tree

FF Firefox

IDE Integrated Development Environment

IE Internet Explorer

JS JavaScript

NEAT Neuroevolution through Augmenting Topologies algorithms

npm Node Package Manager

OOP Object Oriented Programming

RW read-write

TS TypeScript

TWEANN Topology and Weight Evolving Artificial Neural Network algorithms

Příloha B

Obsah přiloženého CD

```
readme.txt ............................ popis obsahu CD
thesis.pdf ................................ text práce
src
  thesis .............. zdrojová forma práce ve formátu LaTeX
  impl ............................ zdrojové kódy projektu
    tsneatevocycle ............................. projekt
      dist ......................... zkompilované soubory
      src ............................ zdrojové soubory
```