VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Využití Neuroevoluční metody NEAT
Using Neuroevolution Method NEAT

2015 Tereza Kovalová

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tereza Kovalová**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Využití Neuroevoluční metody NEAT
Using Neuroevolution Method NEAT

Zásady pro vypracování:

Práce bude zaměřena na využití metody optimalizace neuronové sítě pomocí evolučního algoritmu - metoda Neuroevolution of augmenting topologies (dále jen NEAT)

Cíl a obsah práce:
1. Analýza a popis jednotlivých metod, ze kterých metoda NEAT vychází - Neuronové sítě, Evoluční algoritmy.
2. Seznámení se s metodou NEAT - praktické využití, přínos, výhody a srovnání s předešlými metodami.
3. Popis existujících implementací metody NEAT pro různé platformy, a to především implementace SharpNeat pro jazyk C#, její možnosti a srovnání s ostatními implementacemi.
4. Implementace vlastní aplikace využívající implementace SharpNeat. Aplikace bude zaměřena na optimalizaci protivníka zvolené deskové či logické hry.
5. Dokumentace a popis struktury aplikace.

Seznam doporučené odborné literatury:

[1] Kenneth O. Stanley, Risto Miikkulainen: Evolving Neural Networks through Augmenting Topologies, Evolutionary Computation, 2002, PMID: 12180173
Daniel Ashlock: Evolutionary Computation for Modeling and Optimization, Springer, ISBN 0-387-22196-4
[2] Kevin Gurney: An Introduction to Neural Networks London, Routledge, 1997, ISBN 1-85728-503-4
[3] SharpNeat [online]. 2004, last update 2013-04-29 [cit. 29-9-2013]. Dostupné z WWW: <http://sharpneat.sourceforge.net>

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Buriánek**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2015

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky 1 OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 7. 5. 2015

Tereza Kovalová
....

Ráda bych poděkovala panu Ing. Tomáši Buriánkovi za odborné vedení a cenné rady, které mi pomohly zpracovat tuto bakalářskou práci.

Abstrakt

Tato bakalářská práce se zabývá optimalizací protivníka zvolené logické hry Lodě. Optimalizace byla provedena pomocí metody NEAT a implementace SharpNeat pro jazyk C#. Práce popisuje neuronové sítě, evoluční algoritmy a metodu NEAT.

Klíčová slova: neuronové sítě, evoluční algoritmy, perceptron, NEAT, SharpNeat, C#, optimalizace neuronové sítě

Abstract

The bachelor thesis deals with an adversary optimization of chosen logical game The Ships. The optimization was made by NEAT method and SharpNeat implementation for language C#. This thesis describes neural networks, evolution algorithms and NEAT method.

Keywords: neural networks, evolution algorithms, perceptron, NEAT, SharpNeat, neural network optimization

Seznam použitých zkratek a symbolů

NEAT	–	Neuroevolution of augmenting topologies
Xml	–	eXtensible Markup Language
GUI	–	Graphic User Interface
XOR	–	eXclusive OR

Obsah

1 Úvod 5

2 Neuronové sítě 6
2.1 Organizační dynamika 6
2.2 Aktivní dynamika 6
2.3 Adaptivní dynamika 7
2.4 Učení neuronové sítě 7

3 Síť Perceptronů 9
3.1 Organizační dynamika 9
3.2 Aktivní dynamika 9
3.3 Adaptivní dynamika 9

4 Vícevrstvá síť a backpropagation 10
4.1 Organizační a aktivní dynamika 10
4.2 Adaptivní dynamika 10
4.3 Backpropagation 10

5 Evoluční algoritmy 12
5.1 Mutace 12
5.2 Křížení 12
5.3 Selekce 12

6 Neat 14

7 SharpNeat 17
7.1 Java Neat 17
7.2 Python Neat 17
7.3 Ruby NEAT 18

8 Uživatelská dokumentace 19
8.1 Spuštění hry 19
8.2 Spuštění evoluce 19
8.3 Spuštění koevoluce 19
8.4 Spuštění statistiky 19

9 Programátorská dokumentace 24

10 Statistika 31

11 Závěr 33

12 Reference 34

13 Přílohy **35**

Seznam výpisů zdrojového kódu

1 Třída Buňka. 24
2 Třída Loď. 25
3 Třída PomocnéMetody. 25
4 Třída Výstřel. 27
5 Příklad uložené optimalizované neuronové sítě. 27

Seznam obrázků

1	Příklad acyklické architektury.	8
2	Příklad architektury vícevrstvé neuronové sítě 3-4-3-2.	8
3	Mutace v NEATu.	15
4	Zápasící genomy.	16
5	GUI hlavní okno.	21
6	GUI okno evoluce.	21
7	GUI nová hra.	22
8	GUI ukázka ze hry.	22
9	GUI evoluce.	23
10	GUI statistika.	23
11	Třídní diagram.	29
12	Sekvenční diagram.	30
13	Výsledek statistiky	32

1 Úvod

Neuronová síť je jedním z výpočetních modelů, které se využívají v umělé inteligenci. Neuronová síť se skládá z umělých neuronů, které mají libovolný počet vstupů a jeden výstup. Umělý neuron je inspirován biologickým neuronem.

V biologických sítích jsou zkušenosti, které síť nabývá učením, uloženy v dendridech, zatímco v umělé neuronové síti jsou uloženy ve váhách.

Evoluční algoritmy se nejčastěji používají k řešení problémů optimalizace (např. učení neuronové sítě). Evoluční algoritmy využívají celou populaci prozatimních řešení, jež paralelně procházejí parametrický prostor a navzájem se ovlivňují a modifikují pomocí genetických operátorů.

Hlavním cílem této bakalářské práce je optimalizace protivníka v logické hře Lodě pomocí implementace metody NEAT pro C#.

2 Neuronové sítě

Neuronová síť je tvořena vzájemně propojenými neurony, kde výstup neuronu je vstupem více neuronů [1].

Počet neuronů a jejich vzájemné propojení určuje tzv. architekturu (topologii) neuronové sítě. V neuronové síti rozlišujeme neurony z hlediska jejich použití na vstupní, pracovní a výstupní. Stav neuronové sítě určují stavy všech neuronů v síti a konfiguraci neuronové sítě určují váhy všech spojů v neuronové síti.

Neuronová síť se v čase vyvíjí, mění se propojení a stav neuronů, adaptují se váhy [1]. V souvislosti se změnou charakteristik je účelné celkovou dynamiku neuronové sítě rozdělit do tří dynamik a pak uvažovat nad třemi režimy práce sítě: organizačním (změna topologie), aktivním (změna stavu) a adaptivním (změna konfigurace).

2.1 Organizační dynamika

Organizační dynamika neuronové sítě specifikuje architekturu sítě a její případnou změnu [1]. Změna topologie rozšířením neuronové sítě o další neurony a příslušné spoje se uplatňuje v adaptivním režimu. Rozlišujeme dva typy architektury neuronové sítě: cyklickou (resp. rekurentní) a acyklickou 1 (resp. dopřednou) síť. V acyklické architektuře je skupina neuronů zapojená do kruhu (cyklu). To znamená, že výstup prvního neuronu je vstupem druhého neuronu, výstup druhého neuronu je vstupem třetího neuronu atd., výstup posledního neuronu je vstupem prvního neuronu.

2.2 Aktivní dynamika

Aktivní dynamika specifikuje počáteční stav sítě a způsob jeho změny v čase při pevné topologii a konfiguraci [1]. V aktivním režimu se na začátku nastaví stavy vstupních neuronů na tzv. vstup sítě a zbylé neurony jsou v počátečním stavu. Vstupní prostor resp. stavový prostor neuronové sítě tvoří všechny vstupy resp. stavy sítě. Po inicializaci stavu sítě probíhá vlastní výpočet [1]. Obecně se uvažuje o spojitém modelu, kde se uvažuje v čase o spojitém vývoji neuronové sítě. Většinou se však předpokládá diskrétní čas a síť se nachází v čase 0 a stav sítě se mění jen v čase 1, 2, 3, . . . V každém takovém časovém kroku je podle pravidla aktivní dynamiky vybrán jeden neuron (tzv. sekvenční výpočet) nebo více neuronů (tzv. paralelní výpočet), které aktualizují (mění) svůj stav na základě svých vstupů, tj. stavů sousedních neuronů, jejichž výstupy jsou vstupy aktualizovaných neuronů [1]. Podle toho, jestli neurony mění svůj stav nezávisle na sobě nebo je jejich aktualizace řízena centrálně, rozlišujeme asynchronní a synchronní modely neuronových sítí. Stav výstupních neuronů, který se obecně mění v čase, je tzv. výstupem neuronové sítě (tj. výsledkem výpočtu) [1]. Obvykle se uvažuje taková aktivní dynamika neuronové sítě, že výstup sítě je po nějakém čase konstantní a neuronová síť si tak v aktivním režimu realizuje nějakou funkci a ke každému vstupu sítě vypočítá právě jeden výstup. Funkce neuronové sítě je dána aktivní dynamikou, jejíž rovnice je parametricky závislá na konfiguraci a topologii, které se v aktivním režimu mění. Neuronová síť se používá k vlastním

výpočtům v aktivním režimu. Aktivní dynamika neuronové sítě také určuje funkci jednoho neuronu, jejíž předpis (matematický vzorec) je většinou pro všechny (nevstupní) neurony v síti stejný (tzv. homogenní neuronová síť) [1].

2.3 Adaptivní dynamika

Adaptivní dynamika specifikuje počáteční konfiguraci sítě a jakým způsobem se mění váhy v síti v čase [1]. Veškeré konfigurace tvoří tzv. váhový prostor neuronové sítě. V adaptivním režimu se nastaví váhy všech spojů v neuronové síti na počáteční konfiguraci. Po inicializaci konfigurace sítě probíhá vlastní adaptace [1]. Funkce neuronové sítě v aktivním režimu závisí na konfiguraci. Cílem adaptace je nalézt takovou konfiguraci sítě ve váhovém prostoru, která by v aktivním režimu realizovala předepsanou funkci [1]. Jestliže aktivní režim neuronové sítě slouží k výpočtu funkce neuronové sítě pro daný vstup, pak adaptivní režim slouží k učení této funkce.

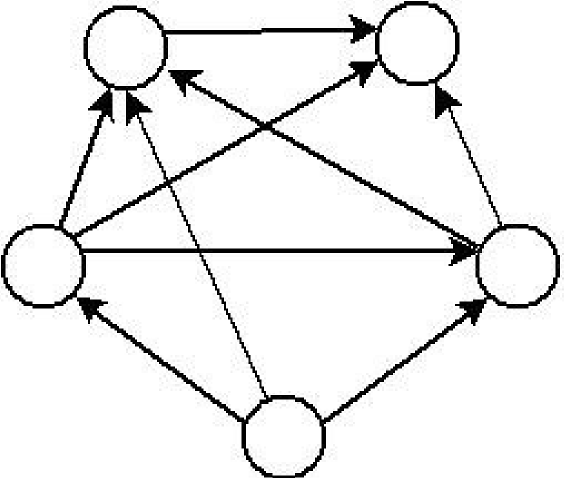
2.4 Učení neuronové sítě

K naučení neuronové sítě potřebujeme tzv. trénovací množinu, která obsahuje vzorky popisující řešenou problematiku a metodu, která dokáže vzorky z trénovací množiny zafixovat do neuronové sítě formou hodnot synaptických vah. Trénovací T množinu můžeme definovat jako množinu prvků (vzorů), které jsou definovány jako uspořádané dvojice trénovací množiny 1, vektoru excitací vstupní vrstvy 2 a vektoru excitací výstupní vrstvy 3 :

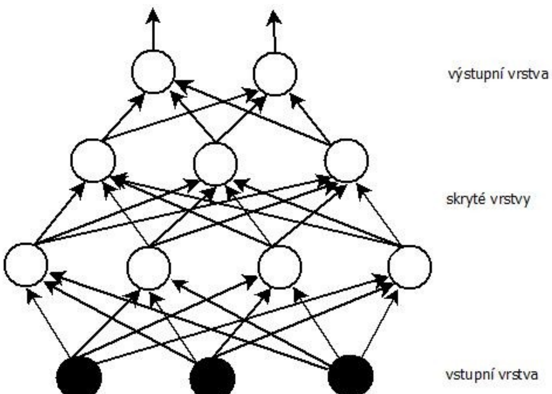
$$T = \{\{I_1, O_1\}\ \{I_2, O_2\}\ \ldots\ \{I_p, O_p\}\} \tag{1}$$

$$I_i = [i_1\ i_2\ \ldots\ i_k]\,,\ i_j \in \langle 0, 1\rangle \tag{2}$$

$$O_i = [o_1\ o_2\ \ldots\ o_m]\,,\ o_j \in \langle 0, 1\rangle \tag{3}$$



Obrázek 1: Příklad acyklické architektury.



Obrázek 2: Příklad architektury vícevrstvé neuronové sítě 3-4-3-2.

3 Síť Perceptronů

Historicky prvním úspěšným modelem neuronové sítě byla síť perceptronů, která se používá dodnes [1]. V historii neuronových sítí popsal Perceptron v roce 1958 Frank Rosenblatt. Potenciál perceptronu je definován jako vážený součet vstupních signálů.

3.1 Organizační dynamika

Na začátku organizační dynamika sítě perceptronů specifikuje pevnou architekturu jednovrstvé sítě n-m. To znamená, že se síť perceptronů skládá z n vstupních neuronů, kde je každý vstupní neuron vstupem každého z m výstupních neuronů.

3.2 Aktivní dynamika

Způsob výpočtu funkce sítě perceptronů určuje aktivní dynamika. Reálné stavy neuronů ve vstupní vrstvě se nastaví na vstup sítě a výstupní neurony počítají svůj binární stav, který určuje výstup sítě.

3.3 Adaptivní dynamika

V adaptivním režimu je požadovaná funkce sítě perceptronů tréninkovou množinou. Síť perceptronů má diskrétní adaptivní dynamiku. Na začátku adaptace v čase 0 jsou váhy konfigurace nastaveny náhodně blízko nuly. V každém časovém kroku učení t = 1, 2, 3, . . . je síti předložen jeden vzor z tréninkové množiny a síť se ho snaží naučit, tj. adaptuje podle něj své váhy. Tréninková strategie určuje pořadí vzorků při učení. Adaptace sítě perceptronů obvykle probíha v tzv. tréninkových cyklech, ve kterých se systematicky prochází všechny vzory tréninkové množiny (popř. každý vzor i vícekrát za sebou). Rychlost učení je mírou vlivu vzorů na adaptaci. Většinou se na začátku volí malá rychlost učení, která později během adaptace roste.

Význam perceptronové sítě je spíše teoretický, protože perceptronová síť může počítat jen omezenou třídu funkcí. Tento jednoduchý model je základem složitějších modelů, jako je obecná vícevrstvá síť s učícím algoritmem backpropagation. Síť perceptronů lze použít jen v případě, kdy jsou klasifikované objekty ve vstupním prostoru oddělitelné nadrovinou (např. speciální úlohy rozpoznávání obrazu apod.).

4 Vícevrstvá síť a backpropagation

Vícevrstvá neuronová síť 2 je tvořena minimálně třemi vrstvami neuronů: vstupní, výstupní a nejméně jednou vnitřní vrstvou. Mezi dvěma sousedními vrstvami se nachází úplné propojení neuronů, kde každý neuron nižší vrstvy je spojen se všemi neurony vyšší vrstvy.

Vícevrstvá neuronová síť s účícím algoritmem zpětného šíření chyb je nejznámějším a nejpoužívanějším modelem neuronové sítě.

Zpracování informace dopředným šířením signálu:

1. Neurony vstupní vrstvy jsou excitovány na odpovídající úroveň (v rozmezí 0 až 1).
2. Tyto excitace jsou přivedeny k následující vrstvě neuronů a zeslabeny nebo zesíleny pomocí synaptických vah.
3. Každý neuron vyšší vrstvy provede sumaci signálů od nižší vrstvy a je excitován na úroveň danou svou aktivační funkcí.
4. Tento proces probíhá přes veškeré vrstvy neuronů až k výstupní vrstvě, kde získáme excitační stavy všech jejích neuronů.

4.1 Organizační a aktivní dynamika

Organizační dynamika vícevrstvého perceptronu specifikuje pevnou topologii vícevrstvé neuronové sítě. Standardně se používá dvou resp. třívrstvá síť.

4.2 Adaptivní dynamika

Adaptivní režim vícevrstvé sítě probíhá podobně jako u sítě perceptronů. Požadovaná funkce je zadána tréninkovou množinou a chyba sítě vzhledem k tréninkové množině je definována jako součet parciálních chyb sítě vzhledem k jednotlivým tréninkovým vzorům a závisí na konfiguraci sítě.

4.3 Backpropagation

Metoda zpětného šíření umožňuje adaptaci neuronové sítě nad danou trénovací množinou. Na rozdíl od dopředného šíření signálu od nižších vrstev k vyšším probíhá u této metody šíření signálu od vyšších vrstev neuronů k těm nižším.

Popis metody zpětného šíření:

1. Vezmeme vektor excitací vstupní vrstvy I_i i-tého prvku trénovací množiny, kterým excitujeme neurony vstupní vrstvy na odpovídající úroveň.
2. Provedeme dopředné šíření signálu až k výstupní vrstvě neuronů.

3. Srovnáme požadovaný stav daný vektorem excitací výstupní vrstvy O_i i-tého prvku trénovací množiny se skutečnou odezvou neuronové sítě.

4. Rozdíl mezi skutečnou a požadovanou odezvou definuje chybu neuronové sítě. Tuto chybu pak vracíme zpět do neuronové sítě formou úpravy synaptických vah mezi jednotlivými vrstvami neuronů směrem od vyšších k nižším, aby byla chyba při následující odezvě menší.

5. Jakmile dojde k vyčerpání celé trénovací množiny, tak vyhodnotí se celková chyba přes veškeré vzory trénovací množiny a jestliže je chyba vyšší než požadovaná, tak se celý proces opakuje.

5 Evoluční algoritmy

Evoluční algoritmy používají evoluční model, který se skládá z metody výběru rodičů a metody vkládání potomků zpět do populace.

Globální optimum je bod ve fitness prostoru jehož hodnota převyšuje všechny ostatní. Lokální optimum je bod ve fitness prostoru, který má tu vlastnost, že žádný řetězec mutací v tomto bodu nemůže jít nahoru aniž by nešel nejprve dolů.

Když se členové populace s nejvyšším fitness se udrží v evolučním algoritmu, tak se algoritmus nazývá algoritmus elitářství. Tito garantovaní členové populace se nazývají elita. Elitismus zaručuje, že populace s fixní fitness funkcí se v pozdějších generacích nemohou zhoršit na menší maximální fitness a v budoucnu budou mít více potomků a jejich gen bude v populaci převládat. Tato nadvláda může zhoršit hledání prostoru genů, protože aktuální elita nemusí obsahovat všechny geny potřebné pro co nejlepší stvoření. Jako kompromis je dobré mít malou elitu.

Model evoluce potřebuje metody pro vkládání potomků. Má-li zůstat počet členů populace stejný, pak se musí udělat místo pro každého potomka. Jeden ze způsobů vložení potomka do populace je nahrazení jakéhokoliv člena populace. Tento zbůsob se nazývá náhodná výměna. Dále existuje zbůsob rulety, kde volíme člena, který bude nahrazen s pravděpodobností nepřímo úměrné k jeho fitness. U absolutního fitness nahrazení nahrazujeme nejméně zdatného člena populace. Další možností jak uložit potomky, je nahrazení rodičů potomky v případě, že jsou potomci zdatnější než rodiče. V této metodě nazvané výměna elity, se zkoumají oba rodiče a jejich dva potomci, dva nejvíce hodící se se vloží mezi rodiče. V náhodné elitní výměně se každý potomek srovnává s náhodně vybraným členem populace a nahrazuje jej pouze v případě, že je alespoň stejně tak dobrý.

5.1 Mutace

Operátor mutace v populaci genů G je funkce, která přenáší gen na jiný další podobný, ale rozdílný gen. $Mutace : G \rightarrow G$ Operátor mutace se také nazývá unární variační operátor.

5.2 Křížení

Operátor křížení pro soubor genů je mapován jako křížení $G \times G \rightarrow G \times G$ nebo $G \times G \rightarrow G$. Body tvořící páry v hlavním prostoru křížení jsou označovány jako rodiče, zatímco body v následující části jsou označovány jako potomci. Očekává se, že potomci nesou některé části rodičovských struktur.

5.3 Selekce

Jeden turnaj výběru má elitu o velikosti dva. Polovina populace přežije, ale dvě nejschopnější stvoření musí přežít. Jiné stvoření přežijí pouze v případě, že jsou ve skupině s horšími bytostmi.

Dvojitý turnaj výběru s n turnaji a vybereme skupinu n bytostí a vybereme jednu nejvhodnější jako rodiče a celý proces opakujeme pro výběr druhého rodiče. Dvojitý turnaj výběru může být proveden s výměnou (stejný rodič může být vybrán dvakrát) nebo bez výměny (stejného rodiče nelze vybrat dvakrát, protože je první rodič při výběru druhého vyloučený).

Výběr ruleta vybere rodiče v přímé úměře jejich fitness. Pokud bytost má fitness fi, pak pravděpodobnost, že je vybrán jako rodič je fi/F, kde F je součet fitness celé populace.

Výběr podle hodnosti funguje obdobně jako výběr rulety s vyjímkou toho, že stvoření jsou seřazeny podle fitness a pak se vyberou podle hodnosti místo vhodnosti. Pokud bytost i má hodnost fi, pak pravděpodobnost, že bude vybrána jako rodič je fi/F, kde F je součet hodností celé populace.

6 Neat

Metoda NEAT využívá tři klíčové techniky: sledování historického značení genů, použití evoluce druhů a postupný rozvoj topologie od jednoduchých původních druhů.

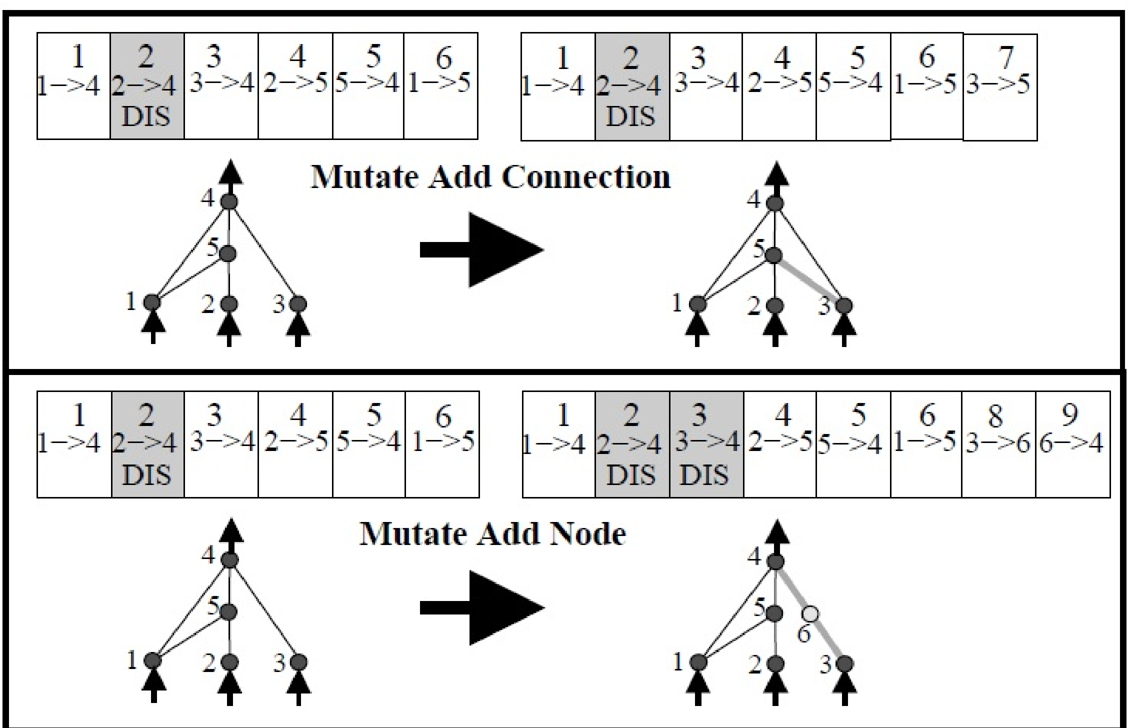
Speciace umožňuje členům populace nejprve soutěžit v rámci svého druhu a až poté v rámci celé populace. Cílem je rozdělit populaci do takových druhů, aby podobné topologie byly ve stejném druhu. V každé generaci jsou genomy sekvenčně seřazeny do kruhu a každý existující druh je zastoupen náhodným genomem uvnitř druhu z předchozí generace. NEAT používá explicitní sdílení fitness, kde se jednotlivci seskupují spíše podle genetické podobnosti, než podle výkonu.

Historické značení genů identifikuje původního předka každého genu. Novým genům jsou přiřazena stále vyšší čísla. Historické značení je informace, která nám řekne, které geny soutěží s kterými geny mezi jakýmikoliv jednotlivci v topologicky různorodé populaci. Dva geny se stejným historickým značením představují stejnou strukturu.

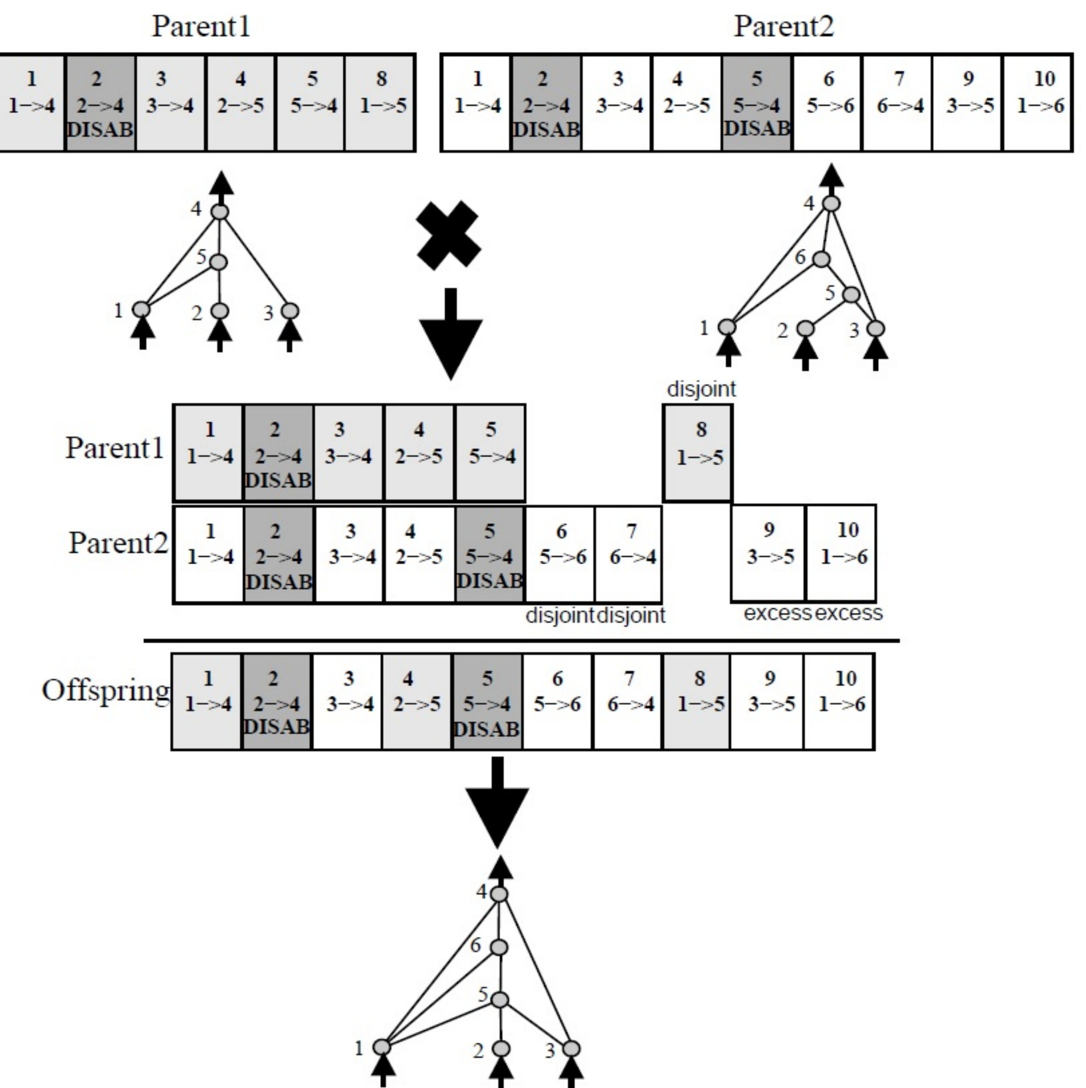
Na obrázku 3 jsou zobrazeny dva typy strukturální mutace. První mutace přidává propojení a druhá přidává uzly.

Na obrázku 4 je znázorněno jak zápasící genomy pro různé síťové topologie využívají inovační čísla. Přestože oba rodiče vypadají rozdílně, jejich inovační čísla nám říkají, které geny zápasí s kterými. Nová struktura, která kombinuje překrývající se části obou rodičů, může být vytvořena stejně, jako jejich různé části a to bez jakýchkoliv topologických analýz. Odpovídající geny se dědí náhodně, zatímco nespojené geny a přebytečné geny se dědí z rodiče s větší fitness.

Metoda NEAT využívá přímé kódování a začíná bez skrytých uzlů. Klíčovou myšlenkou NEATu není konečná struktura řešení, ale spíše struktury všech řešení v průběhu hledání řešení. Konektivita každého přechodného řešení představuje prostor parametru a čím více propojení existuje, tím více je potřeba optimalizovat parametry. Z tohoto důvodu může být množství struktur minimalizované skrz evoluci. Navíc může být zkoumána i rozměrnost prostorů, což vede ke značnému zvýšení výkonu. Hlavní pohled na NEAT je, že historický původ dvou genů je přímý důkaz homologie v případě, že geny mají stejný původ. NEAT provádí umělé synapse na základě historických značení, což umožňuje přidávat nové struktury, aniž by ztratily přehled o tom, který gen je který v průběhu simulace. NEAT začíná s minimální populací a rostoucí strukturou. Historické značení umožňují NEATu provádět křížení lineárními genomy bez nutnosti topologické analýzy. Minimalizace rozměrů dává NEATu výkonnostní výhodu ve srovnání s ostatními. NEAT vyvíjí mnoho různých struktur různých druhů najednou, z nichž každý reprezentuje prostor různých rozměrů. NEAT se vždy snaží vyřešit problém okamžitě, takže je méně pravděpodobné, že uvázne. NEAT je schopen provádět křížení i v případě, že genomy mají různou velikost. Vzhledem k tomu, že NEAT může vždy přidávat nové struktury a nezastaví se jako jiné metody v v případech, kdy globální optima jsou výrazně odlišná od globálních optim, je vhodný zejména pro problémy, kde je pravděpodobné, že jiné metody uváznou. NEAT je účinný způsob pro uměle se vyvíjející neuronové sítě a ukazuje, že vyvíjející se topologie spolu s hmotností může být velkou výhodou.



Obrázek 3: Mutace v NEATu.



Obrázek 4: Zápasící genomy.

7 SharpNeat

SharpNeat je framework napsaný v jazyce C#. Framework je možné stáhnout z webu [11]. Pro použití frameworku SharpNeat v projektu potřebujeme přidat do projektu reference na knihovnu SharpNeatDomains a SharpNeatLib.

Abychom mohli vytvořit vlastní aplikaci, která bude ohodnocovat neuronovou síť pomocí evoluce, tak musíme nejprve vytvořit třídu, která bude implementovat rozhraní `INeatExperiment`, tato třída nám pak poskytne metody pro vytvoření evolučního algoritmu, metodu pro vytváření genomů a metodu pro dekódování genomů.

Dále musíme vytvořit třídu pro evaluaci, která bude implementovat rozhraní `IPhenomeEvaluator`. Tato třída nám poskytne prostředky pro učení neuronové sítě. V metodě `Evaluate` necháme hrát neuronovou síť proti náhodnému nebo ideálnímu hráči a neuronová síť se bude každou odehranou hrou zdokonalovat.

V případě, že máme třídu pro evaluaci a třídu pro experiment již připravenou, tak si vytvoříme instanci třídy pro experiment. Na vytvořený objekt experimentu zavoláme metodu pro inicializaci, vytvoříme instanci evolučního algoritmu `NeatEvolutionAlgorithm`, k vytvořenému objektu přidáme událost pro aktualizaci, ve které budeme při každém volání ukládat aktuální nejlepší genom současné generace. Na objekt evolučního algoritmu zavoláme metodu `Start` a spustíme evoluci ve vlákně na pozadí. Evoluce bude probíhat tak dlouho, dokud na objekt evolučního algoritmu nezavoláme metodu `Stop`.

SharpNeat také umožňuje evaluaci neuronové sítě pomocí koevoluce. Na rozdíl od evoluce třída pro evaluaci implementuje rozhraní `ICoevolutionPhenomeEvaluator` a třída pro experiment implementuje stejné rozhraní jako pro evoluci. Pro koevoluci potřebujeme třídu, která umožní paralelní evaluaci seznamu genomů a bude implementovat rozhraní `IGenomeListEvaluator`.

Vytvoření experimentu, evolučního algoritmu a spuštění a zastavení koevoluce probíhá stejně jako u evoluce.

7.1 Java Neat

Implementace NEATu pro Javu se nazývá JNEAT. JNEAT napsal Ugo Vierucci. JNEAT vychází z originálu C++, který napsal Kenneth Stanley. JNEAT na rozdíl od SharpNeatu je možné použít ve Windows i Linuxu. Implementace je dostupná ke stažení na webu [12] ve verzi v1.2 z roku 2002. Implementace obsahuje GUI a implementaci experimentů pro XOR a 3-bit parity.

7.2 Python Neat

Implementace NEATu pro Python se nazývá NEAT-Python a je ke stažení na webu [13]. PythonNEAT se využívá na růszných platformách stejně jako JNEAT. Implementace obsahuje experiment XOR.

7.3 Ruby NEAT

RubyNEAT je první implementací NEATu v jazyce Ruby. Implementace NEATu pro Ruby se nazývá RubyNEAT. Implementace je dostupná ke stažení na webu [14]. Implementace RubyNEAT se nejčastěji používá na platformě Linux. Implementace RubyNEATu obsahuje moduly: kontroler, evaluator, populaci.

8 Uživatelská dokumentace

Aplikace umožňuje hraní hry Lodě proti náhodnému hráči nebo optimalizovanému hráči, který se do aplikace nahraje ze souboru. Dále aplikace slouží k vytvoření optimalizovaného hráče evolucí nebo koevolucí a jeho uložení do souboru. Program obsahuje také funkci pro statistiku tří hráčů, kteří hrají proti sobě. U statistiky uživatel zadá cestu k již uloženému optimalizovanému hráči evolucí a koevolucí, jako třetí hráč je použit náhodný hráč.

V hlavním okně se nachází herní plocha vlastní lodě, která zobrazuje rozmístění vlastních odí hráče, výstřely protihráče a herní plochu - radar, kde uživatel provádí výstřely na protihráče. Nachází se zde také čtyři tlačítka, které umožňují spustit novou hru, otevřít nové okno aplikace s evolucí, koevolucí nebo statistikou.

Okno pro evoluci a koevoluci obsahuje výpis aktuální generace a maximální ohodnocení. Před spuštěním evoluce či koevoluce má uživatel možnost nastavit počet iterací, které se provedou a zvolit umístění kam se výstup z evoluce nebo koevoluce uloží.

V okně pro statistiku se zobrazí přehled všech odehraných her. U každé hry se zobrazí dva hráči, kteří proti sobě hráli a vítěz hry. Dále okno obsahuje statistiku počtu vítězství jednotlivých hráčů. Při spuštění statistiky je uživatel vyzván, aby zvolil umístění uložených hráčů, kteří budou hrát proti sobě a proti náhodnému hráči.

8.1 Spuštění hry

Uživatel spustí aplikaci 5 a klikne na tlačítko nová hra. Zobrazí se dialog, ve kterém uživatel zvolí soubor, ve kterém je uložený optimalizovaný hráč a zvolí otevřít nebo dialog zavře a začne hra proti náhodnému hráči 7.

8.2 Spuštění evoluce

Uživatel spustí aplikaci 5 a klikne na tlačítko evoluce. Zobrazí se nové okno 6. Uživatel zvolí počet iterací a klikne na tlačítko start. Pak se zobrazí dialog, ve kterém uživatel nastaví, kam se má výsledek evoluce uložit. Po skončení každé iterace se v okně vypíšou aktuální informace o průběhu evoluce.

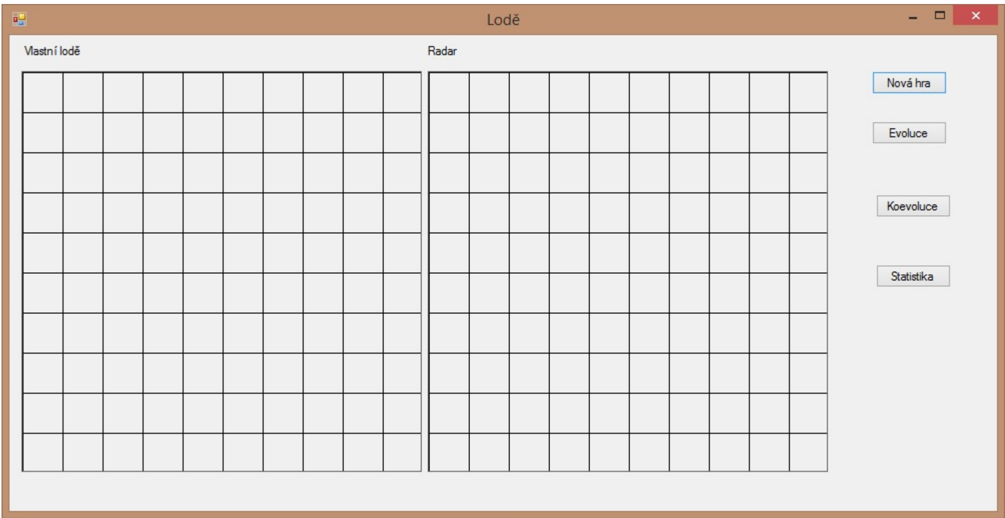
8.3 Spuštění koevoluce

Uživatel spustí aplikaci 5 a klikne na tlačítko koevoluce. Zobrazí se nové okno. Uživatel zvolí počet iterací a klikne na tlačítko start. Pak se zobrazí dialog, ve kterém uživatel nastaví, kam se má výsledek koevoluce uložit. Po skončení každé iterace se v okně vypíšou aktuální informace o průběhu koevoluce.

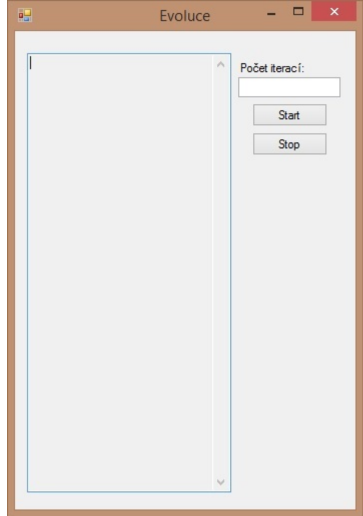
8.4 Spuštění statistiky

Uživatel spustí aplikaci 5 a v hlavním okně aplikace klikne na tlačítko statistika. Zobrazí se nové okno, kde uživatel klikne na tlačítko spustit. Zobrazí se dialog, ve kterém uživatel zvolí umístění hráče optimalizovaného evolucí a následně hráče optimalizovaného

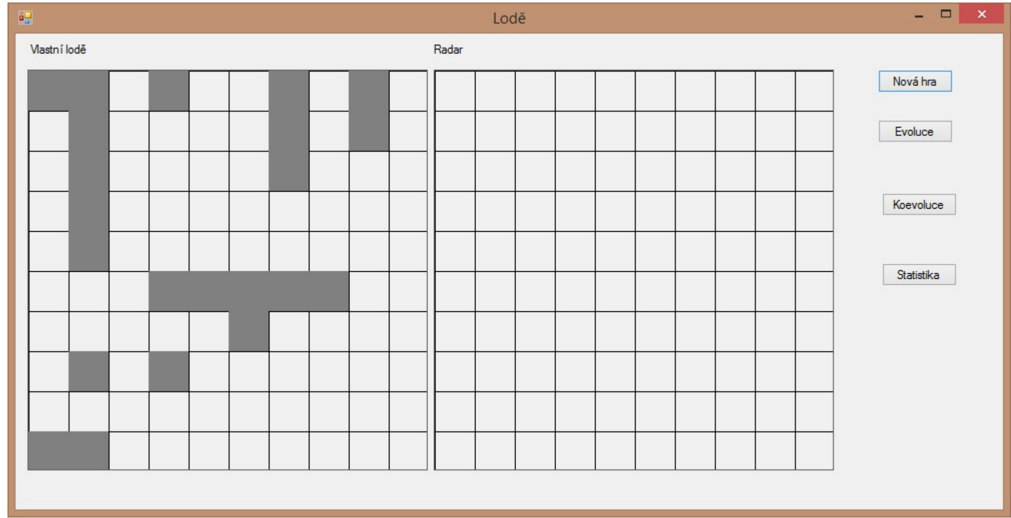
koevolucí. v okně statistiky 10 se vypíší výsledky všech odehraných her a u každého hráče se zobrazí počet her, ve kterých vyhrál.



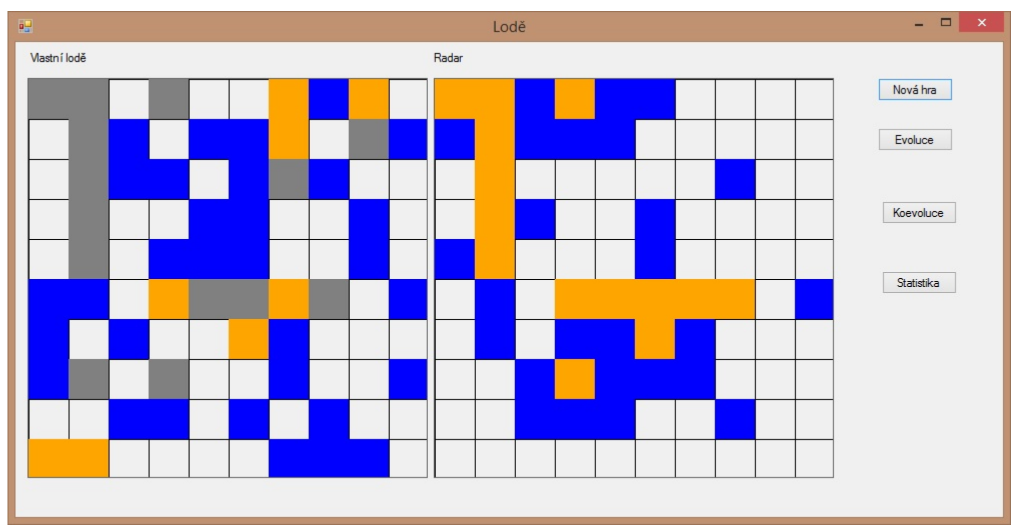
Obrázek 5: GUI hlavní okno.



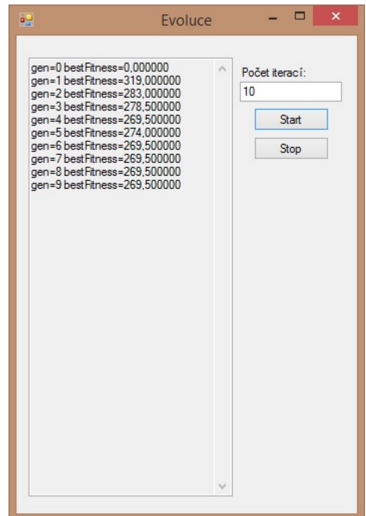
Obrázek 6: GUI okno evoluce.



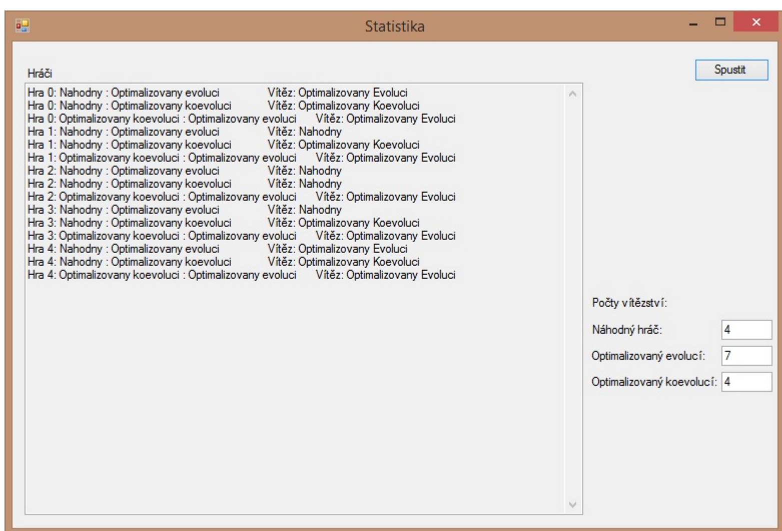
Obrázek 7: GUI nová hra.



Obrázek 8: GUI ukázka ze hry.



Obrázek 9: GUI evoluce.



Obrázek 10: GUI statistika.

9 Programátorská dokumentace

Aplikace obsahuje čtyři projekty, dvě konzolové aplikace, Windows Form aplikaci a jednu knihovnu tříd. Windows Form projekt obsahuje formuláře pro hru, evoluci, koevoluci a statistiku. Konzolové aplikace evoluce a koevoluce slouží pouze jako testovací aplikace pro evoluci a koevoluci s výpisem aktuální generace a maximálního ohodnocení do konzole.

Knihovna tříd obsahuje třídu `Hra`, která slouží ke hře uživatelského hráče a optimalizovaného nebo náhodného.

Třída `Hra` obsahuje metodu `ZačniUživatelskouHru`, která se jako parametry předá instance uživatelského hráče a instance hráče, který bude náhodný nebo optimalizovaný. V metodě se každému hráči naplní jeho hrací pole, které bude obsahovat jeho vlastní lodě a pole radar, do kterého se budou zaznamenávat výstřely protihráče. Oběma hráčům se do jejich hracího pole umístí celkem osm lodí na náhodné pozice za předpokladu, že se lodě vzájemně nedotýkají. Třída má metodu `VýstřelUživatele`, která se vyvolá, když uživatel klikne v GUI na panel radar a předá souřadnice v podobě instance třídy `Výstřel`. V metodě výstřel uživatele se pomocí metody `MohuVystřelit` dotážeme protivníka, jestli už jsme na danou pozici nestříleli dříve a v případě, že ne, tak se provede výstřel a do radaru protivníka se zaznamená výstřel. Kdybychom již na danou pozici vystřelili dříve, tak se výstřel opakuje dokud nebude úspěšný na ještě nezasáhnutou pozici.

Třída `Buňka` reprezentuje jedno políčko na hracím poli. `Buňka` uchovává informaci o souřadnicích a stavu. Stav buňky nám říká, jestli na dané pozici je voda nebo lod' a jestli byla lod' nebo voda již zasažena.

```
public class Bunka
{
    public int X { get; set; }
    public int Y { get; set; }
    public StavBunky Stav { get; set; }
    public Bunka(int x, int y, StavBunky stav)
    {
        X = x;
        Y = y;
        Stav = stav;
    }
}
```

Výpis 1: Třída Buňka.

Abstraktní třída `Hráč` slouží jako předloha pro třídy uživatelského, náhodného a optimalizovaného hráče. Třída uchovává informace o skóre hráče, jménu hráče, jeho hracím poli a radaru. Obsahuje metodu pro naplnění hracího pole a radaru buňkami, metodu `MohuVystřelit`, které se předávají souřadnice a na základě těchto souřadnic zjistí, jestli protivník na pozici již střílel. Dále obsahuje metodu `Výstřel`, které se předává parametr instance třídy `Výstřel`. Metoda zkontroluje, jestli se na dané pozici nachází voda nebo lod' a provede zásah do hracího pole a radaru. V případě, že je zasažená lod', tak zkontroluje, jestli už nebyla potopená.

Třída `HraProDva` je téměř shodná s třídou `Hra` a slouží pro hru náhodného a optimalizovaného hráče.

Třída `HraProDvaOptimalizované` slouží pro hru dvou optimalizovaných hráčů a je téměř shodná se třídou `HraProDva`. Obsahuje metodu `Hraj`, které se předají dva hráči a vrátí vítězného hráče hry.

Třída `Lod'` reprezentuje jednu lod', která obsahuje pole pozic, ve kterém jsou uloženy všechny pozice, na kterých se daná lod' nachází. U lodi uchováváme informace o její délce, o jaký typ lodi se jedná, jestli již byla položená a potopená.

```
public class Lod
    {
        public Pozice[] PolePozic;
        public TypLodi Typ { get; set; }
        public int Delka { get; set; }
        public bool Polozena { get; set; }
        public bool Potopena { get; set; }
        public Pozice this[int i]
        {
            get
            {
                return PolePozic[i];
            }

            set
            {
                PolePozic[i] = value;
            }
        }
        public Lod()
        {
            Polozena = false;
            Potopena = false;
        }
    }
```

Výpis 2: Třída Lod'.

Statická třída `PomocnéMetody` má jednu statickou metodu `VytvořLodě`, která vytvoří a vrátí seznam instancí třídy `Lod'`.

Třída `RadarBuňka` reprezentuje jedno políčko na radaru a uchovává informace o souřadnicích, na kterých se buňka nachází a stav buňky, jestli byla buňka hraná, zasažená nebo jestli se jedná o vodu.

```
public static class PomocneMetody
    {
        public static List<Lod> VytvorLode()
        {
            List<Lod> lode = new List<Lod>();

            for (int i = 0; i < 8; i++)
            {
                lode.Add(new Lod());
```

```
            }

            lode[0].Typ = TypLodi.BITEVNI;
            lode[0].Delka = 5;
            lode[0].PolePozic = new Pozice[6];

            lode[1].Typ = TypLodi.LETADLOVA;
            lode[1].Delka = 5;
            lode[1].PolePozic = new Pozice[6];

            lode[2].Typ = TypLodi.KRIZNIK;
            lode[2].Delka = 3;
            lode[2].PolePozic = new Pozice[3];

            lode[3].Typ = TypLodi.TORPEDOBOREC;
            lode[3].Delka = 2;
            lode[3].PolePozic = new Pozice[2];

            lode[4].Typ = TypLodi.TORPEDOBOREC;
            lode[4].Delka = 2;
            lode[4].PolePozic = new Pozice[2];

            lode[5].Typ = TypLodi.PONORKA;
            lode[5].Delka = 1;
            lode[5].PolePozic = new Pozice[1];

            lode[6].Typ = TypLodi.PONORKA;
            lode[6].Delka = 1;
            lode[6].PolePozic = new Pozice[1];

            lode[7].Typ = TypLodi.PONORKA;
            lode[7].Delka = 1;
            lode[7].PolePozic = new Pozice[1];

            return lode;
        }
    }
```

Výpis 3: Třída PomocnéMetody.

Třída `Statistika` slouží ke hře náhodného hráče, optimalizovaného evolucí a optimalizovaného koevolucí. Všichni tři hrají proti sobě a u každého se zaznamenává počet vítězství. Třída má metodu `HráčEvoluce`, které se předá cesta k souboru, ve kterém je uložen optimalizovaný hráč evolucí a metoda vrátí instanci optimalizovaného hráče. Metoda `HráčKoevoluce` je totožná s předchozí metodou a vrátí objekt optimalizovaného hráče koevolucí.

Třída `UživatelskýHráč` dědí z třídy `Hráč`. Obsahuje metodu pro naplnění hracího pole náhodně rozmístěnými loděmi, metodu `KontrolaDotyku`, kterou využívá předchozí metoda a kontroluje každé políčko, na kterém má být položená loď, jestli se nedotýká jiné lodi. Metoda `PoložLoď` se volá v případě, že náhodně umístěná loď prošla

kontrolou dotyku s jinou lodí. Metoda projde pole pozic u pokládané lodi a na stejné pozice zaznamené lod' v hracím poli.

Třída `Výstřel` reprezentuje jeden výstřel hráče a uchovává pozice x a y, na které bylo vystřeleno.

```
public class Vystrel
    {
        public int X { get; set; }
        public int Y { get; set; }

        public Vystrel(int x, int y)
        {
            X = x;
            Y = y;
        }
    }
```

Výpis 4: Třída Výstřel.

Třída `LoděEvaluator` slouží k ohodnocení neuronové sítě hraním hry Lodě proti náhodnému hráči.

Třída `LoděNeatExperiment` implementuje rozhraní `INeatExperiment` a uchovává detaily nastavení experimentu.

Třída `LoděCoevolutionNeatExperiment` implementuje rozhraní `INeatExperiment` a uchovává detaily nastavení experimentu.

Třída `CoevolutionEvaluator` slouží k ohodnocení neuronové sítě hraním hry Lodě.

Optimalizovaný hráč se ukládá do Xml souboru, který obsahuje neuronovou síť, která je složena z uzlů a propojení, které mají určitou váhu.

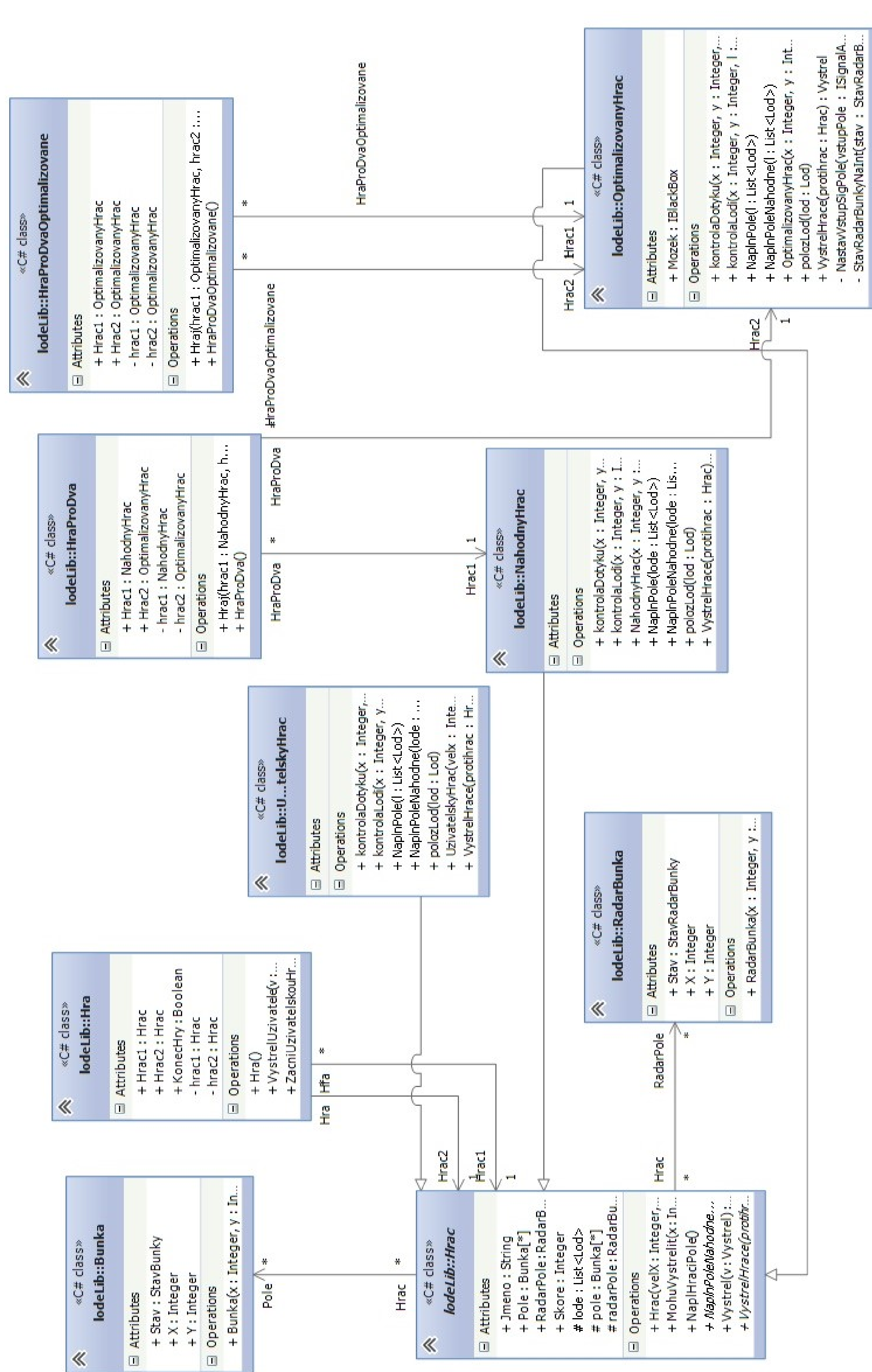

```
<Root>
  <ActivationFunctions>
    <Fn id="0" name="SteepenedSigmoid" prob="1" />
  </ActivationFunctions>
  <Networks>
    <Network id="110" birthGen="4" fitness="164">
      <Nodes>
        <Node type="bias" id="0" />
        <Node type="in" id="1" />
        <Node type="in" id="2" />
        <Node type="in" id="3" />
        <Node type="in" id="4" />
        <Node type="in" id="5" />
        <Node type="in" id="6" />
        <Node type="out" id="7" />
        <Node type="out" id="8" />
        <Node type="out" id="9" />
        <Node type="out" id="10" />
        <Node type="out" id="11" />
      </Nodes>
      <Connections>
        <Con id="217" src="0" tgt="117" wght="1,0893409326672554" />
        <Con id="319" src="1" tgt="119" wght="−1,077832626178860 7" />
```

```
      <Con id="330" src="1" tgt="130" wght="1,7623269138857722" />
      <Con id="355" src="1" tgt="155" wght="−1,7135390266776085" />
      <Con id="395" src="1" tgt="195" wght="−4,2115622898563743" />
      <Con id="429" src="2" tgt="129" wght="−3,8348331162706017" />
      <Con id="437" src="2" tgt="137" wght="3,3264686493203044" />
      <Con id="490" src="2" tgt="190" wght="4,1757720615714788" />
    </Connections>
  </Network>
 </Networks>
</Root>
```

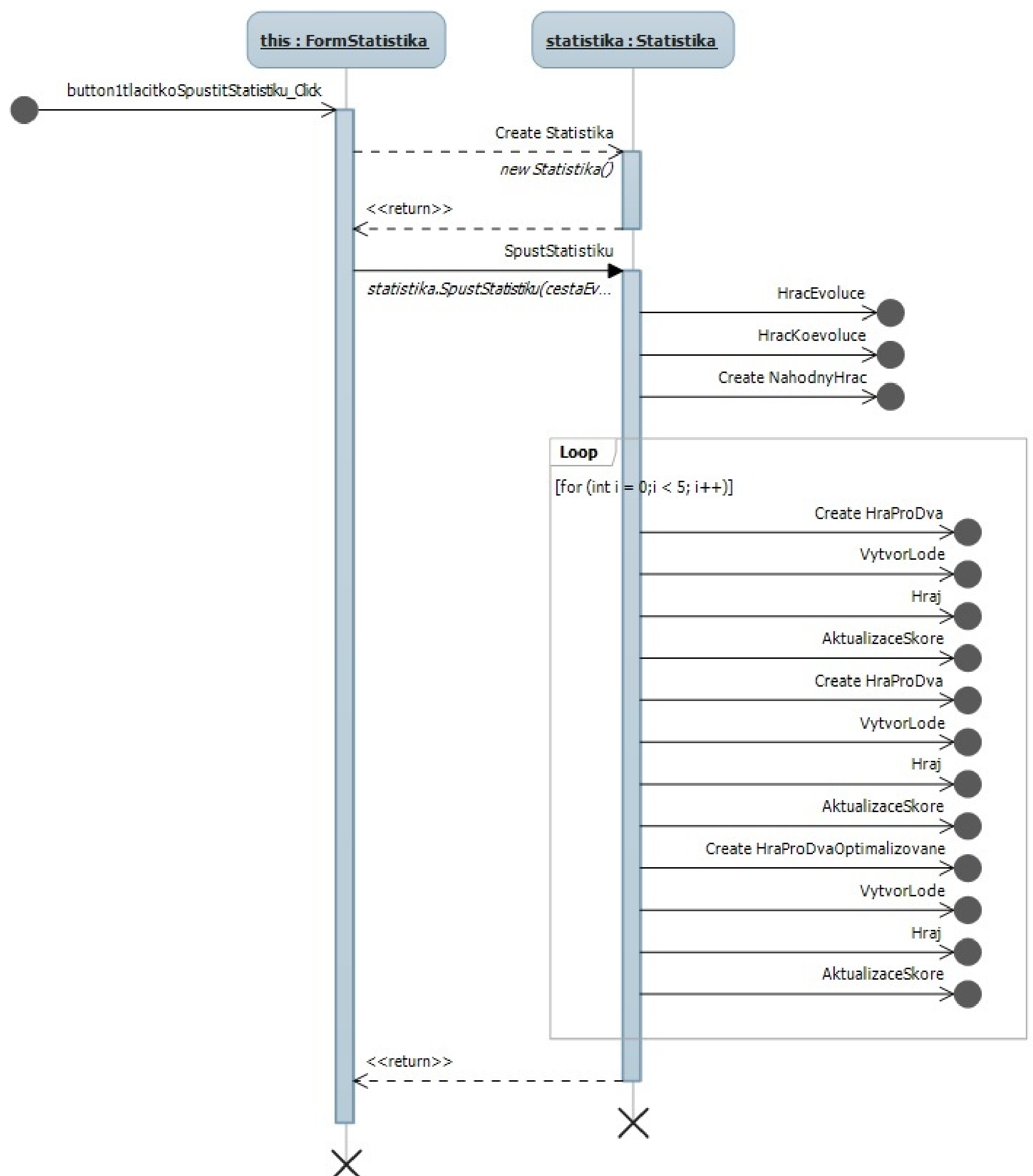
Výpis 5: Příklad uložené optimalizované neuronové sítě.

Na obrázku 11 je zobrazen diagram tříd, který zobrazuje vztahy mezi třídami pro hráče a jednotlivé hry.

Na obrázku 12 je zobrazen sekvenční diagram, který ukazuje průběh aplikace, když uživatel spustí statistiku. Vytvoří se instance třídy `Statistika` a zavolá se metoda, která spustí statistiku. V této metodě se zavolají metody, které načtou hráče optimalizovaného evolucí a koevolucí a vytvoří novou instanci náhodného hráče. V cyklu se vždy vytvoří pro každou hru nová instance hry pro dva hráče. Pro náhodného a optimalizovaného hráče a instance hry pro dva optimalizované hráče. Záznam každé hry se ukládá do slovníku a ten se po ukončení cyklu vrátí do GUI, kde se celý slovník projde, vypíše záznamy odehraných her a spočítá se počet výher každého hráče.



Obrázek 11: Třídní diagram.



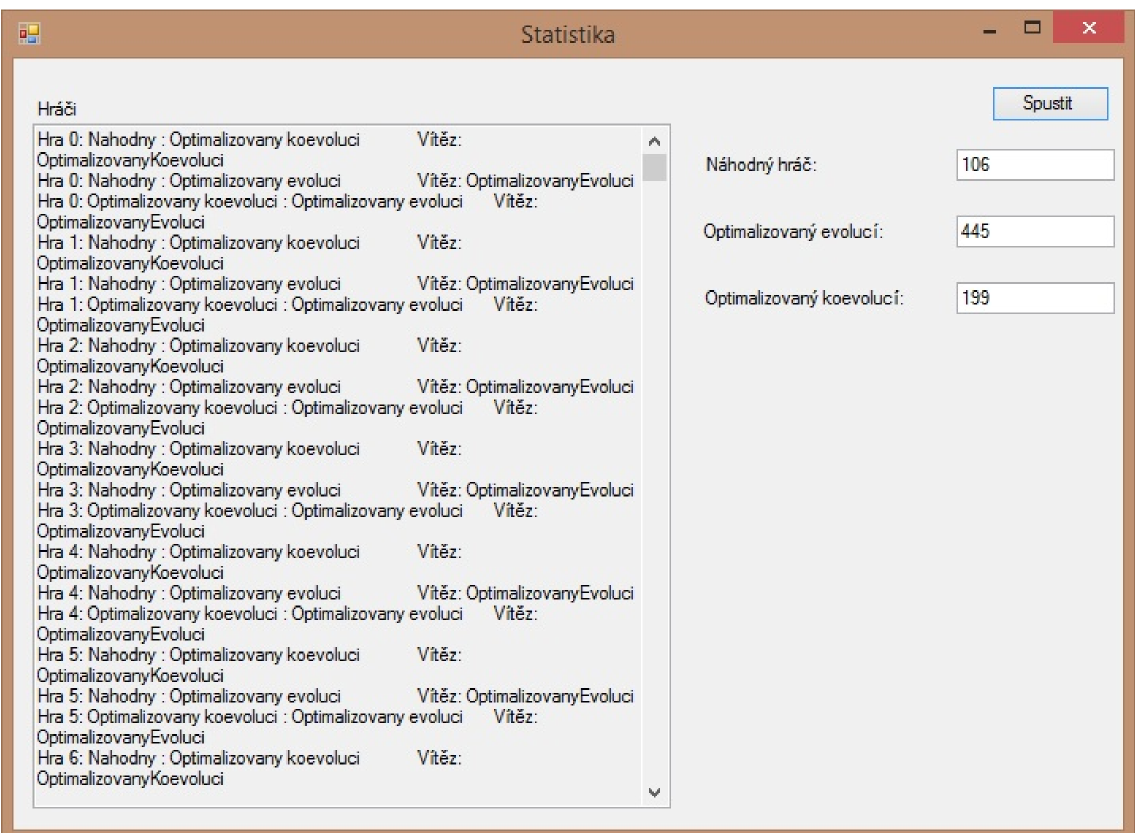
Obrázek 12: Sekvenční diagram.

10 Statistika

Statistika optimalizovaných hráčů evolucí, koevolucí a náhodného hráče. Hráči hrají proti sobě každý s každým daný počet her. Předpokládáme, že nejhůře bude hrát náhodný hráč, lépe bude hrát hráč optimalizovaný evolucí a nejlépe hráč optimalizovaný koevolucí.

Výsledky statistiky:

Hráč	Počet výher
Náhodný	106
Optimalizovaný evolucí	445
Optimalizovaný koevolucí	199



Obrázek 13: Výsledek statistiky

11 Závěr

Výsledek provedené statistiky 13 optimalizovaných hráčů splnil předpoklad, že nejhůře bude hrát náhodný hráč. Předpokládali jsme, že hráč optimalizovaný koevolucí bude hrát nejlépe, ale lépe hrál hráč optimalizovaný evolucí. U obou optimalizovaných hráčů byl zvolen stejný počet iterací a stejná populace. Ze zvoleného počtu iterací a výsledků statistiky předpokládám, že pro optimální optimalizaci protivníka koevolucí, aby hrál lépe než hráč optimalizovaný evolucí, musí být zvolen mnohonásobně větší počet iterací.

Výsledkem této práce je funkční aplikace logické hry Lodě, která využívá implementaci NEAT pro programovací jazyk C#.

Aplikace umožňuje uživateli optimalizovat protivníka pomocí evoluce nebo koevoluce a výsledného optimalizovaného hráče uložit do Xml souboru, jehož umístění si zvolí uživatel.

Aplikace umožňuje provézt statistiku náhodného hráče, optimalizovaného evolucí a optimalizovaného koevolucí. Uživatel zvolí umístění odkud se do aplikace ze souboru načtou optimalizovaní hráči. Výsledkem statistiky je skóre všech hráčů, kteří ve statistice hrají každý s každým.

Uživatel může hrát hru proti náhodnému hráči nebo optimalizovanému. Optimalizovaný hráč se načte na začátku hry ze souboru, který zvolí uživatel.

12 Reference

[1] Jiří Šíma, Roman Neruda, *Teoretické otázky neuronových sítí,* Praha: MATFYZPRESS, 1996.

[2] Mirza Cilimkovic, *Neural Networks and Back Propagation Algorithm,* Institute of Technology Blanchardstown.

[3] David Kriesel, *A Brief Introduction to Neural Networks,* Dostupné z WWW: www.dkriesel.com

[4] Kenneth O. Stanley, Risto Miikkulainen, *Evolving Neural Networks through Augmenting Topologies,* Evolutionary Computation, 2002, PMID: 12180173.

[5] Kenneth O. Stanley, *NEAT Software Doc File,* Evolutionary Computation, 2002.

[6] Kenneth O. Stanley, Nate Kohl, Rini Sherony, Risto Miikkulainen, *Neuroevolution of an automobile crash warning system,* ACM New York, 2005.

[7] Daniel Ashlock, *Evolutionary Computation for Modeling and Optimization*,Department of Computer Sciences, The University of Texas at Austin, 2001.

[8] J. R. Koza, *Genetic Programming: On the Programming of Computers by means of Natural Evolution*, MIT Press, Massachusetts, 1992.

[9] *Tutorial – Evolving Neural Networks with SharpNEAT 2 (Part 1) [online]*, 2010, Dostupné z WWW: <http://www.nashcoding.com/2010/07/17/tutorial-evolving-neural-networks-with-sharpneat-2-part-1/>.

[10] *Tutorial – Evolving Neural Networks with SharpNEAT 2 (Part 2) [online]*, 2010, Dostupné z WWW: http://www.nashcoding.com/2010/07/24/tutorial—evolving-neural-networks-with-sharpneat-2-part-2/

[11] *Implementace SharpNEAT [online]*, Dostupné z WWW: http://sourceforge.net/projects/sharpneat/

[12] *Implementace JNEAT [online]*, Dostupné z WWW: http://nn.cs.utexas.edu/soft-view.php?SoftID=5.

[13] *Implementace Python-NEAT [online]*, Dostupné z WWW: https://github.com/neat-python/neat-python.

[14] *Implementace RubyNeat [online]*, Dostupné z WWW: https://github.com/flajann2/rubyneat.

[15] *SharpNeat [online].*, 2004, last update 2013-04-29 [cit. 29-9-2013]. Dostupné z WWW: http://sharpneat.sourceforge.net.

13 Přílohy

DVD obsahující tento dokument ve formátu pdf a archiv se zdrojovými kódy a Xml soubory s předpřipravenými optimalizovanými hráči.