VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

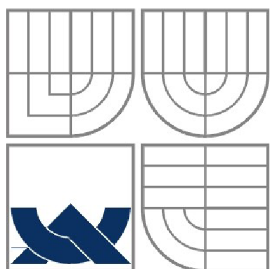
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

EVOLUCE COREWAR VÁLEČNÍKŮ POMOCÍ GENETICKÝCH ALGORITMŮ

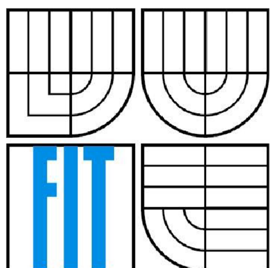
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN TŘÍSKA
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

EVOLUCE COREWAR VÁLEČNÍKŮ POMOCÍ GENETICKÝCH ALGORITMŮ

EVOLUTION OF COREWAR WARRIORS USING GENETIC ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Martin Tříska
AUTHOR

VEDOUCÍ PRÁCE — Ing. Jiří Zuzaňák
SUPERVISOR

BRNO 2010

Abstrakt

Evoluční algoritmy jsou progresivní a neustále se vyvíjející část informatiky. Jsou využívány zejména k řešení mnohodimenzionálních problémů s četnými lokálními maximami, které není možné řešit analyticky. Tato práce pojednává o možnosti jejich využití pro tvorbu programů v jazyce Redcode, které budou schopny bojovat dle pravidel hry Corewars. Navrhuje možnosti reprezentace programů jazyka Redcode pro účely evolučních algoritmů, řeší návrh platformy pro evaluaci fitness těchto jedinců a diskutuje možnosti jejich křížení a mutace. Součástí práce je rovněž aplikace schopná vývoje takovýchto programů.

Abstract

Evolutionary algorithms are a progressive and constantly evolving part of computer science. They are used mainly to solve the multidimensional problems with many local maxima, which are impossible to solve analytically. This thesis discusses how to use them for creating programs in Redcode language, which will be able to fight by the rules of game Corewars. Suggests possible representations of programs written in Redcode for evolutionary algorithms, discusses platform for evaluating their fitness and possible implementations of crossover and mutation. This thesis also contains application capable of development of such programs.

Klíčová slova

Corewars, Redcode, evoluční algoritmy, evoluční strategie, genetické algoritmy, softcomputing

Keywords

Corewars, Redcode, evolution algorithms, evolution strategies, genetic algorithms, softcomputing

Citace

MARTIN TŘÍSKA:Evoluce Corewar válečníků pomocí genetických algoritmů, bakalářská práce, Brno, FIT VUT v Brně, 2010

Evoluce Corewar válečníků pomocí genetických algoritmů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Jiřího Zuzaňáka
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Martin Tříska
19.05.2010

Poděkování

Tímto by jsem chtěl poděkovat Ing. Jiřímu Zuzaňákovi za jeho vedení a cenné rady při tvorbě této práce.

© Martin Tříska, 2010
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1

Úvod....3

1 Corewars....4

1.1 Redcode....4

1.2 MARS (Memory Array Redcode Simulator)....6

1.3 Používané strategie....6

1.4 Historie....7

1.5 Komunita....9

2 Evoluční algoritmy....11

2.1 Pojmy....11

2.2 Obecné schéma evolučního algoritmu....12

2.3 Typy evolučních algoritmů....12

2.4 Vytvoření počáteční populace....13

2.5 Operátor selekce....13

2.6 Operátor křížení – rekombinace....16

2.7 Operátor mutace....18

2.8 Robustnost evolučních algoritmů....19

3 Evoluční algoritmy jako nástroj pro vývoj CoreWars válečníků....21

3.1 Vytvoření počáteční generace....21

3.2 Fitness funkce válečníků....21

3.3 Reprezentace válečníků....22

3.4 Mutace válečníků....23

3.5 Křížení válečníků....24

3.6 Předchozí práce v tomto směru....24

4 Návrh implementace....27

4.1 Volba jazyka a prostředí....27

4.2 Vytvoření počáteční populace....28

4.3 Mutace a křížení válečníků....29

4.4 Fitness funkce a selekce....30

4.5 Automatická regulace řídících parametrů....31

5 Výsledky práce....33

5.1 Typický vývoj populace....33

5.2 Příklad vypěstovaného válečníka....36

6 Závěr....37

Literatura....38
Seznam příloh....39

Úvod

Tato bakalářská práce se zabývá vývojem Corewar válečníků, pomocí evolučních algoritmů.

Corewar válečník je program napsaný v jazyce Redcode, což je programovací jazýk, uzpůsobený na úspěch ve hře Corewars. Corewars je programátorská hra, v rámci které bojují 2 či více programů v paměti virtuálního stroje a pokoušejí se o jeho ovládnutí ukončením všech ostatních programů. Hra Corewars je podrobněji popsána v první kapitole této práce.

Evoluční algoritmy (EA) jsou v praxi využívány k řešení mnoha druhů problémů. Zejména se pak jedná o mnohodimenzionální problémy s množstvím lokálních maxim. EA hledají řešení pomoci analogií na biologické systémy a evoluční procesy, které v nich probíhají. EA jsou podrobněji popsány v druhé kapitole této práce.

V další kapitole je pojednáváno o možnosti aplikace EA na vývoj Corewar válečníků. Jsou tu například diskutovány možné způsoby reprezentace válečníků jako jedinců v EA, možnosti přístupu ke křížení a mutaci válečníků, či možnosti jejich ohodnocování pomocí fitness funkce.

Ve čtvrté kapitole této práce popisuje můj návrh a implementaci programu pro vývoj Corewar válečníků. Jsou v ní diskutovány všechny problémy, na které jsem při vývoji aplikace narazil.

V poslední kapitole této práce vyhodnocuji výsledky, kterých tato aplikace dosáhla a uvádím také příklad válečníka, který byl touto aplikací vytvořen. V teto kapitole rovněž popisuji možnost pro další pokračování vývoje v tomto směru.

1 Corewars

Corewars je programátorská hra, ve které dva nebo více programů bojuje v prostředí virtuálního stroje zvaného **MARS** (z anglického "Memory Array Redcode Simulator") a snaží se o jeho ovládnutí ukončením všech ostatních programů. Tyto programy jsou psány ve speciálním assembleru nazývaném **Redcode**.

1.1 Redcode

Redcode a MARS byly navržené jako jednoduchá abstraktní platforma určená k odstínění komplexnosti a složitosti dnešních reálných platforem. Redcode je speciální druh assembleru. Programy se skládají z instrukcí, s pevnou strukturou. Programy napsané v Redcode se nazývají Válečníci (Warriors).

Hlavní vlastnosti redcode:

- Velmi malý počet instrukcí 10 (ve verzi ICWS-88) a 18 (ve verzi ICWS-94)
- Redcode se skládá pouze z instrukcí – není možné uchovávat samostatná data. Ty se uchovávají v podobě argumentů instrukcí.
- Každá instrukce má vždy právě 2 numerická pole (argumenty). Ke každému poli je také asociován adresní mód.
- Ve verzi ICWS-94 obsahuje instrukce také modifikátor, který určuje nad jakou částí dat pracuje instrukce – A pole, B pole nebo obě pole (příp. obě v převráceném pořadí).
- Každá instrukce má stejnou velikost a vykonání každé instrukce trvá stejný čas.
- Všechny adresy jsou počítány jako relativní k vykonávané instrukci (tj. adresa 0 ukazuje na aktuálně vykonávanou instrukci). V abstrakci MARSu je jeho paměť považována za cyklickou – tj. všechny adresy jsou počítány modulo velikost paměti. Důsledkem tohoto je jakákoli adresa validní. Také proto nemá smysl uvažovat o absolutních adresách – paměť nemá začátek ani konec – tj. není odkud absolutní adresování počítat.
- Všechna čísla jsou považována za nezáporná a menší jako velikost paměti MARSu. Proto existuje jednoznačný 1:1 vztah mezi číslem a instrukcí.

Struktura instrukce v Redcode:

1. operační kód – určuje sémantiku instrukce
2. modifikátor – určuje nad kterými poli instrukce pracuje
3. adresní mód pro A pole – určuje typ adresování použitý pro A pole
4. A pole
5. adresní mód pro B pole
6. B pole

Tato struktura je pevná pro každou instrukci a musí být dodržena i pokud jsou některé části instrukce v daném kontextu zbytečné.

Příklady instrukcí:

MOV.AB $ 0, $ 1

MOV - instrukce bude kopírovat instrukci

.AB - přesune se instrukce adresována polem A na adresu danou polem B

$ - relativní adresování (číslo bude znamenat posun oproti aktuálně vykonávané instrukci)

0 - posun je nula => pole A ukazuje na vykonávanou instrukci

$1 - pole B ukazuje na následující instrukci

Výsledek: instrukce zkopíruje sama sebe o 1 adresu dále v paměti.

ADD.BA $ -2, # 5

ADD - instrukce přičítat

.BA - pole B bude „source" a pole A „destination"

$ - relativní adresování (číslo bude znamenat posun oproti aktuálně vykonávané instrukci)

-2 - A pole ukazuje na instrukci o 2 před ní

- absolutní adresování – v tomto kontextu znamená, že se bude přičítat přímo číslo

5 - pole B obsahuje číslo 5

Výsledek: instrukce přičte k poli A instrukce, na o 2 nižší adrese číslo 5.

1.2 MARS (Memory Array Redcode Simulator)

MARS je abstraktní virtuální stroj ve kterém Corewar bitvy probíhají. Před začátkem samotného souboje jsou do paměti MARSu nahrány programy, které se mají utkat v bitvě. Poté se do fronty procesů zapíší adresy startovních instrukcí programů. Poté se cyklicky provádějí instrukce vždy jedna od každého běžícího programu. Programy mají možnost vytvářet nové procesy. V tomto případe se však vytvoří vnitřní fronta procesů v rámci programu a jednotlivé procesy programu se budou rovnoměrně dělit o přidělený výpočtový čas, neboť v jednom „kole" se vždy vykoná pouze jedna instrukce celého programu a ne každého procesu.

Pro ilustraci: Mějme 2 bojující programy. První má tři aktivní procesy, druhý má dva aktivní procesy. MARS bude vykonávat instrukce procesů v tomto pořadí:

Uvedeno ve tvaru [program / proces]. Jedno okénko = jedna instrukce daného procesu. Odstíny modré znázorňují procesy prvního programu, odstíny zelené procesy druhého programu.

Program je považován za aktivní pokud má aktivní aspoň jeden proces. Proces končí svoji činnost pokusem o vykonání instrukce DAT.

Od verze ICWS '94 existuje varianta, ve které má každý program jisté místo v paměti s výhradním přístupem, kam si může spolehlivě ukládat potřebná data. Toto místo se nazývá P-space (od Private space).

1.3 Používané strategie

Bylo popsáno několik základních strategií, na kterých se zakládá většina vytvářených válečníků. . Zde vyjmenujeme pouze nejznámější strategie. Dnešní válečníci jsou téměř vždy kombinací více těchto strategií.

- **Imp**

  Je nejjednodušší strategie, při které se pouze jedna instrukce neustále rozkopírovává právě o jednu instrukci dále. Při dalším spuštění tohoto procesu bude Instruction Pointer posunutý právě na tuto nakopírovanou instrukci, a tak se imp zkopíruje zase dále. Tato strategie je absolutně nepoužitelná pro útok proti jiným válečníkům. Dnes je používána zejména jako

část jiného válečníka, přičemž se využívá faktu, že „Imp-y“ je možné produkovat velmi rychle a ve velkých množstvích a toho, že každý z těchto impů je schopen sám přežít i když zbytek válečníka je již zabit a tak zabezpečit alespoň remízu.

- **Bomber / stone**

  Velmi jednoduchá strategie založená na rozsévání „bomb“ po paměti. Pokud se některý válečník pokusí o vykonání instrukce, kam byla umístěna bomba, je typicky ukončen. Nejčastěji je k tomu využíváno instrukce DAT avšak jsou i jiné možnosti či dokonce používání více-instrukčních bomb. Prvním zástupcem této kategorie se stal *Dwarf* napsán *A. K. Dewdney.*

- **Replicator / paper**

  Program založený na této strategii se rozkopírovává po paměti a paralelně spouští všechny svoje kopie. Replikátoři zřídka zabijí svého protivníka, avšak také je těžké zabít je, což často ústí v remízu. Prvního replikátora uveřejnil *Chip Wendell* pod názvem *Mice*.

- **Scanner / scissor**

  Takto se nazývá strategie válečníků navržená zejména pro zabíjení replikátorů. Tito válečníci nejprve zaplaví paměť instrukcemi SPL 0, co způsobí, že každý válečník který se pokusí o vykonání této instrukce začne vytvářet další procesy, které nedělají nic jiného, pouze vytvářejí další dělící se procesy. Tím se neustále zpomaluje vykonávání jakéhokoli užitečného kódu. Až je protivník natolik pomalý, že není schopen vykonat nic užitečné, paměť se celá přepíše instrukcemi DAT co způsobí ukončení všech procesů protivníka.

1.4 Historie

V této podkapitole je uvedená chronologicky uspořádaná historie CoreWars ve vztahu k evolučním algoritmům a experimentům s těmito algoritmy.

1984

- *D. G. Jones* a *A. K. Dewdney* představují Redcode a MARS v publikaci *Core War Guidelines*.
- *A. K. Dewdney* představuje Core Wars v jeho počítačové sekci časopisu *Scientific American*.
- *Kevin A. Bjorke* vydává jeho implementaci MARSu v jazyce C.
- *Berry Kercheval* portuje MARS pro Usenet.

1985

- *A. K. Dewdney* uveřejňuje v *Scientific American* svůj druhý článek o Core Wars.
- *Robert Martin* publikuje jeho implementaci Core Wars pro Macintosh

- Okolo Marka Clarksona se formuje mezinárodní organizace *International Core Wars Society* (ICWS).

1986

- Je vytvořen ICWS'86 Redcode Standard.
- Je uspořádán první mezinárodní turnaj v Core Wars. První místo získává warrior *Chipa Wendella* s názvem *Mice*.

1987

- *A. K. Dewdney* uveřejňuje svůj třetí článek o Core Wars v *Scientific American.*
- Začíná vycházet *Core War Newsletter*. Vydává ho *William R. Buckley.*
- Vzniká japonská Core Wars společnost – *ICWS Japan.*
- Druhé kolo mezinárodního Core Wars turnaje. Vyhrává válečník *Ferret* od *Roberta R. Reeda.*

1988

- ICWS'88 Redcode Standard
- Třetí kolo mezinárodního Core Wars turnaje. Vyhrává válečník *Cowboy* od *Eugena P. Lilitka.*

1990

- Autoři Steen Rasmussen, Carsten Knudsen, Pasmus Feldberg a Morten Hindsholm vydávají knihu: *The coreworld: emergence and evolution of cooperative structures in a computational chemistry.*

1992

- John Perry uveřejňuje článek o evoluci v Core Wars pod názvem *The Evolution of Predation*

1993

- *Steven Morrell* napsal rozsáhlý článek s názvem *My First Corewar Book.*

1996

- *Steven Morrell* napsal článek *Mathematical Models for Step Sizes.*

1997

- *Franz* sestavil program pro evoluci Core War válečníků. Program se jmenoval *SYS4* a byl napsán v Perlu.
- *Jason Boer* napsal také program evolující válečníky s názvem *GA_WAR* – v jazyce C.

1998

- *Terry Newton* zveřejnil program *Redmaker* – další program na evoluci Core War válečníků.

2000

- Další programy na evoluci válečníků:
    - *Martin Ankerl – Y.A.C.E*
    - *Dave Hillis - Redrace*

2003

- Opět další programy na evoluci válečníků:
    - *Will Varfar - Species*
    - *Terry Newton - Fizzle*

1.5 Komunita

Okolo Core Wars vznikla rozsáhlá a živá komunita, v níž se odehrávají pravidelně turnaje, upravují se pravidla Core Wars i samotná abstrakce MARSu. V roku 1985 vznikla mezinárodní organizace ICWS která tyto změny organizuje a koordinuje. Postupně se vytvořilo také více soutěžních kategorií v závislosti na velikosti paměti MARSu, použité verze Redcode, počtu povolených procesů či možnosti použití P-space. Vznikly také kategorie pro začátečníky v Core Wars a kategorie pro souboje vždy několika (zpravidla více jako 2) válečníků najednou.

Způsoby hodnocení válečníků

Neexistuje žádný oficiální systém pro hodnocení válečníků, avšak časem se stali uznávané zejména 2 typy testů:

Wilkies Benchmark

Tento test postaví testovaného válečníka proti sadě 12-ti referenčních válečníků. S každým z nich odehraje 100 soubojů. přičemž za výhru získává válečník 3 body, za remízu 1 bod. Výsledné skóre se podělí číslem 12, čím získáme hodnocení v rozmezí 0-300. Většina válečníků má však problém dosáhnout hodnocení 100 bodů.

Postavení v King of the Hill

Nejznámější server provozující turnaje v CoreWars. Autoři zasílají své válečníky na server, kde jsou poté v patřičném kole postaveni každý proti každému. Z těchto soubojů je opět spočítáno takzvané Hill score – opět 3 body za výhru a 1 bod za remízu. Pokud některý nový válečník dosáhne vyššího hodnocení, jako nejnižší v tabulce (Hill), tak jej nahradí a vytlačí staršího válečníka z tabulky.

Servery provozující turnaje

- KOTH (King of the Hill)

  Domácí server ICWS se standardními kategoriemi.

  http://www.koth.org/
- Corewars hostované na sourceforce.net.

  http://corewars.sourceforge.net/
- Server provozující turnaj v sedmi kategoriích včetně kategorie pro začátečníky.

  http://sal.math.ualberta.ca/

2 Evoluční algoritmy

Evoluční algoritmy jsou společným vyjádřením pro třídu moderních matematických postupů, které využívají modely evolučních procesů v přírodě.[1] Většinou se jedná o netriviální mnohodimenzionální problémy, které není možné řešit analyticky. Existuje více přístupů známých pod názvy Genetické algoritmy či Evoluční strategie.

Všechny tyto postupy jsou však založeny na podobných principech a mají množství společných rysů. Obvykle se pracuje s množinou možných řešení, ze kterých se vybere počáteční množina řešení. Z těchto řešení jsou vybrány ta nejlepší, které jsou pak vzájemně kříženy a mutovány, čím vznikají nová řešení, která zabírají místo těch méně úspěšných.

2.1 Pojmy

Pro další využití si musíme nyní vysvětlit některé pojmy, běžně používané v oblasti evolučních algoritmů.

Jedinec, generace

Jednotlivá řešení v kontextu evolučních algoritmů nazýváme *jedinci*. Množinu aktuálně zpracovávaných jedinců potom nazýváme generací. Jedince někdy nazýváme také *fenotypem*. Zakódovaní jedince pro potřeby evolučního algoritmu potom nazýváme jeho *genotypem* či *chromozómem.* Genotyp dělíme dále na *geny* což jsou informace o jednotlivých vlastnostech jedince. Různé možné hodnoty genu pak nazýváme *alelami*.

Fitness funkce

Pro funkčnost evolučních algoritmů potřebujeme funkci, která nám ohodnotí konkrétní řešení, aby jej bylo možné porovnat s ostatními jedinci z generace a tak vybrat ty nejlepší jedince. Tuto funkci běžně nazýváme *fitness funkcí.* Její implementace se liší podle konkrétního řešeného problému.

Operátor křížení, operátor mutace

Evoluční algoritmy jsou založené na postupné mutaci a křížení již známých řešení. Proto potřebujeme metody/operátory, které budou schopné:

- z dvou či více jedinců vzájemnou rekombinací vytvořit nového jedince - nazýváme *operátor křížení*

- z jednoho jedince vytvořit potomka mírnou úpravou existujícího jedince – nazýváme *operátor mutace*

Obnova populace. operátor selekce

Součástí evolučního algoritmu je operace, při níž z úspěšnějších jedinců staré generace vytvoříme generaci novou. Tento proces se nazývá *obnova populace*. Prostředek vybírající jedince podílející se na obnově populace se nazývá *operátor selekce.*

Náhodný generický drift

Nazýváme takto náhodné události, při nichž občas zahynou jedinci s vysokou fitness, nebo jiní jedinci s nízkou fitness jsou zahrnuti do reprodukčního cyklu. Tato vlastnost je mohutným nástrojem boje proti uvíznutí v lokálních maximech. Nejpatrnější je genetický drift v malých populacích.

2.2 Obecné schéma evolučního algoritmu

Při navrhování evolučních algoritmů se inspirujeme evolučním procesem probíhajícím v přírodě. Snažíme se najít analogii mezi částmi řešeného problému a přírodními procesy. Nejprve se vytvoří počáteční generace a každý jedinec je ohodnocen. Pomocí operátoru selekce se poté vybere množina jedinců, kteří se budou podílet na vytvoření nové generace. Pomocí rekombinace (křížení), mutace a reprodukce (přímé zkopírování) je poté vytvořena nová generace, která je opět ohodnocena a celý cyklus se opakuje. Ukončovací podmínky mohou být dány minimální kvalitou výsledného řešení nebo předem určeným časem běhu.

Tento postup je možné shrnout do osnovy:

1. vytvoření počáteční populace
2. ohodnocení generace
3. selekce úspěšných jedinců
4. rekombinace a mutace, obnova populace
5. pokud řešení není dostačující, návrat k bodu 2.

2.3 Typy evolučních algoritmů

Jak již bylo vzpomenuté, existuje více přístupů k evolučním algoritmům. Hlavními směry jsou *Genetické algoritmy* a *Evoluční strategie*. Tyto směry se liší hlavně ve způsobu zakódování jedinců, křížení a mutaci jedinců a ve způsobech výběru jedinců, kteří se budou podílet na reprodukci. Tyto rozdíly budeme diskutovat v odstavcích věnovaných těmto praktikám.

Reprezentace jedinců

Zde pouze vzpomeneme, jaké typy reprezentace genomu se používají v jednotlivých typech evolučních algoritmů.

U *genetických algoritmů* jde o binární zakódování. Fenotyp je zakódován pomocí binárního řetězce, nad kterým jsou potom prováděny všechny operace jako křížení či mutace. Genotyp jednoho jedince pak může vypadat takto:

0	1	1	0	0	1	0	0	0	1

Naproti tomu u *evolučních strategií* reprezentujeme genotyp jedince n-ticí celých nebo reálných čísel. Genotyp jedince potom může vypadat takto:

2	4	-5	2,3	-1,98	-17,2	0

2.4 Vytvoření počáteční populace

Způsob vytvoření počáteční populace závisí na řešeném problému. Nejčastěji se využívá vygenerování náhodných řešení (jedinců) s minimálními hodnotami fitness. Pokud je to však možné, je pro urychlení konvergence řešení vhodné inicializovat počáteční generaci nějakým sofistikovanějším způsobem.

2.5 Operátor selekce

Existuje několik druhů operátorů selekce. Jejich společnou vlastností je, že se snaží upřednostňovat ve výběru úspěšnější jedince na úkor těch méně úspěšných. U genetických algoritmů to jsou proporcionální selekce, lineární uspořádání, exponenciální uspořádání a turnajová selekce. U evolučních strategií obvykle vybíráme jedince uniformně.

Proporcionální selekce

Proporcionální selekce byla vůbec prvním používaným operátorem selekce. Její myšlenkou je dát jedincovi šanci podílet se na vytváření nové generace úměrnou jeho fitness.

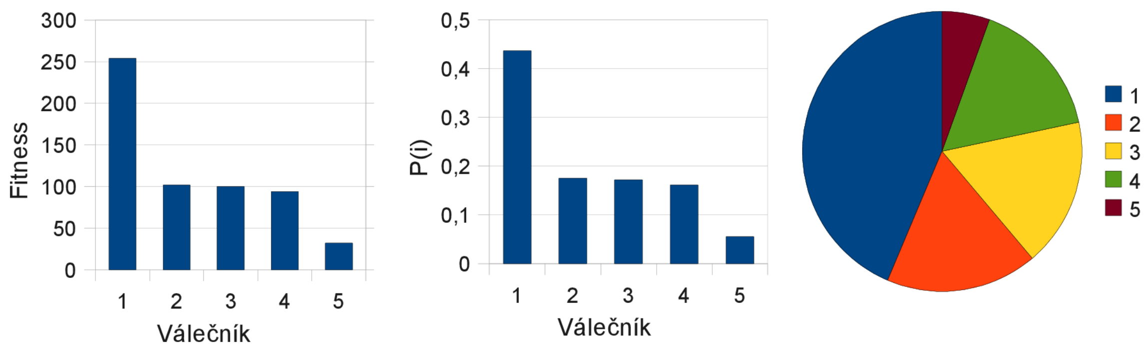
Pravděpodobnost výběru i-tého jedince potom můžeme zapsat jako:

$$p_i = \frac{f_i}{\sum_{j=0}^{N} f_j}$$

(1)

Tomuto typu výběru říkáme i metoda rulety. Tento způsob má však svá úskalí – pokud se v populaci vyskytne (ve srovnání s ostatními) výrazně silný jedinec, tak je populace postupně nahrazena tímto jedincem a jeho mutacemi. Pro omezení tohoto jevu existují různé techniky upravující fitness vhodným způsobem, aby zmenšili rozdíl mezi nejlepším a nejhorším jedincem.

Následující grafy znázorňují příklad proporcionální selekce. První graf nám zadává ukázkovou populaci 5-ti válečníků, druhý a třetí graf znázorňují pravděpodobnost výběru daného jedince. Druhý graf je uveden pouze pro názornost, že první a druhý graf vypadá stejně a jediné, co se změnilo je význam a stupnice na ose *y*. (Hodnoty v grafu jsou poděleny sumou hodnot fitness celé populace *viď vzorec(1)*) Na třetí graf se můžeme dívat také jako na „ruletu“. Při výběru válečníka do procesu reprodukce náhodně vybereme místo na obvodu kruhu. Kterému válečníkovi toto místo připadá, ten je vybrán k reprodukci.



Lineární uspořádání – ranking

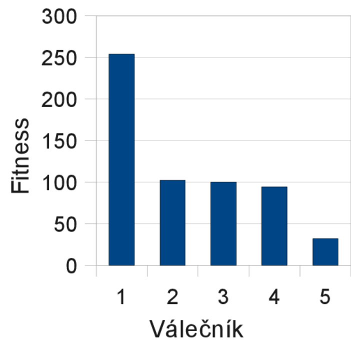
Zbavuje nás problému příliš velkého rozdílu mezi fitness jednotlivých jedinců, avšak na druhé straně může být příliš nespravedlivý. Tento selektor uspořádá všechny jedince generace dle jejich fitness a pak buď pouze vezme nejlepších *N*, nebo opět vybírá jedince ***i*** s pravděpodobností:

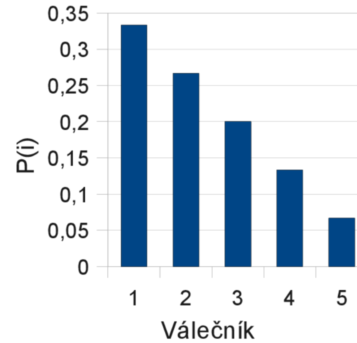
$$p_i = \frac{2-s}{N} + \frac{2i(s-1)}{N(N-1)}$$

(2)

kde ***s*** je selekční tlak a ***N*** je počet jedinců v generaci.

V následujících grafech máme opět stejnou populaci jedinců, jako v minulém příkladě a jejich pravděpodobnosti vybrání do cyklu reprodukce. Druhý graf vznikl aplikací vzorce (2) za použití selekčního tlaku s=2.





Exponenciální uspořádání

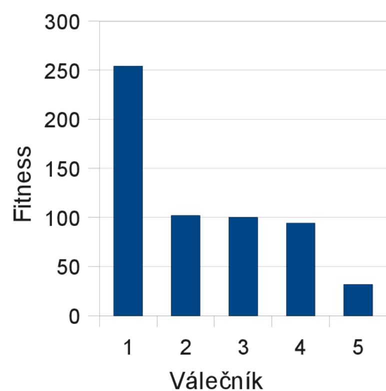
Liší se od předcházejícího pouze tím, že pravděpodobnost výběru není uspořádána v setříděné populaci lineárně ale exponenciálně. Tím jsou častěji vybíráni nejlepší na úkor variability rodičů.

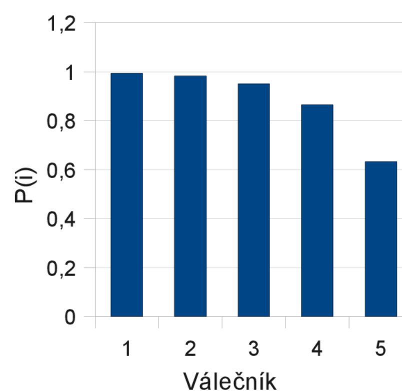
Můžeme jej opět zapsat jako:

$$p_i = \frac{1 - e^{-i}}{c}$$

(3)

kde ***c*** je normalizační konstanta, pomocí které můžeme upravovat selekční tlak na populaci. Následující grafy opět znázorňují stejnou populaci jedinců a jejich pravděpodobnost výběru, jakou by jim přisoudilo exponenciální uspořádání dle vzorce(3) s konstantou *c=1*.





Tento druh selekce je v teorii evolučních algoritmů označován jako jeden z nejlepších.

Turnajová selekce

Při tomto druhu selekce vybereme ***k*** jedinců, z nichž postupuje do další generace nejlepší. Tento postup opakujeme *N* krát, kde *N* je počet jedinců v nové generaci. Turnajová selekce dosahuje podobné výsledky jako exponenciální uspořádání. Navíc postrádá nutnost uspořádání generace podle fitness a proto je velmi často využívána.

Selekce v evolučních strategiích

V algoritmech založených na evolučních strategiích obvykle vybíráme jedince uniformě s tím, že nová populace je připojena k stávající a pak seřazena dle fitness. Takto pokud je potomek lepší, vytlačí z populace jeho rodiče. Tento přístup k selekci označujeme jako (μ+λ) a nazýváme jej také elitizmem. Existuje i přístup, ve kterém může rodiče vytlačit i slabší potomek. Označujme to jako (μ,λ) přístup. Tato technika pomáhá opustit lokální maxima.

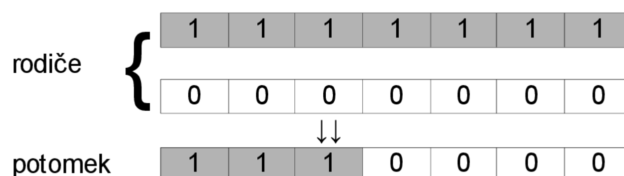
2.6 Operátor křížení – rekombinace

Křížení neboli rekombinace je důležitou součástí evolučních algoritmů. Jeho největším přínosem je výměna informací mezi jedinci. Tento operátor je klíčový zejména pro teorii *stavebních bloků.* Dle této teorie jsou genetické algoritmy samy schopné rozpoznat kvalitní stavební bloky – úseky genomu, které samostatně plní určité funkce. Tyto bloky potom pře-uspořádávají do funkčních jedinců.

Existuje celá řada operátorů křížení. Zde vzpomeneme pouze ty nejčastěji využívané. U genetických algoritmů jsou to *jednobodové křížení, vícebodové křížení* a *uniformní křížení.* U evolučních strategií to budou zejména *křížení průměrem* a *diskrétní křížení*.

Jednobodové křížení

Aplikuje se zejména u genetických algoritmů a tedy probíhá nad binárními vektory. Náhodně je vybráno číslo v intervalu <0;N> kde *N* je délka genomu. Následně jsou od tohoto bodu dále geny jedinců vyměněny. Jednobodové křížení dvou jedinců, z kterých genom jednoho pozůstává pouze z jedniček a genom druhého pouze z nul by mohlo vypadat následovně:



Vícebodové křížení

Probíhá stejně jako jednobodové křížení pouze s tím rozdílem, že u vícebodového křížení je vybráno více bodů, kde se vymění rodičovský jedinec, od kterého bude následník přebírat část genomu. Výsledek vícebodového křížení potom může vypadat následovně:

rodiče	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0
						↓↓			
potomek	1	1	1	0	0	1	1	0	0

Uniformní křížení

Tento operátor prochází genom dvou rodičovských jedinců a s pravděpodobností p_u geny vymění. Tato metoda křížení prohledává prostor možných řešení mnohem intenzivněji než předchozí metody a může do řešení přinášet žádanou různorodost. Na druhé straně je často zavrhována pro její přílišné rozbíjení genomu a stavebních bloků.

Výsledek uniformního křížení může vypadat následovně:

rodiče	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0
					↓↓				
potomek	1	1	0	1	0	1	1	0	1

Křížení průměrem

Tento způsob křížení se využívá v evolučních strategiích, kde reprezentaci genomu představují celá nebo reálná čísla. Křížení průměrem potom pouze průměruje hodnoty na příslušných pozicích rodičů a přiřazuje tyto hodnoty potomkovi. Tento způsob křížení však postupně snižuje variabilitu jedinců a časem vede k jednotvárnosti genetického materiálu. Proto není možné algoritmus postavit pouze na této metodě křížení, bez přítomnosti mutací.

Výsledek křížení průměrem může vypadat takto:

rodiče	16	-4	18	22	-5
	19	0	7	18	3
			↓↓		
potomek	18	-2	13	20	-1

Diskrétní křížení

Je druhým způsobem křížení využívaným v evolučních strategiích. Potomek při něm přebírá gen vždy od jednoho nebo druhého rodiče. Způsoby výběru jedince, od kterého bude gen přebrán jsou opět různé – můžeme aplikovat jednobodové či vícebodové křížení podobně jako u genetických algoritmů, pouze tentokrát již kopírujeme čísla a ne pouze části binárního vektoru. Nebo můžeme opět dospět k uniformnímu křížení při kterém pravděpodobnosti převzetí genu od některého z rodičů určuje pravděpodobnost. Výsledný jedinec potom může vypadat následovně:

rodiče	16	-4	18	22	-5
	19	0	7	18	3
			↓↓		
potomek	16	0	18	18	3

2.7 Operátor mutace

Mutace je zdrojem nových informací pro evoluční algoritmy a přispívá k variabilitě populace.

Opět existuje více možností implementace tohoto operátoru a opět je implementace závislá na zvoleném způsobu reprezentace jedinců. U genetických algoritmů je operátor mutace nejčastěji implementován prostou bitovou inverzí. Naproti tomu u evolučních strategií se většinou postupuje cestou Gaussovského rozložení.

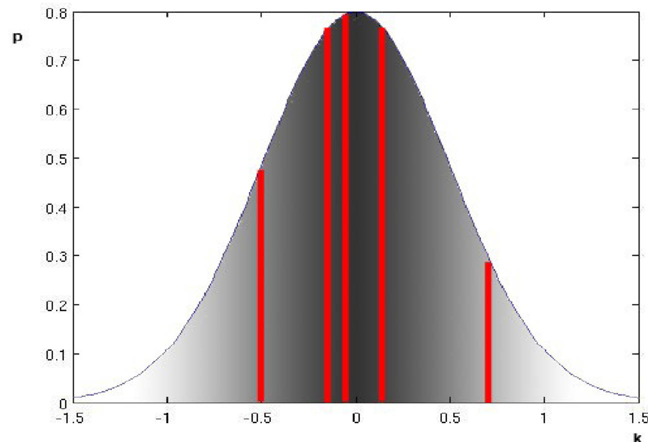
Mutace bitovou inverzí

Aplikuje se u genetických algoritmů, kde se při mutaci každý bit genomu invertuje s pravděpodobností p_m. Tato pravděpodobnost se obvykle nastavuje mezi 0,0005 a 0,01. Příliš velké p_m způsobuje nestabilitu evolučního procesu. Příliš malé potom zase není schopné přinést do populace žádanou variabilitu, konvergence se zpomaluje a hrozí uvíznutí v lokálních maximech.

Mutace Gaussovským rozložením

Je užívána v evolučních strategiích, kde je genom reprezentován celými či reálnými čísly. V tomto případě mutace mění hodnotu příslušného čísla (genu) o hodnotu danou Gausovským pravděpodobnostním rozložením se střední hodnotou v nule a rozptylem σ. Tím je zabezpečeno, že nejčastěji je hodnota genu pozměněna pouze o málo, větší změny nastávají s malou pravděpodobností.

V následujícím grafu je znázorněno Gaussovské rozložení pravděpodobnosti se střední hodnotou 0 a odchylkou 0,5. Zakreslené čáry v grafu znázorňují hodnoty vektoru mutace, vygenerovaného tímto rozložením.



Mějme vektor reálných čísel, představující genotyp jedince:

1	0,8	4,2	-2,1	-3,2

Dále pomocí zmíněného rozložení vygenerujme vektor mutace:

0,1	-0,5	-0,15	0,7	-0,05

Sečtením vektoru jedince a mutačního vektoru získáme zmutovaného jedince:

1,1	0,3	4,05	-1,4	-3,25

2.8 Robustnost evolučních algoritmů

EA(Evoluční algoritmy) nacházejí uplatnění zejména u složitých nelineárních mnoho dimenzionálních problémů, které nejsme schopni řešit analyticky či jinak. Jsou s výhodou nasazovány v úlohách, které by bylo možné definovat následovně:

- Prostor řešení je příliš rozsáhlý a chybí expertní znalost, která by umožnila zúžit prostor slibných řešení.
- Nelze provést matematickou analýzu problému.
- Tradiční metody selhávají.
- Jde o úlohy s mnohočetnými extrémy, kriterii a omezujícími podmínkami. [1]

EA takto nacházejí využití v mnoha odvětvích konstrukce, strojírenství, návrhu obvodů, či tvorbě ekonomických či sociologických modelů. Musíme však upozornit i na jejich nevýhody a úskalí:

- Kvalitu řešení lze ohodnotit pouze relativně. Nelze otestovat, zda se jedná o globální optimum
- Pro mnohé úlohy je typická velká časová náročnost.
- Pro velmi rozsáhlé úlohy poskytuje řešení příliš vzdálená od optima.
- Ukončení optimalizace je explicitní na základě časového limitu nebo stagnace kriteriální funkce. [1]

3 Evoluční algoritmy jako nástroj pro vývoj CoreWars válečníků

V oblasti vývoje CoreWars válečníků pomocí evolučních algoritmů bylo již provedeno několik pokusů realizovaných několika autory. Všechny však dosahují pouze částečných úspěchů a žádný z válečníků vytvořený čistě pomocí evolučních algoritmů zatím nebyl schopen se úspěšně postavit válečníkům napsaným lidmi a tak vstoupit do tabulky King of the Hill.

V této kapitole se pokusíme probrat problematiku aplikace evolučních algoritmů na vývoj válečníků, a problémy s tím spojené. Vzpomeneme problematiku vytváření počáteční generace, hodnocení válečníků – tvorby fitness funkce a také problémy spojené s křížením a mutací válečníků. Také stručně vyjmenujeme některé předchůdce v této oblasti a jimi dosažené úspěchy.

3.1 Vytvoření počáteční generace

Díky modulární aritmetice použité v RedCode jsou všechny programy validní a proto je možné vygenerovat čistě náhodnou počáteční generaci (např. v ga_war.c [4]). Dále je možné začít od několika manuálně vložených instrukcí – např. v [5] se populace inicializuje sérií 4 instrukcí SPL (split) čím se autor snaží o zvýšení robustnosti výsledných válečníků, kteří budou mít většinou několik paralelních vláken.

Jako poslední existuje možnost inicializovat populaci plně funkčním válečníkem a evoluční algoritmy použít pouze k jeho optimalizaci.

3.2 Fitness funkce válečníků

Fitness funkci pro válečníky obvykle určuje jejich skóre, které získají v soubojích s jinými válečníky. Zde existují dvě možnosti:

- vnitřní hodnocení – oproti jiným aktuálně vyvíjeným válečníkům
- vnější hodnocení – oproti sadě referenčních válečníků

Dále musíme rozhodnout počet soubojů odehraných v rámci jednoho určování fitness. Ideální by bylo odehrát turnajový systém – každý proti každému. Při větších populacích však takovýto postup příliš zpomaluje evoluci. Ve všeobecnosti můžeme říci, že s rostoucím počtem soubojů roste přesnost a spravedlnost fitness funkce, na druhé straně se její vyhodnocení zpomaluje.

3.3 Reprezentace válečníků

Další problém, který musíme řešit je způsob reprezentace válečníků. Práce přímo nad textovou reprezentací se ukázala jako nepraktická. V úvahu potom připadají například tyto možnosti reprezentace:

- n-tice celých čísel
- přímý pohled jako na binární data
- posloupnost gramatických pravidel

Nejjednodušším přístupem je zřejmě reprezentace každé instrukce válečníka jako 6-tice čísel z nichž:

1. vyjadřuje instrukci, je reprezentováno číslem v intervalu <0;17>
2. vyjadřuje modifikátor, je reprezentováno číslem v intervalu <0;7>
3. vyjadřuje adresní mód pro A pole, je reprezentováno číslem v intervalu <0;8>
4. vyjadřuje obsah A pole, je reprezentováno číslem v intervalu <0;CORE_SIZE[*]>
5. vyjadřuje adresní mód pro B pole, je reprezentováno číslem v intervalu <0;8>
6. vyjadřuje obsah B pole, je reprezentováno číslem v intervalu <0;CORE_SIZE[*]>

Válečník je potom vyjádřen množinou 1 až N takovýchto 6-tic čísel. Poté můžeme nad touto reprezentací pracovat jako v *Evolučních strategiích.*

Dalším možným způsobem je podívat se na předcházející reprezentaci jako na binární vektor, a poté na něj uplatňovat metody Genetických algoritmů. Zde však narážíme na některé problémy, spojené s specifikací RedCode. Nakolik nejsou všechny rozsahy čísel ve tvaru 2^k, kde k je celé číslo, není možné každou z těchto hodnot přímočaře interpretovat k bity. Proto pokud se rozhodneme implementovat tento způsob reprezentace, budeme muset implementovat také funkce pro přepis těchto čísel do binárního vektoru. Toto je možné opět implementovat více způsoby. Buď můžeme každému číslu přidělit takový počet k bitů, aby maximální nabývaná hodnota $M \leq 2^k$. Poté pokud při reprodukci vznikne část vektoru, která nemá při přepisu do 6-tice čísel význam, je vybráno náhodné číslo, které význam má.

Druhý způsob je implementovat funkci, která přepíše tuto 6-tici čísel na binární vektor tak, aby maximálně využila použité kódovací bity. To způsobí, že některé bity budou v závislosti na kontextu kódovat různé vlastnosti.

---

* CORE_SIZE je velikost paměti MARSu

Příklad:

Mějme vlastnost, která může nabývat 5-ti různých hodnot. Zakódujme je pomocí sekvencí:
000, 001, 010, 011, 100
Při pohledu na tuto sekvenci zjistíme, že pokud je na prvním místě bit 1, tak již víme, že sekvence reprezentuje poslední hodnotu. Proto 2 zbylé bity již můžeme použít na zakódovaní další hodnoty.

Oba předchozí způsoby kódování válečníků se pokoušeli obytně zakódovat každou instrukci válečníka zvlášť. Třetí způsob se snaží o nalezení jisté gramatiky, pomocí které by bylo možné program napsaný v RedCode rozepsat na posloupnost gramatických pravidel. Mutace a křížení by se potom aplikovala na vektor těchto pravidel. Tento přístup je celkem slibný, avšak velkým problémem s jeho realizací je nalezení dané gramatiky a gramatických pravidel.

3.4 Mutace válečníků

Přístup k mutaci válečníků záleží na jejich zvolené reprezentaci. Pokud jsme zvolili reprezentaci pomocí bitového vektoru, mutaci provedeme prostou inverzi bitů, kterou provedeme s každým bitem s pravděpodobností *p(m)* jak bylo popsáno v kapitole 2.6.

Pokud jsme zvolili variantu kódování válečníků pomocí 6-tic celých čísel, má smysl postupovat pro různá čísla různým způsobem. Hodnoty čísel reprezentující instrukci, modifikátor či adresní módy, jsou plně nezávislé - žádné 2 instrukce si nejsou bližší. Nemá smysl aby mutace instrukce SPL → ADD probíhala častěji jako mutace SPL → MOV apod. Proto nemá smysl uvažovat o mutaci Gaussovským rozložením a tak jednoduše s pravděpodobností *p(m)* změníme hodnotu na náhodnou hodnotu z odpovídajícího rozsahu. Naproti tomu čísla reprezentující obsahy polí A a B vyjadřují přímo hodnoty parametrů instrukcí a proto u nich zřejmě má smysl aplikovat mutaci pomocí Gaussovského rozložení, jak bylo popsáno 2.7 . Tímto způsobem dosáhneme toho, že mutace:
ADD #**2** ; $1 → ADD #**3** ; $1
bude probíhat častěji jako mutace:
ADD #**2** ; $1 → ADD #**18**; $1
což se zdá být logický postup.

Jednou z možností přístupu k mutaci válečníků je i varianta, při které procházíme válečníka po instrukcích, přičemž s pravděpodobností *p(v)* je aktuálně procházená instrukce vypuštěna (smazána) a obvykle se stejnou pravděpodobností je na každém místě vygenerována nová náhodná instrukce. (Tento druh mutace byl uplatněn v [4]).

Pokud postavíme reprezentaci válečníků na posloupnosti gramatických pravidel, mutace budou probíhat přímo nad těmito pravidly. V tomto kontextu může mutace v důsledku změnit celý podstrom

programu. Funkce přepisu gramatických pravidel na Redcode však způsobí, že jednotlivé instrukce budou zapadat do jistých sémantických bloků.

3.5 Křížení válečníků

Při volbě metody křížení opět hlavní úlohu sehrává zvolená reprezentace válečníků. Při reprezentaci bitovým vektorem zvolíme jednobodové nebo vícebodové křížení jak bylo popsáno v kapitole 2.6. Můžeme pouze omezit, aby body křížení ležely vždy mezi instrukcemi – tedy považovat instrukci za nedělitelný gen. Stejně budeme postupovat i pokud reprezentujeme válečníka pomocí vektoru 6-tic čísel.

U reprezentace válečníka pomocí vektoru gramatických pravidel budeme při křížení manipulovat s celými postupnostmi těchto pravidel a tím přesouvat mezi válečníky vždy celé podstromy programu (předpokládám, že zvolená gramatika bude generovat stromovou strukturu). Tímto způsobem je možné přenášet mezi válečníky celé bloky funkčního kódu.

3.6 Předchozí práce v tomto směru

Jason Boer – ga_war.c [4]

Jedním z prvních pokusů o evoluci CoreWars válečníků byl program Jasona Boera nazvaný ga_war napsaný v jazyce *c*. Algoritmus je převážně postavený na Evolučních strategiích. Boer zvolil v tomto programu přímou textovou reprezentaci válečníků. Všechny operace probíhají přímo nad polem znaků, což vede ke snížené čitelnosti kódu a také dost nízké rychlosti. Pro simulaci soubojů válečníků bylo použité prostředí *pmars* (portable mars) které je v programu externě volané. Proto je nutné válečníky před soubojem zapsat do souboru. Z tohoto důvodu **je pro bezpečí nutné tento program spouštět vždy z RAM disku. Při spuštění z fyzického disku hrozí nebezpečí jeho poškození** z důvodu příliš častého přístupu. Samozřejmě by to také výpočet programu neúměrně zpomalilo.

Jako operátor mutace je v tomto programu použitá funkce, která s pravděpodobností *p(m)* změní část instrukce (op code, modifikator, adresní mód, hodnotu pole) na libovolnou jinou přípustnou hodnotu. Dále se každá instrukce s pravděpodobností *p(v)* vynechá a mezi každé 2 instrukce se s pravděpodobností *p(i)* vloží náhodně vygenerovaná instrukce. Program neimplementuje žádný operátor křížení. Selekce je vykonávána tak, že v zvláštním adresáři (se jménem *valhalla*) se udržují nejlepší válečníci. S určitou pravděpodobností je potom nejslabší jedinec populace přepsán náhodným válečníkem z adresáře *valhalla.* Pokud některý válečník dosáhne většího skóre jako

nejhorší válečník ve valhalla, je válečník zkopírován do adresáře valhalla a poražený válečník sestupuje do normální populace.

I přes svá zjevná omezení tento program vygeneroval některé celkem schopné válečníky.

George Lebl – Sys4

Také patří mezi první programy pokoušející se o evoluci válečníků. Autor si však upravil specifikaci prostředí, a tak není možné hovořit přímo o CoreWars válečnících. V tomto programu je paměť simulátoru považována za dvou-rozměrné pole, ve kterém válečníci bojují proti svým sousedům. Program má zajímavé uživatelské rozhraní, ve kterém je možné v reálném čase sledovat vývoj situace v paměti a v pozdějších verzích dokonce simulaci pozastavit a stav paměti poupravit.

Největší popisovaný problém tohoto programu je absenci vhodné funkce pro iniciaci počáteční populace a nedostatečně propracovaná fitness funkce, která porovnává vyvíjené jedince pouze mezi sebou. Pak když je populace z velké části zaplněna nefunkčními jedinci, kteří zemřou bez přičinění jejich soupeře, získávají i velmi naivní válečníci neúměrně vysoká hodnocení. Typickým příkladem je např. válečník, který pozůstávající pouze z instrukce SPL – tedy v každém kroku program vytvoří nový proces. Program tvoří sada pearlových skriptů.

Terry Newton

Tento autor se CoreWars věnuje již několik let, během kterých uveřejnil několik programů pro evoluci CoreWars válečníků. Prvním byl program *Redmaker*, který byl velmi podobný programu Sys4, pouze byl napsán v Qbasic. Dalším jeho programem je *Fizzle*, kterého největší předností bylo zakomponování soubojů proti referenčním válečníkům do fitness funkce. Tím se zvedla objektivita hodnocení válečníků a začali se vyzdvihovat bojeschopní válečníci schopni nejen přežít, ale i účinně útočit na protivníka.

Poslední skupinou programů pro evoluci válečníků tohoto autora jsou programy *REBS* (Redcode Bash Script), *RedMixer* a *MEVO*. všechny tyto programy se opět vracejí k dvou-rozměrné paměti, ve které válečníci bojují proti svým sousedům. REBS je napsán v GNU/Linux Bash, RedMixer v Qbasic a poslední MEVO ve FreeBasic.

David Andersen – The Garden

Projekt The Garden patří k nejlépe propracovaným i zdokumentovaným projektům v tomto směru. Válečníci jsou reprezentováni pomocí bitového vektoru. Informace je vždy zkomprimována tak, aby při libovolné inverzi bitu vektor kódoval platnou instrukci, tak jak je to popsáno v kapitole 3.3. Naplnění počáteční populace je implementováno buď jako čistě náhodné, nebo náhodné s manuálně

vloženými prvními 4-mi instrukcemi nastavenými na SPL (program si na začátku vytvoří 4 procesy, které budou začínat na náhodných pozicích).

Mutace je v tomto programu implementována bitovou inverzí, jak na existujících jedincích tak při reprodukci. Křížení je řešeno pomocí jednobodového či vícebodového křížení jak je popsáno v kapitole 2.6. Jako fitness funkce může sloužit výsledek několika soubojů proti ostatním vyvíjeným válečníkům, proti sadě referenčních válečníků či kombinace obou přístupů.

4 Návrh implementace

V následující kapitole popíšu postup při návrhu mého řešení aplikace pro vývoj Corewar válečníků. Budeme se zabývat volbou jazyka/jazyků pro implementaci a prostředí pro simulaci soubojů válečníků. Dále popíšu způsoby vytváření počáteční generace, vyzkoušené metody mutace a křížení válečníků, použité fitness funkce pro hodnocení válečníků a vyzkoušené operátory selekce. Vzpomeneme také techniky jako auto-regulace řídících parametrů.

4.1 Volba jazyka a prostředí

První otázka před kterou jsem byl postaven byla, v jakém jazyce a za pomoci jakých prostředků chci moji aplikaci implementovat.

Můj první pokus o implementaci pozůstával z několika krátkých propojených skriptů jazyka Bash. Pro tuto kombinaci jsem se rozhodl pro rychlost a snadnost jejího provedení. Tato prvotní aplikace pracovala přímo nad textovou reprezentací válečníků a pozůstávala ze jednotlivých skriptů pro mutaci a samotnou evoluci. Na simulování soubojů mezi válečníky jsem použil program *pmars* (Portable MARS). Tato implementace obsahovala pouze minimalistickou verzi nutnou pro běh evolučního algoritmu. Nebyla tu dokonce implementována fitness funkce v pravém slova smyslu. Selekce probíhala pouze na základě vždy jednoho souboje – utkají se 2 válečníci, poražený je zahozen, výherce svojí mutací vygeneruje 2 potomky do další generace.

Podle očekávání tato implementace neposkytovala žádné uspokojivé výsledky. Její určení bylo zejména pro seznámení se s Corewars a evolučními algoritmy a zjištění problémů v implementaci abych měl představu, jakým směrem se dále ubírat.

Po zkušenosti s touto implementací jsem se rozhodl přejít do poněkud univerzálnějšího a rychlejšího prostředí. Mojí druhou volbou byl jazyk C. Stále jsem však pro simulaci soubojů mezi válečníky využíval externího volání programu *pmars*. Válečníci určeni pro souboj se pro tento program museli nejprve zapsat do souboru a výsledek souboje byl opět programem zapsán do souboru, ze kterého se poté musel přečíst. Tento způsob simulace soubojů byl zjevně neefektivní a pomalý. Nakolik program *pmars* je také napsán v jazyce C, pokusil jsem se využít implementaci simulace souboje válečníků přítomnou ve zdrojových textech programu. Pro nedostatek dokumentace jsem však dosti tápal v otázce interní reprezentace válečníků v tomto programu.

Konečným řešením se poté ukázal být v jazyce C implementovaný simulátor *exhaust* od autora *Joonase Pihlaja*, který má výrazně lépe zpracovanou dokumentaci a je také oproti *pmars* rychlejší. Konkrétně jsem využil jeho verzi optimalizovanou pro rychlost od autora *Martina Ankerla* [7].

Tato aplikace reprezentuje každou instrukci válečníka pomoci 6-tice kladných celých čísel. Válečník je potom reprezentován vektorem takovýchto 6-tic, jak je to popsáno v kapitole 3.3.

4.2 Vytvoření počáteční populace

První verze aplikace vytvářela počáteční generaci čistě náhodně. Tento přístup se však ukázal jako nevyhovující, protože se populace zaplnila jedinci, kteří nebyli sami o sobě schopni přežít. Z tohoto důvodu jsem zvolil strategii, při které opakovaně generuji náhodné válečníky, pokud vygenerovaný válečník není schopen porazit válečníka pozůstávajícího z jediné instrukce *JMP $0, $0*. Tento válečník nedělá nic jiné, jako že neustále dokola vykonává tuto instrukci. Tím bylo zabezpečeno, že vygenerovaní válečníci budou nejenom schopni přežít, ale také nebudou statičtí – podaří se jim zasáhnout do kódu soupeře. Tento způsob generování nás sice stojí na počátku čas, avšak ukázalo se to být dobrou investicí vzhledem na rychlejší konvergenci řešení.

Vytváření válečníků tímto způsobem je však stále čistě náhodný. Dalším krokem tedy bylo pokusit se aplikovat na vytváření válečníka nějaké heuristické postupy, při kterých by jsme prostor řešení alespoň nějak eliminovali a tak generovali válečníky efektivněji. Aplikoval jsem 2 také postupy. Základem prvního bylo, že v Redcode pro každé číslo existuje 1:1 vztah k instrukci na dané relativní adrese. Nasledovala úvaha: nemá smysl se pokoušet pracovat s instrukcemi, kterých obsah neznáme. I pokud by měl některý válečník fungovat na bázi scaneru – tedy aktivně by vyhledával soupeřove instrukce, tak by toto vyhledávání začínalo v jeho okolí a iterativně by se vzdálenost prohledávané instrukce zvětšovala. Navíc je velmi nepravděpodobné, že by takhle vysoce sofistikovaný válečník vznikl při náhodném generování. Proto je logické čísla v argumentech instrukcí generovat namísto v intervalu <0;CORE_SIZE-1> pouze v intervalu obsahujícím samotného válečníka a jeho blízké okolí. Pokud bude válečník délky N a chceme uvažovat okolí válečníka M na každou stranu, generujeme čísla z intervalu <-M;N+M>.

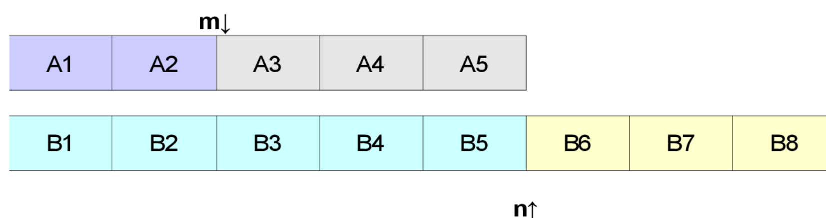
Druhou úvahou pro heuristické zlepšení počátečního generování válečníků byla následující myšlenka. Doposud jsme samotné instrukce (instrukční kódy) generovali čistě náhodně a uniformně. Tedy všechny typy instrukcí jsme generovali se stejnou pravděpodobností. Je však možné, že bude výhodnější generovat instrukce podle jiného rozložení. Pro jeho zjištění jsem napsal další krátký bash-ový skript, pomocí kterého jsem spočetl četnosti jednotlivých instrukcí ve větším vzorku válečníků. Každou instrukci poté generuji s pravděpodobností odpovídající tomuto rozložení.

4.3 Mutace a křížení válečníků

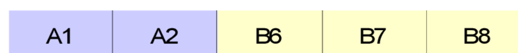
Při mutaci válečníků se u každé instrukce s různými pravděpodobnostmi změní každá část instrukce (instrukční kód, modifikátor, adresní módy, obsahy polí A,B). Při změně instrukčního kódu opět generujeme nový instrukční kód podle rozložení popsaného v předcházející podkapitole. Modifikátor a adresní módy se mění náhodně dle uniformního rozložení. Obsahy polí A a B se mění dle Gaussovského rozložení jak to bylo popsáno v kapitole 3.4.

Křížení válečníků je implementováno jako jednobodové, přičemž bod křížení se nachází vždy mezi jednotlivými instrukcemi – tedy samotné instrukce nejsou křížením ovplyvněny. Nakolik délka válečníka je proměnlivá, metoda křížení musela být pro tento případ mírně pozměněna. U každého rodičovského válečníka je vybráno náhodně místo, ve kterém bude vytvořen zlom křížení (nazvěme je *n* a *m*). Poté je náhodně zvoleno, ze kterého válečníka bude použit začátek, tedy instrukce <0 až n>, a ze kterého konec, tedy instrukce <m až size>.

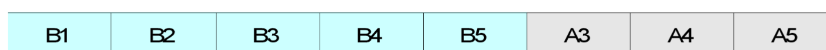
Mějme 2 válečníky různé délky podstupující křížení. Označme si instrukce jednoho válečníka *A1* až *A5* a instrukce druhého válečníka *B1* až *B8*. Dále si zvolme body křížení *m* a *n*.



V závislosti na náhodném rozhodnutí, ze kterého válečníka vezmeme začátek, a ze kterého konec, vznikne buď válečník:



nebo válečník:



Nyní ještě musíme rozhodnout, z jakého bodu bude výsledný válečník startovat. (Redcode umožňuje zadání adresy, na které program začíná, tedy vykonávání programu nemusí začínat instrukcí na adrese nula.) Nejprve se tedy podíváme, zda se ve vybraných úsecích nacházejí instrukce, na kterých začínali rodičovské programy. Pokud se v potomkovi nachází právě jedna instrukce, na které začínal rodič, je jako startovací bod zvolena tato instrukce. Pokud jsou přítomny startovací instrukce obou rodičů, je z nich startovní instrukce vybrána náhodně. Pokud není přítomna ani jedna startovní instrukce rodičů, startovní instrukce se vybere náhodně z celé délky potomka.

4.4 Fitness funkce a selekce

Fitness funkce v mé aplikaci prošla docela dlouhým postupným vývojem. První verze obsahovala aditivní fitness, tedy u každého válečníka se pamatovala momentální fitness, která se po jeho souboji vždy zvýšila/snížila v závislosti na výhře/remíze/prohře. Nebylo tedy přesně dodrženo klasické schéma evolučního algoritmu, ve kterém vždy nejprve vyhodnotíme celou aktuální populaci, poté vybereme ty úspěšnější a křížením a mutací obnovíme generaci.

Vycházel jsem z modelu, ve kterém se populace obměňovala postupně s tím, že vyšší pravděpodobnost k výběru pro reprodukci měli jedinci s vyšší aktuální hodnotou fitness. V každém cyklu algoritmu byli tedy náhodně vybráni 2 jedinci, kteří se utkali a na základě výsledku souboje byli upraveny jejich fitness. Při této úpravě se přihlíželo také na dosavadní fitness válečníků, kteří stáli proti sobě. Např. pokud válečník s vysokou fitness remizoval s válečníkem s nízkou fitness, tak válečníkovi s vysokou fitness byla hodnota fitness snížena a naopak válečníkovi s nízkou fitness zvýšena. Poté se pomocí lineárního rankingu vybral jeden jedinec, kterého kopií byl nahrazen nejslabší jedinec v populaci. Nakonec byl na oba válečníky, kteří se v tomto kole účastnili souboje s jistou pravděpodobností aplikován operátor křížení a operátor mutace.

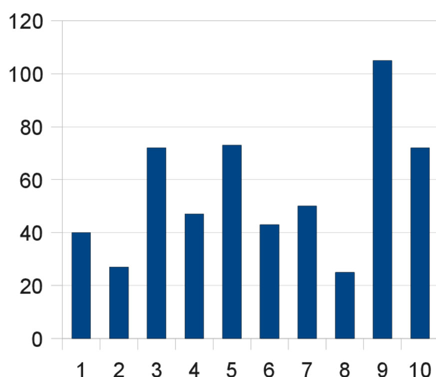
Tento přístup měl chybu v tom, že pro vyhodnocení kvalit válečníka rozhodně nestačí jeden souboj, a souboje, které byli zahrnuty v jeho fitness aditivním principem z velké části nevypovídali o daném válečníkovi, protože od soubojů, ve kterých tuto fitness získal mohl být už mnoho krát pozměněn.

Z tohoto důvodu jsem přešel ke způsobu hodnocení pomocí fitness funkce, která nechala proběhnout větší množství soubojů proti náhodným soupeřům z populace a také proti sadě lidmi napsaných referenčních válečníků, kteří jsou již schopni účinně bojovat.

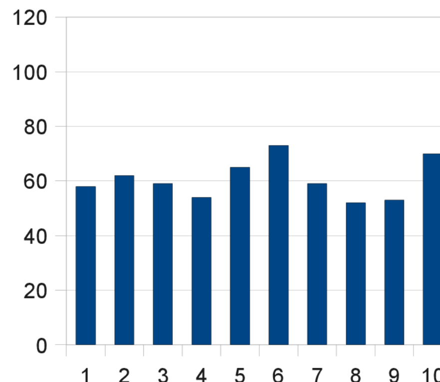
Tento způsob přinesl věrohodnější a přesnější hodnocení jedinců, na druhé straně se celý proces výrazně zpomalil. Bylo potřebné nalézt hranici, kolik soubojů je nutné uhrát během hodnocení jednoho válečníka, aby bylo hodnocení dostatečně přesné, ale na druhé straně aby nebylo zbytečně pomalé. Za tímto cílem jsem si nechal v každé generaci jednoho náhodně vybraného válečníka ohodnotit mojí fitness funkcí 10x za sebou. Poté jsem měnil hodnoty parametrů, kolik soubojů se odehraje proti ostatním jedincům v populaci a kolik proti referenčním válečníkům a sledoval jsem jaký rozptyl mají jednotlivá hodnocení stejného válečníka.

Pro znázornění uvádím grafy, ve kterých jsou vykresleny hodnoty, které vrátila fitness funkce při deseti zavoláních na stejného válečníka. Graf vlevo zobrazuje situaci, když byla fitness funkce nastavena na odehrávání 10-ti soubojů proti jedincům z populace a 10-ti soubojů proti referenčním válečníkům. V tomto případě vidíme, že hodnoty jsou ještě příliš rozptýlené a nabývají hodnot 27 až 105. Do grafu vpravo byly vykresleny výsledky při odehrávání 10-ti soubojů proti jedincům z

populace a 30-ti soubojů proti referenčním válečníkům. V tomto případě jsou již hodnoty které vrátila fitness funkce celkem konzistentní a všechny pochází z intervalu 52 až 73. Toto nastavení jsem proto vyhlásil za vyhovující.



10 soubojů s referenčními válečníky a 10 soubojů s válečníky z populace.



30 soubojů s referenčními válečníky a 10 soubojů s válečníky z populace.

Po této změně fitness funkce jsem také přešel ke klasickému schématu evolučních algoritmů jak bylo popsáno v kapitole 2.2 tedy: ohodnocení generace → selekce → křížení a mutace → ohodnocení populace...

V této verzi aplikace selekci vykonávám pomocí turnajové selekce, jak byla popsána v kapitole 2.5.

4.5 Automatická regulace řídících parametrů

Nakolik jednotlivé parametry ovlivňující chod evoluce může být vhodné měnit, rozhodl jsem se na některé parametry použít metody automatické regulace. Konkrétně jsem je uplatnil na 2 typy parametrů.

Za prvé to byly parametry určující pravděpodobnost mutací a křížení. Dle teorie evolučních algoritmů je vhodné pokud přibližně 1/3 potomků dosáhne lepšího hodnocení jako jeho rodič/e. Pokud je počet úspěšnějších potomků vyšší, zvýšíme pravděpodobnost a míru mutací, pokud je úspěšnějších potomků méně, míru mutací snižujeme. [1] Tento postup jsem ještě upravil ve smyslu, že úpravy parametrů mutací a křížení se začínají uskutečňovat až od 20-té generace.

Zpočátku evoluce je totiž populace naplněna z velké části náhodnými jedinci. Při jejich náhodné mutaci vznikají jedinci lépe ohodnoceni s pravděpodobností téměř 50%. Toto se pokousí algoritmus řešit pomocí zvýšení míry mutací, čím však pouze podpoří náhodnost válečníků v populaci. To opět vede k relativně vysoké úspěšnosti potomků, co se algoritmus pokouší opět řešit

zvýšením míry mutací. Pokud se tento mechanizmus zapne až po určité době, jsou už válečníci v populaci schopnější a je větší pravděpodobnost, že náhodná mutace spíše něco pokazí, jako opraví. Jak jsou válečníci v průběhu evoluce stále vyspělejší, je pravděpodobnost že bude pozměněný potomek lépe ohodnocen jako jeho rodič je stále menší, na co algoritmus reaguje snižováním míry mutací a tak se již nerozbíjejí vytvořené struktury, pouze se odlaďují stále drobnějšími změnami.

Druhým typem parametrů, který aplikace upravuje za běhu je počet bodů udělených fitness funkcí válečníkovi za výhru/remízu s referenčním válečníkem. Na počátku je logické za poražení schopného referenčního válečníka přidělit násobně více bodů, jako za poražení téměř náhodného válečníka z populace. Postupem generací, jak se válečníci v populaci stále zlepšují, je opodstatněné za poražení referenčního válečníka přidělovat méně bodů. V mojí aplikaci jsem se rozhodl pro přidělování bodů za poražení referenčního válečníka dle vzorce: $B = \frac{Rs}{(Rv+1)} * 2$ kde Rs je počet soubojů s referenčními válečníky a Rv je počet výher. Počet bodů za remízu je vždy roven 1/3 bodů za výhru.

5 Výsledky práce

S aplikací navrženou pomocí postupů popsaných v kapitole 4 této práce jsem provedl množství experimentů. Jejich cílem bylo pomocí evolučních algoritmů „vypěstovat“ Corewar válečníka, který by byl schopný co nejlépe obstát v boji proti lidmi napsaným válečníkům. V této kapitole poskytuji rozbor výsledků, kterých se mi při těchto experimentech podařilo dosáhnout.

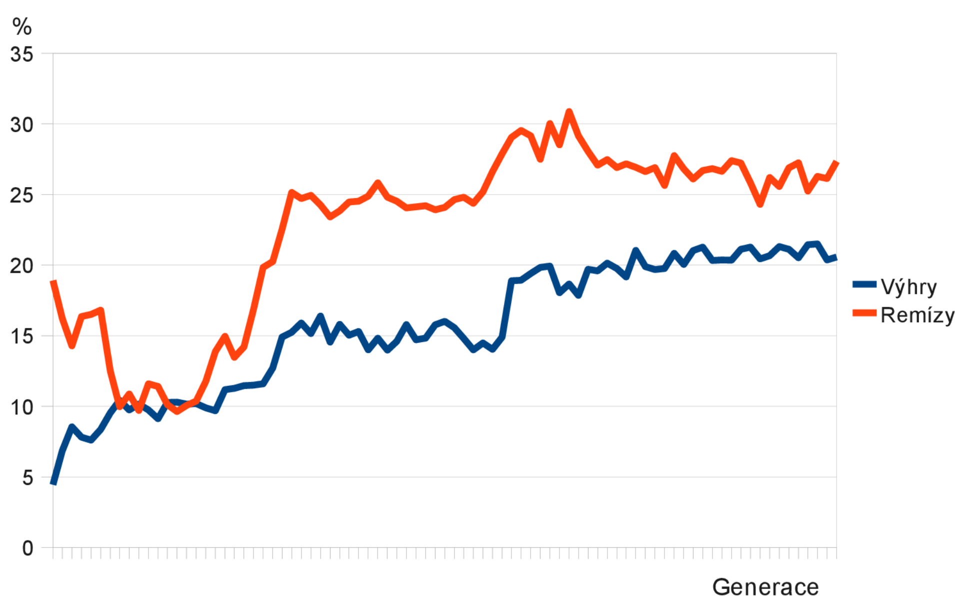
5.1 Typický vývoj populace

Pro hodnocení kvality populace jsem využil soubojů s referenčními válečníky, kteří byli napsáni a publikováni různými autory. Konkrétně jsem použil válečníky:

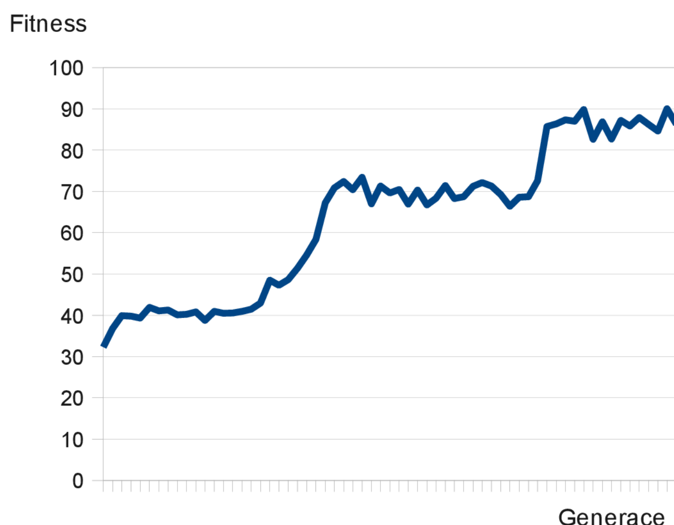
`PaperLp, Sapphire, Shelter1f3, Syzygy1.0, Syzygy1.3, Vampirism1.2, aeka, flashpaper, mice, rave, silkwarrior13`

Při mých experimentech s aplikací jsem potom sledoval právě úspěchy vyvíjených válečníků proti těmto vybraným referenčním válečníkům. V této podkapitole nejprve uvedu grafy úspěšnosti válečníků v populaci proti referenčním válečníkům, v závislosti na generaci. Poté se pokusím výsledky analyzovat a vyvodit z nich závěry. Po provedení množství experimentů s různými nastaveními parametrů evoluce jsem vybral ukázkový vývoj populace. Tohoto vývoje bylo dosaženo v populaci o 50-ti jedincích s nízkou pravděpodobností mutace a křížení.

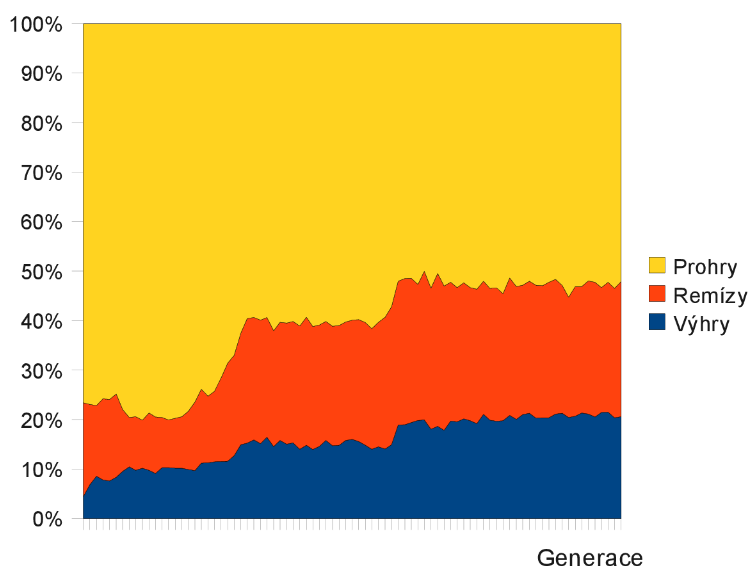
Následující graf znázorňuje procentuální podíly výher a remíz při soubojích jedinců z populace proti referenčním válečníkům.



V počáteční populaci vyhrávali jedinci nad referenčními válečníky v méně jako 4,5% případů. Poté nastává rovnoměrný vzestup, až po přibližně 425 generacích se tato hodnota ustálila mezi 20% a 22%. Remizovali jedinci z počáteční populace v téměř 20% soubojů. Nasleduje pokles počtu remíz, který pravděpodobně souvisí s prudkým vzestupem výher – algoritmus zvýhodňoval válečníky, kteří vyhrávali tam, kde předchozí generace remizovali. Pokles počtu remíz je dokonce větší, jako nárůst počtu vítězství. Je to dáno tím, že vítězství jsou ohodnocena vždy trojnásobným bodovým ziskem, jako remíza. Pokud vyneseme na graf samotnou fitness, kterou válečníci získají za souboje, bude mít graf stoupající tendenci *(viď následující graf)*. Poté u počtu remíz následuje další vzestup který se po 425 generacích ustálí mezi 25% a 29%.

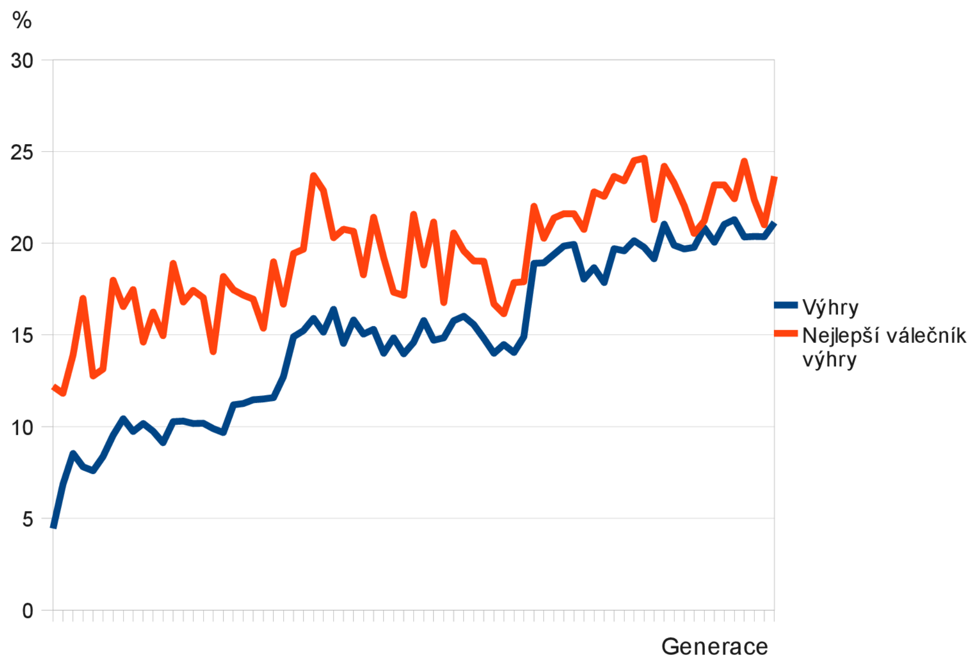


Následující graf zobrazuje procentuální podíly výher, remíz a proher v soubojích proti referenčním válečníkům, jako podíly plochy. Uvádím ho pro jeho nejvyšší názornost.



Vidíme, že zatímco počáteční populace prohrávala ve více jako 76% soubojů, populace po 425 generacích již prohrává přibližně v polovině soubojů. Významný je také nárůst počtu výher, který se během těchto 425-ti generací téměř zpětinásobil.

Na dalším grafu zobrazuji porovnání průměru výher vždy celé generace a výher nejlepšího jedince z dané generace.



V grafu vidíme zejména 2 věci.

1. hodnoty pro nejlepšího válečníka jsou více chaotické – vyplývá z toho, že pro ohodnocení jednoho válečníka používáme méně soubojů a tak je znatelnější náhodný faktor (zda se válečníkovi zadaří, nebo ne)
2. Nejlepší válečník v generaci je lepší jako průměr generace zejména během začátku evoluce, později je rozdíl již minimální a způsobený spíše náhodnými jevy při ohodnocování (válečník je stejný jako ostatní v populaci, ale zadařilo se mu při soubojích).

5.2 Příklad vypěstovaného válečníka

Závěrem uvádím příklad Corewar válečníka, kterého se mi podařilo vypěstovat pomocí výše popsané aplikace.

046.red:

```
;assert 1
;autor Martin Triska
;name 046
ORG START
dat.ab {   10 , }    3
dat.i  >    6 , <    7
mov.i  *    6 , }    3
dat.ba #    3 , <    2
dat.i  <    5 , }    5
spl.i  #    6 , $    7
dat.i  #    6 , }    3
djn.f  #    3 , @    2
dat.b  }    5 , #    5
dat.ab }    6 , >    3
djn.f  #    3 , >    2
dat.x  <    5 , *    5
dat.ab {   10 , }    3
spl.i  #    6 , <    7
mov.i  {    6 , }    3
djn.f  #    3 , <    2
dat.i  <    5 , *    5
spl.i  #    6 , $    7
dat.ab #    6 , }    3
dat.a  #    3 , @    2
dat.b  }    5 , #    5
spl.x  }    6 , >    3
djn.f  #    3 , >    2
dat.x  <    5 , *    5
dat.ab {   10 , }    3
spl.i  *    6 , <    7
mov.i  *    6 , }    3
djn.f  #    3 , <    2
dat.i  <    5 , }    5
spl.i  #    6 , $    7
dat.a  *    6 , }    3
djn.f  }    3 , @    2
add.ba <   10 , }    3
START spl.i  #    6 , >    7
mov.i  *    6 , }    3
djn.f  #    3 , <    2
dat.b  <    5 , #    5
dat.i  #    6 , $    7
dat.f  #    6 , }    3
spl.f  #    3 , #    2
dat.b  >    5 , #    5
dat.ab }    6 , >    3
dat.x  #    3 , >    2
dat.x  <    5 , *    5
djn.f  @    3 , >    2
jmz.b  <    5 , >    5
dat.ba {    6 , >    3
dat.b  #    3 , <    2
add.f  #    5 , *    5
dat.b  @    5 , <    5
spl.i  <    6 , <    3
dat.x  {    3 , }    2
dat.a  {    5 , >    5
dat.x  @    5 , @    5
mov.f  }    2 , }    9
```

6 Závěr

V průběhu psaní této práce jsem nabyl mnoho nových teoretických i praktických poznatků o evolučním programování, jeho možnostech využití i problémech při jeho nasazení. Dále jsem se seznámil s programátorskou hrou Corewars a jazykem Redcode a v práci jsem důležité prvky této hry popsal. S využitím těchto nových znalostí jsem implementoval aplikaci postavenou na principech evolučních algoritmů, která úspěšně vytváří programy v jazyce Redcode. Tyto programy sice nejsou schopné obsadit popřední místa řebříčků, avšak jsou do jisté míry schopné obstát i v boji proti lidmi napsaným programům.

Tímto jsem dle mého názoru splnil zadání práce. Detaily popisující jazyk Redcode a programátorskou hru Corewars jsou uvedeny v 1 kapitole. Studiem evolučních algoritmů a metod evoluční strategií se zabývá kapitola 2. Třetí bod zadání pojednávající o návrhu evolučního algoritmu, včetně metod křížení, mutace a evaluace fitness je podrobně rozepsán v 3 kapitole. Čtvrtá kapitola popisuje návrh a samotnou implementaci řešení diskutovaného problému. Dosažené výsledky jsou diskutovány v závěrečné, páté kapitole práce.

Perspektiva další práce v této oblasti dle mého názoru spočívá v nalezení vhodnější reprezentace válečníků tak, aby malé změny v genomu odpovídali malým změnám ve fenotypu jedinců. V kapitole 3.3 jsem již zmínil jeden z možných způsobů řešení, pomocí nalezení gramatických pravidel. Nalezení takovéto gramatiky však bude velmi náročné, vzhledem k možnostem, které nabízí jazyk Redcode a vzhledem ke skutečnosti, že Corewar válečníci velmi často využívají konstrukce, při kterých se jejich kód sám při běhu programu upravuje.

V oblasti evolučních algoritmů je ještě mnoho neprozkoumaných možností, kterých průzkum jistě povede k dalšímu pokroku v teto zajímavé části informatiky.

Literatura

[1] SCHWARZ, Josef; SEKANINA, Lukáš. *Aplikované evoluční algoritmy EVO, Studijní opora*. [s.l.] : [s.n.], 2006. 101 s.

[2] ANDERSEN David G. *The Garden: Evolving Warriors in Core Wars,* August 23. 2001, 13s. [online]. 5.5.2010 [cit. 5.5.2010]. Dostupný z WWW: <http://www.angio.net/res/garden.pdf>

[3] *A Brief History of Corewar,* [online]. 5.5.2010 [cit. 5.5.2010]. Dostupný z WWW: <*http://corewar.co.uk/history.htm*>

[4] BOER J. *Ga_war.c*, zdrojový soubor, 20.03.1997 [online]. 5.05.2010 [cit. 5.5.2010]. Dostupný z WWW: <http://www.corewar.co.uk/boer/ga_war.c >

[5] HILLS D. Evolving core warriors 1998, [online]. 5.05.2010 [cit. 5.5.2010]. Dostupný z WWW: <http://www.infionline.net/~wtnewton/corewar/evol/evolving.txt>

[6] NEWTON Terry, *Evolved Core-Warriors,* [online]. 5.05.2010 [cit. 5.5.2010]. Dostupný z WWW: <http://www.infionline.net/~wtnewton/corewar/evol/>

[7] ANKERL Martin, *Martin Ankerl's site,* [online]. 8.05.2010 [cit. 8.5.2010]. Dostupný z WWW: <http://corewar.co.uk/ankerl/>

[8] KVASNIČKA, Vladimír. *Evolučné algoritmy*. Slovenská technická univerzita : [s.n.], 2000. 215 s. ISBN 80-227-1377-5.

Seznam příloh

Příloha 1. CD obsahující zdrojové texty, návod na instalaci a návod na použití.