VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

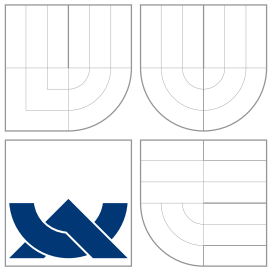
GENETICKÉ PROGRAMOVÁNÍ S JAZYKEM BRAINFUCK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

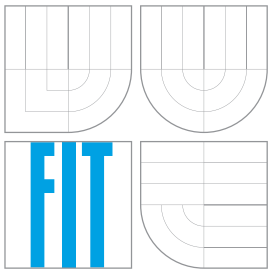
AUTOR PRÁCE
AUTHOR

MILOŠ MINAŘÍK

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENETICKÉ PROGRAMOVÁNÍ S JAZYKEM BRAINFUCK

GENETIC PROGRAMMING WITH BRAINFUCK LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MILOŠ MINAŘÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYŠEK GAJDA
SUPERVISOR

BRNO 2008

Abstrakt

Tato práce se zabývá výzkumem možností použití jazyka brainfuck ke genetickému programování. V teoretické části jsou rozebrány základní mechanismy genetického programování (selekce, křížení, mutace), jejich různé varianty a použití. Praktická část obsahuje kapitoly zabývající se samotnou implementací programového rozhraní pro tuto práci a prováděné experimenty. Výsledky získané vyhodnocením experimentů jsou shrnuty v záverečné části, kde je rovněž nastíněn další možný vývoj v oblasti genetického programování s jazykem brainfuck.

Klíčová slova

genetické programování, brainfuck, evoluční algoritmy

Abstract

This thesis deals with the research of possible usage of brainfuck language for genetic programming. In the theoretical part, there are described basic mechanisms of genetic programming (selection, crossover, mutation), their different versions and usage. Practical part includes chapters considering implementation of program interface for this thesis and the experiments done. Results from the experiments are discused in the final chapter, where the future development in the are of genetic programming with brainfuck language is also mentoined.

Keywords

genetic programming, brainfuck, evolution algorithms

Citace

Miloš Minařík: Genetické programování s jazykem
Brainfuck, bakalářská práce, Brno, FIT VUT v Brně, 2008

Genetické programování s jazykem Brainfuck

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyška Gajdy

.....
Miloš Minařík
13. května 2008

Poděkování

Děkuji Ing. Zbyšku Gajdovi za pomoc s teoretickým základem této práce a rady k formální stránce textové části.

Obsah

Kapitola 1

Úvod

1.1 Současný stav

V současné době jsou evoluční techniky velmi perspektivním odvětvím, neboť existuje celá řada problémů, jejichž řešení klasickými metodami by trvalo na běžném počítači i několik tisíc let. Celou řadu těchto problémů lze však pomocí těchto technik řešit v čase nesrovnatelně kratším.

Tato zřejmá výhoda způsobila, že se evoluční techniky používají v celé řadě různých oborů, například strojírenství, lékařství, výpočetní technice, návrhu elektronických přístrojů a dalších.

Vzhledem k výraznému pokroku především v 90. letech minulého století je pravděpodobné, že evoluční techniky nabízí ještě poměrně velký potenciál. Objevy nalezené pomocí těchto technik v posledních letech to jen potvrzují.

1.2 Motivace

Toto téma jsem si vybral, protože evoluční techniky jsou dle mého významným odvětvím výpočetní techniky. Vzhledem k tomu, že rychlost počítačů se nezvyšuje příliš výrazně, je nutné hledat jiné metody, jak řešit úlohy s rozsáhlým stavovým prostorem. Existuje celá řada problémů, jejichž řešení klasickými metodami by trvalo velmi dlouho. Při použití evolučních technik je možné tuto dobu velmi významně zkrátit. Rovněž mi přijde poměrně fascinující, že počítač může takto nalézt i řešení lepší, než jaké je schopen vymyslet člověk. Pomocí evolučních technik byla objevena i řada nových řešení, která byla natolik výjimečná, že byla patentována (viz [9]).

Dalším důvodem, proč jsem si toto téma vybral je skutečnost, že se evolučními technikami zabývám již delší dobu a genetické programování s jazykem brainfuck se mi zdálo jako vhodné téma pro rozšíření mých znalostí a zkušeností v této oblasti.

1.3 Obsah kapitol

Druhá kapitola obsahuje popis jazyka brainfuck. Je zde rozebrána historie jazyka, jeho základní příkazy a několik příkladů jeho použití. V kapitole 3 je popsán teoretický základ evolučních algoritmů a jejich techniky. Čtvrtá kapitola se již věnuje samotnému genetickému programování. Jsou zde například uvedeny různé způsoby reprezentace jedinců. Kapitola 5 se zabývá samotnou implementací programové části, tedy použitými třídami, funkcemi

apod. V šesté kapitole jsou rozebrány provedené experimenty, provedené změny návrhu, výsledky hledání a další. Sedmou kapitolou je závěr, v němž jsou zhodnoceny dosažené výsledky a navržen další vývoj. Na konci jsou přílohy obsahující deklarace tříd, struktur a funkcí v jazyce C++.

Kapitola 2

Brainfuck

V této kapitole jsou uvedeny základní informace. Více podrobností lze nalézt například v [2] a [3], ze kterých jsem čerpal.

2.1 Ezoterické jazyky

Ezoterický programovací jazyk bývá navržen pro účely experimentování, jako obtížný programovací jazyk, který má prověřit schopnosti programátora, nebo zkrátka jen jako vtip, spíše než pro praktické využití. Existuje malá, ale velmi aktivní internetová komunita, která vytváří ezoterické programovací jazyky, píše pro ně programy a diskutuje o jejich výpočetních vlastnostech (například, zda je jazyk turing-kompletní). Kromě toho existuje celá řada různých fór, kde se tito lidé setkávají.

2.1.1 Historie

Prvním známým ezoterickým jazykem byl jazyk INTERCAL, který v roce 1972 navrhnul Donald R. Woods a James M. Lyon. I když tento jazyk byl první, jsou v dnešní době populární spíše jazyky brainfuck a Befunge, oba z roku 1993. Většina nových ezoterických jazyků je značně ovlivněna právě těmito dvěma jazyky.

Slovo ezoterický použil poprvé Chris Pressey na svých webových stránkách a tento termín se později velmi rozšířil a začal se běžně používat.

2.1.2 Účel

Ezoterické programovací jazyky se vytváří z řady různých důvodů. Nejvýraznějším znakem těchto jazyků bývá, že nejsou navrženy pro běžné a produktivní použití jako běžně používané programovací jazyky. V zásadě lze rozlišit několik základních kategorií určení jazyka:

Minimalistické
Častým cílem ezoterických programovacích jazyků je, aby jazyk obsahoval co nejméně instrukcí. Mezi tyto jazyky patří například brainfuck, OISC a Lazy K.

Nové koncepce
Mezi programátory ezoterických jazyků je poměrně oblíbenou činností hledání nových alternativních přístupů k návrhu jazyků. Zástupci této skupiny jsou například Befunge, Thue a Unlambda.

Podivné

Některé jazyky jsou tvořeny s cílem, aby byly podivné a obtížně se v nich programovalo. Účelem jazyka INTERCAL bylo, aby se co nejvíce lišil od běžných jazyků (i když obsahuje několik podobnosti s konvenčními programovacími jazyky. Jazyk Malbolge byl navržen tak, aby se téměř nedal používat.

Tematické

Tyto jazyky jsou navrženy dle různých témat, které nemají nic společného s počítači. Například var'aq je založen na fiktivním jazyku Klingonů. Programy zapsané v jazyce Shakespeare vypadají jako Shakespearovy hry, v jazyce Chef jako kuchařské recepty a v jazyce Piet jsou programy bitmapami, které vypadají jako obrazy nizozemského malíře Pieta Mondriana.

Vtipy

Tato skupina zahrnuje programovací jazyky, které byly navrženy jako vtipy. V některých z nich však přesto lze programovat (například l33t o OOk!), zatímco jiné jsou k programování téměř nepoužitelné (HQ9+ a Bitxtreme).

2.2 Brainfuck

Brainfuck je ezoterický programovací jazyk známý svým extrémním minimalismem. Byl navržen jako výzva pro programátory a není vhodný pro praktické použití. Jeho formálním předchůdcem je jazyk P prime prime (P"), který navrhnul Corrado Böhm v roce 1964. Název tohoto jazyka bývá často upravován na brainf*ck či brainfsck.

2.2.1 Návrh jazyka

Tento jazyk vytvořil v roce 1993 Urban Müller. Jeho cílem bylo navrhnout jazyk, který by bylo možné implementovat co nejmenším překladačem. Inspirací mu byl programovací jazyk FALSE, jehož kompilátor měl 1024 bajtů. Pro brainfuck napsány i kompilátory menší než 200 bajtů.

Jazyk se skládá z osmi příkazů (viz 2.2.2) Program v jazyce brainfuck je sekvencí těchto příkazů. Do programu lze vložit i další znaky, které jsou při jeho provádění ignorovány. Příkazy se kromě cyklů provádí sekvenčně.

Jazyk brainfuck využívá jednoduchý model, který kromě programu obsahuje pole alespoň 30 000 paměťových buněk o velikosti 1 bajt s počáteční hodnotou 0. Dále obsahuje pohyblivý ukazatel do tohoto pole (nastavený na počátku na nejlevější buňku) a dva datové toky pro vstup a výstup (většinou spojené s klávesnicí a monitorem). Tyto datové toky většinou používají ASCII kódování.

Tabulka 2.1: Příkazy jazyka brainfuck

Příkaz	Význam
>	Zvýší hodnotu ukazatele (přesune ukazatel o buňku vpravo).
<	Sníží hodnotu ukazatele (přesune ukazatel o buňku vlevo).
+	Zvýší o 1 hodnotu aktualní buňky (buňky s ukazatelem).
-	Sníží o 1 hodnotu aktuální buňky.
.	Odešle na výstup hodnotu aktuální buňky.
,	Načte ze vstupu hodnotu a uloží ji do aktuální buňky.
[	Pokud je hodnota aktuální buňky 0, přesune se na odpovídající příkaz].
]	Je-li hodnota aktuální buňky různá od 0, přesune se na příslušný příkaz [.

2.2.2 Příkazy

Jazyk brainfuck obsahuje následujících osm základních jednoznakových příkazů, které jsou uvedeny v tabulce 2.1.

Jak již napovídá název jazyka, programy napsané v jazyce brainfuck jsou velmi těžké na pochopení. Je tomu tak částečně proto, že i jednodušší úloha vyžaduje značné množství příkazů a ze samotného textu programu se obtížně určuje stav programu v určitém bodu. To jsou, kromě omezených vstupních a výstupních možností, důvody, proč se brainfuck nepoužívá pro běžné programování. Brainfuck je však turing-kompletní a teoreticky tedy může vyřešit libovolně složitou strojově řešitelnou úlohu. Pochopitelně za předpokladu neomezené paměti.

2.2.3 Příklady

V této části je uvedeno několik ukázek programů v jazyku brainfuck doplněných komentáři. Účelem této části je, aby čtenář plně pochopil strukturu programů a získal povědomí o složitosti programů napsaných v jazyce brainfuck a o základních programových konstrukcích.

Hello World!

```
++++++++++
[>+++++++>++++++++++>+++>+<<<<-] Úvodní smyčka pro nastavení
                                 počátečních hodnot pole
>++.                             Vypíše 'H'
>+.                              Vypíše 'e'
+++++++.                         Vypíše 'l'
.                                Vypíše 'l'
+++.                             Vypíše 'o'
>++.                             Vypíše ' '
<<++++++++++++++.                Vypíše 'W'
>.                               Vypíše 'o'
+++.                             Vypíše 'r'
------.                          Vypíše 'l'
--------.                        Vypíše 'd'
>+.                              Vypíše '!'
```

```
>.                                    Vypíše znak nového řádku
```

Kód je pro lepší čitelnost rozdělen na více řádků a opatřen komentáři. Protože brainfuck ignoruje všechny jiné znaky než `<>+-.,[]`, není pro komentáře nutná žádná zvláštní syntaxe. Kód na prvním řádku uloží do buňky č. 0 hodnotu 10, která slouží k určení počtu iterací cyklu na druhém řádku. Druhý řádek nastaví v cyklu hodnoty buněk č. 1 - 4 po řadě hodnoty 70, 100, 30 a 10. Na třetím řádku se k buňce č. 2 přičte hodnota 2, čímž dostaneme hodnotu 72, tedy ASCII hodnotu písmene H, které se následně vypíše. Obdobným způsobem se vygenerují i ostatní znaky včetně znaku pro nový řádek. Jak je na příkladu jasně vidět, jsou programy v jazyce brainfuck poměrně komplikované a i natolik triviální záležitost jakou je program Hello World! obsahuje poměrně mnoho příkazů.

Vymazání obsahu buňky

```
[-]
```

Význam tohoto kódu je zcela zřejmý. Od hodnoty aktuální buňky se cyklicky odečítá 1, dokud není její hodnota 0.

Součet

```
[->+<]
```

Tento úsek kódu sečte hodnotu aktuální buňky s hodnotou následující buňky. Součet se však provádí destruktivně, takže hodnota aktuální buňky bude na konci cyklu 0. Stejný kód lze použít i k přesunu hodnoty do sousední buňky za předpokladu, že má sousední buňka hodnotu 0.

2.2.4 Přenositelnost

Částečně kvůli tomu, že Urban Müller nenapsal úplnou specifikaci jazyka brainfuck se postupem času vyvinulo několik různých dialektů. Tyto dialekty se liší především v následujících částech:

Velikost buňky

V klasické distribuci mají buňky velikost 8 bitů, což je nejběžnější velikost i u ostatních distribucí. Chcete-li však v jazyce brainfuck číst netextová data, je nutné rozlišovat i znak konce souboru. Proto bývají používány i buňky o velikosti 16 bitů. Některé implementace používají i 32 nebo 64bitové buňky, někdy dokonce i buňky o neomezené velikosti pro ukládání velkých čísel. Programy využívající takto velké buňky však bývají většinou značně pomalé, neboť hodnotu buněk lze měnit jen přičítáním a odčítáním jedničky.

Programy však lze v jazyce brainfuck většinou napsat tak, aby si vystačily s velikostí buňky 8 bitů. V praxi to znamená vyhýbat se hodnotám nad 255 u bezznaménkových buněk a dávat pozor na meze -128 a +127 u buněk se znaménkem.

Velikost paměti

Klasická distribuce pracuje s pamětí o velikost 30 000 paměťových buněk, přičemž ukazatel je na počátku nastaven na nejlevější buňku. Všechny implementace jazyka brainfuck by

tedy měly poskytovat minimálně stejný počet paměťových buněk. Kupodivu tomu tak ve skutečnosti není. Například pro uložení miliontého členu Fibonacciho posloupnosti je třeba dokonce ještě více paměťových buněk. Nejjednodušším způsobem, jak dosáhnout toho, aby byl jazyk turing-kompletní je tedy pole paměťových buněk, které je zprava neomezené.

Některé implementace rozšiřují paměť i směrem vlevo. Není to však běžná funkce a programy by ji tedy neměly využívat.

Když se ukazatel dostane mimo hranice paměti, zobrazí některé implementace chybové hlášení. Další implementace se pokusí paměť zvětšit, zatímco jiné tuto skutečnost ignorují, což může způsobit nepředvídatelné chování. Dalším možným řešením tohoto problému je přesun ukazatele na druhý konec pole paměťových buněk.

Kód konce řádku

Různé operační systémy a někdy také různá programovací prostředí používají odlišné verze ASCII. Nejdůležitějším rozdílem je kód znaku pro konec řádku. MS-DOS a Windows používají většinou CRLF, tedy hodnotou 13 následovanou hodnotou 10. UNIX a jeho následovníci včetně Linuxu a Mac OS X používají jen hodnotu 10, zatímco starší systémy Mac jen hodnotu 13. Kompilátor, který vytvořil Urban Müller, používá jako kód konce řádku na vstupu i na výstupu hodnotu 10, což je i mnohem pohodlnější než používat CRLF. Implementace jazyka brainfuck by tedy měly být schopné pracovat s programy, které tuto hodnotu znaku konce řádku vyžadují. Většinou tomu tak je, některé implementace to však nesplňují.

Tento předpoklad je také v souladu s většinou programů napsaných v jazyce C, které pracují se znakem konce řádku , který má hodnotu 10. V systémech používajících CRLF standardní knihovna jazyka C transparentně přemapuje na výstupu znak a na výstupu naopak, pokud nejsou vstupní a výstupní datové toky otevřeny v binárním režimu.

Konec souboru

Chování příkazu , se při dosažení konce souboru v různých implementacích liší. Některé implementace nastaví aktuální buňku na hodnotu 0, zatímco jiné využívají EOF konstantu jazyka C (což bývá většinou -1). Dalším přístupem je ponechání buňky beze změny. Tento přístup využívá i kompilátor Urbana Müllera. Pro poslední uvedený přístup lze upravit i programy využívající dva předešle přístupy. Pokud program například očekává jako konec souboru hodnotu 0, stačí před každým příkazem , nastavit hodnotu aktuální buňky na 0. Stejně je tomu i v druhém případě. Tato možnost řešení je tedy nejjednodušší.

Kapitola 3

Evoluční algoritmy

V oblasti umělé inteligence jsou evoluční algoritmy (EA) podmnožinou evolučních výpočetních technik. EA využívají některé mechanismy inspirované biologickou evolucí, například reprodukci, křížení, mutaci a selekci. Kandidátní řešení optimalizovaného problému zde sehrávají úlohu jedinců v populaci a fitness funkce určuje schopnost přežití jednotlivých jedinců. Na populaci se následně opakovaně použijí výše zmíněné mechanismy (křížení, mutace...) a tímto způsobem se nalezne řešení.

Evoluční algoritmy produkují vhodná řešení pro různé typy problémů. Je tomu tak proto, že nevychází z žádných předpokladů o prostředí. Tato obecnost je podložena úspěchy v mnoha různých oborech lidské činnosti, například strojírenství, umění, biologii, ekonomii, genetice, robotice a mnoha dalších. Kromě použití pro matematickou optimalizaci se evoluční výpočetní techniky a algoritmy používají i jako experimentální prostředí, ve kterém se ověřují teorie o biologické evoluci a přirozeném výběru.

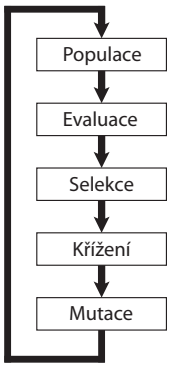
Podle [5] je značným omezením evolučních algoritmů nedostatečné oddělení genotypu a fenotypu. V přírodě projde oplodněné vajíčko složitým procesem zvaným embryogeneze, na jehož konci se z něj stane dospělý fenotyp. Tento nepřímý postup činí genetické prohledávání mnohem robustnějším (například snižuje pravděpodobnost fatálních mutací) a zlepšuje schopnost organismu vyvíjet se. Těmito aspekty se zabývá obor zvaný umělá embryogeneze.

3.1 Implementace biologických procesů

Počáteční generace většinou obsahuje první generaci tvořenou náhodně vygenerovanými kandidátními řešeními. Na tyto jedince je poté aplikována fitness funkce. Existují dva základní druhy fitness funkce. U prvního druhu se fitness funkce nemění, což se využívá především u optimalizací funkcí se známými výsledky nebo při testování proti množině pevně daných hodnot. Druhou možností je fitness funkce, která se může měnit například koevolucí testovacích hodnot.

Během selekce se vybírají rodiče pro další generaci. Tento výběr je ovlivněn fitness hodnotou jedince. Čím vyšší je fitness hodnota, tím vyšší má jedinec šanci na výběr. Následně se na vybrané rodiče použijí mechanismy křížení a mutace. Při křížení se vyberou dva jedinci jako rodiče a výsledkem je jeden nebo dva jedinci (potomci). Mutace pracuje jen s jedním jedincem a na jejím konci je výsledkem opět jeden nový jedinec. Takto vytvoření jedinci soupeří se starými kandidáty o místo v další generaci, do které jsou vybráni ti nejúspěšnější (s nejvyšší fitness hodnotou).

Tento postup se opakuje, dokud není nalezeno vhodné řešení nebo dokud není dosaženo nastavené omezení, například maximální počet generací. Celý proces je shrnutý na následujícím obrázku.



Obrázek 3.1: Evoluční algoritmus

3.2 Techniky evolučních algoritmů

Níže uvedené techniky jsou si podobné, ale mírně se liší v implementaci a povaze problému.

3.2.1 Genetické algoritmy

Tato skupina je nejoblíbenějším typem EA. Hledá se řešení ve formě řetězců čísel. Tyto řetězce jsou většinou binární, i když nejlepší reprezentace jsou většinou takové, které odráží řešený problém, avšak ty nebývají většinou binární. Na tato řešení se kromě selekce a mutace používá i křížení. Tento typ EA se často používá při řešení optimalizačních problémů. Podrobnosti o tomto typu lze nalézt například v [7].

3.2.2 Genetické programování

Zde jsou řešení ve formě počítačových programů a jejich fitness hodnota se určuje podle schopnosti programu řešit daný problém.

3.2.3 Evoluční programování

Evoluční programování je podobné genetickému programování s tím rozdílem, že struktura programu je pevně daná a mohou se měnit pouze jeho parametry. Další informace naleznete v [6].

3.2.4 Evoluční strategie

Řešení jsou zde reprezentována vektory reálných čísel a používají se většinou adaptivní koeficienty mutace. Evoluční strategie jsou podrobněji popsány například v [4].

Kapitola 4

Genetické programování

V této kapitole jsou uvedeny jen základní a stručné informace o genetickém programování. Podrobnější informace můžete nalézt například v [8]. Genetické programování je metoda evolučních algoritmů inspirovaná biologickou evolucí, pomocí níž se hledá počítačový program provádějící uživatelem definovanou úlohu. Je to specializovaná skupina genetických algoritmů, jejíž jednotlivci jsou počítačovými programy. Je to tedy technika strojového učení používaná k optimalizaci populace počítačových programů podle fitness funkce, která reprezentuje schopnost programu vykonávat danou úlohu. Jednou z hlavních osobností genetického programování je John R. Koza, který je rovněž průkopníkem aplikací genetického programování v různých složitých problémech optimalizace a hledání.

GP je velmi náročné na výpočetní zdroje, takže v 90. letech minulého století se využívalo zejména k řešení poměrně jednoduchých problémů. V poslední době však bylo díky pokrokům v GP a exponenciálním růstu rychlosti CPU dosaženo několika úžasných výsledků v různých oblastech, například kvantovém počítání, návrhu elektronických přístrojů, hraní her, třídění, hledání a mnoha dalších. Mezi těmito výsledky jsou i opětovné nálezy různých objevů vynalezených po roce 2000. Pomocí GP bylo nalezeno i mnoho řešení, která byla patentována (viz [9]).

Mnoho zajímavých informací především o různých technikách selekce, křížení a mutace můžete nalézt například v [10] či [1].

4.1 Reprezentace chromozomů

Chromozomy lze reprezentovat dvěma základními způsoby.

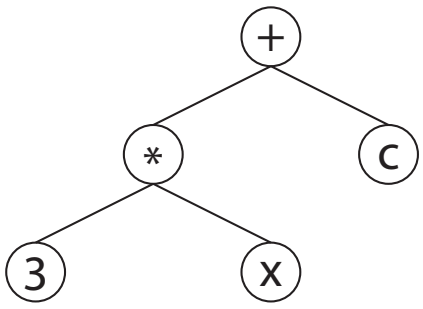
4.1.1 Stromová reprezentace

Toto je nejčastější způsob reprezentace jedinců. Tento způsob je nejvhodnější především u jazyků, které přirozeně pracují se stromovou reprezentací, například LISPu a dalších funkcionálních programovacích jazycích. Další výhodou tohoto způsobu je snadné použití mechanismů křížení a mutací. Rovněž lze u tohoto způsobu provádět jednoduché rekurzivní vyhodnocení.

4.1.2 Lineární reprezentace

Lineární reprezentace je vhodná především pro klasické imperativní jazyky. Tato reprezentace má především tu výhodu, že programy lze v některých případech snadno převádět do

strojového kódu a dosáhnout tak vyššího výkonu.



Obrázek 4.1: Stromová reprezentace

Obrázek 4.2: Lineární reprezentace

4.2 Selekce

Selekce je mechanismus výběru jedinců z populace. Využívá se v podstatě těchto základních přístupů, z nichž každý má své výhody a nevýhody.

4.2.1 Lineární pořadová selekce

Jedinci v populaci se seřadí do seznamu podle hodnoty fitness od nejvyšší po nejnižší a následně se vybere požadovaný počet jedinců. Tento způsob má tu výhodu, že nejlepší jedinec je vybrán vždy, ale má neblahý vliv na různorodost populace.

4.2.2 Vážená ruleta

Tuto metodu si lze představit jako klasickou ruletu, kde je každému jedinci přiřazen určitý počet políček. Čím vyšší je fitness hodnota jedince, tím více políček jedinec zabírá. Výběr je tedy sice náhodný, ale lepší jedinci jsou zvýhodněni.

4.2.3 Turnaj

Turnajová metoda je velmi oblíbená, neboť její implementace je snadná a dosahuje se při ní poměrně dobrých výsledků. Spočívá v tom, že se náhodně vybere určitý počet jedinců, kteří jdou do turnaje. Z turnaje vychází jako vítěz jedinec s nejvyšší hodnotou fitness.

Tato metoda tedy opět zaručuje zvýhodnění lepších jedinců a zároveň také dostatečnou různorodost populace.

4.2.4 Elitářství

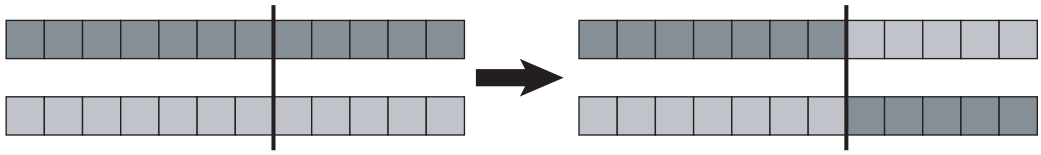
Dvě předešle metody sice zvýhodňují lepší jedince, avšak nezaručují, že se nejlepší jedinec do další generace opravdu dostane. Oproti tomu tato metoda vybere určitý počet (třeba i jednoho) jedinců s nejvyšší fitness hodnotou. Tyto jedince je pak možné umístit do další generace nebo je jen uchovávat někde mimo (tzv. tajné elitářství). Tento způsob zaručuje, že se nemůže ztratit nejlepší nalezené řešení. Lze jej pochopitelně vhodně kombinovat s předchozími dvěma způsoby.

4.3 Křížení

Křížení je základním mechanismem tvorby nových jedinců, neboť bez křížení by populace zůstávala i při použití mutací poměrně málo diverzifikovaná. Mechanismů křížení je několik, ale lze je rozčlenit v zásadě do dvou hlavních kategorií.

4.3.1 Jednobodové křížení

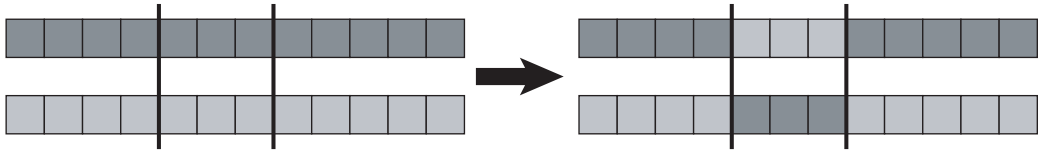
U této metody se náhodně vybere bod, ve kterém se chromozomy rozdělí. Následně se mezi oběma chromozomy vymění části, jak je znázorněno na následujícím obrázku.



Obrázek 4.3: Jednobodové křížení

4.3.2 Vícebodové křížení

Do této části spadá několik různých metod. Liší se například způsobem dělení chromozomů, ale základ zůstává vždy stejný. Chromozomy se rozdělí na několik částí a ty se následně vymění.



Obrázek 4.4: Vícebodové křížení

4.4 Mutace

Mutace jsou rovněž velmi důležitým mechanismem evolučních technik, protože udržuje diverzitu populace. Mutací může být mnoho typů. Nejčastějším typem je změna malé náhodně vybrané části chromozomu. Může to být například bit, znak nebo uzel stromu. Existují však i jiné typy, například záměna dvou částí jednoho chromozomu. V experimentech v této práci je uvedeno i několik dalších typů mutace, které lépe vyhovují řešené úloze.

4.5 Metagenetické programování

Metagenetické programování je technikou vývoje systému genetického programování pomocí genetického programování. U tohoto přístupu se vyžaduje vývoj chromozomů, křížení a mutace, místo aby je určil programátor.

U tohoto přístupu však nelze dokázat, že se řešení nalezne rychleji než při použití klasického GP nebo jiných vyhledávacích algoritmů.

Kapitola 5

Implementace

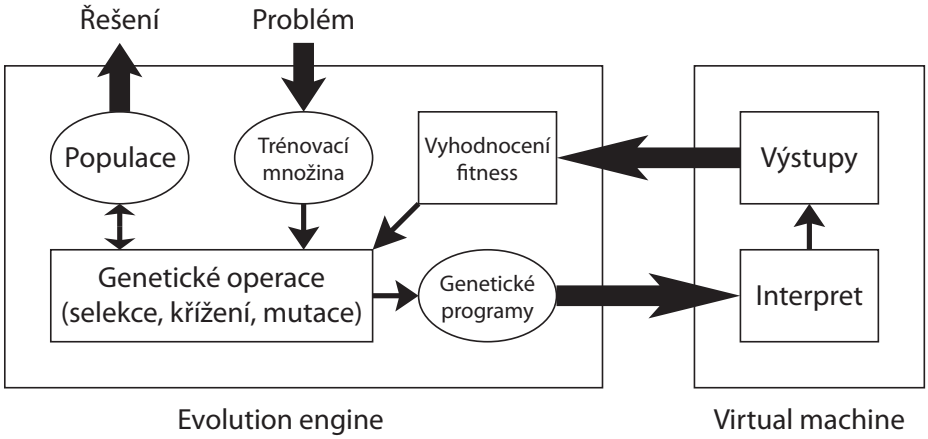
Programová část je implementována v jazyce C++. V následující částech budou uvedeny používané třídy spolu s popisem jejích základních funkcí a rovněž použité mechanismy selekce, křížení a mutace. Podrobnější popis jednotlivých funkcí je obsažen v příloze.

5.1 Přehled

Základní součástí programového rozhraní je třída experiment, která obsahuje všechny potřebné parametry pro provádění daného experimentu (minimální a maximální délku chromozomu, trénovací množinu, maximální počet generací, pravděpodobnosti křížení a mutace a další). Je v ní obsažen rovněž i objekt třídy generace, který obsahuje funkce vykonávající všechny základní mechanismy genetického programování (selekce, křížení, mutace). Generace se skládá z množiny jedinců, kteří představují kandidátní řešení. Tito jedinci pak obsahují reprezentaci chromozomu (lineární nebo stromovou) a fitness hodnotu. Všechny zmíněné třídy zahrnují rovněž funkce pro výpisy různých statistických veličin experimentu, aby bylo možné pohodlně zpracovávat výsledky.

Programátorovi stačí vytvořit jen objekt třídy experiment. O vytvoření objektů všech zbývajících tříd se již objekt experiment postará sám. Pokud by však chtěl programátor změnit například mechanismus křížení, nevyhne se zásahu do zdrojového kódu, což vyžaduje jistou znalost jazyka.

Evoluční experiment komunikuje s virtuálním počítačem (virtual machine), který zpracovává předávané jedince a poskytuje jejich výstupy. Vše je přehledně znázorněno na obrázku 5.1.



Obrázek 5.1: Mechanismus funkce programové části

5.2 Struktury a výčtové typy

5.2.1 Výčtový typ INTERP_RET

Tento výčtový typ slouží k uložení návratové hodnoty z interpretu. V tomto výčtovém typu lze vracet následující hodnoty:

OK	Program byl interpretem vykonán bez problémů.
OUT_OF_BOUNDS	Kód programu způsobil posun ukazatele mimo hranice paměti.
MAX_ITER	Bylo dosaženo maximálního počtu iterací interpretu.
SYNTAX_ERR	V kódu programu byla nalezena syntaktická chyba.

Hodnotu SYNTAX_ERR interpret vůbec nevrací, neboť jsou do něj předávány jen syntakticky správné programy. V tomto výčtovém typu je spíše pro případnou změnu návrhu, kdy by se předávaly i syntakticky nesprávné programy.

5.2.2 Struktura inOut

Do této struktury se ukládá řetězec vstupních znaků spolu s odpovídajícím řetězcem očekávaných výstupních znaků.

5.3 Třídy

5.3.1 Třída insOuts

Tato třída obsahuje vektor struktur inOut (viz A.2). Tento vektor obsahuje vstupy a očekávané výstupy pro jednotlivé běhy programu.

5.3.2 Třída treeNode

Tato třída implementuje uzel stromu reprezentujícího chromozom programu. Pomocí jejích funkcí lze měnit hodnotu uzlu, jeho předka i potomky. Obsahuje rovněž i funkce pro další činnosti, například získání chromozomu ze stromu s kořenem v tomto uzlu.

5.3.3 Třída individual

Třída individual reprezentuje jedince v populaci. V této třídě je v závislosti na reprezentaci jedince obsažen buď řetězec znázorňující chromozom nebo odkaz na kořen stromu. Třída obsahuje i mnoho dalších podpůrných funkcí.

5.3.4 Třída generation

Třída generation zastupuje generaci jedinců. Obsahuje funkce pro základní mechanismy evolučního programování, tedy selekci, křížení a mutaci. Umožňuje také vyhodnocovat všechny členy generace a tisknout statistiky.

5.3.5 Třída experiment

Třída experiment zastupuje experimentální úlohu. Je to hlavní třída experimentu, která po vytvoření při zadání potřebných parametrů již sama vytvoří všechny další podřízené třídy a stará se o jejich správu. Mezi její parametry patří počet jedinců, minimální a maximální délka chromozomu, procentuální šance křížení a mutací a počet členů turnaje.

5.4 Funkce

V této části jsou uvedeny různé funkce, které nejsou obsaženy přímo ve třídách, ale volají se z nich a jsou tedy nezbytnou součástí programu.

5.4.1 generateChromosome

Tato funkce slouží ke generování chromozomů. Je volána z konstruktoru třídy individual (viz A.5). Tuto funkci lze tedy přepsat tak, aby byla pro řešený problém co nejvhodnější. Například je možné určit příkazy na levé a pravé straně používané k načtení a výpisu hodnot, které budou obsahovat všechny chromozomy a omezit tak prohledávaný prostor.

5.4.2 fixBrackets

V této funkci se chromozom upravuje tak, aby byly všechny závorky spárovány. Chromozom obsahující levé nebo pravé hranaté závorky bez příslušných protějšků totiž není syntakticky správný a není tedy platným chromozomem.

5.4.3 expInsOuts

Účelem této funkce je vygenerování vstupů a očekávaných výstupů pro jeden běh programu. Opakovaným voláním této funkce lze tedy vytvořit celou trénovací množinu uloženou v objektu třídy insOuts (viz A.3).

5.4.4 evalFitness

Tato funkce počítá fitness hodnotu jedince. Díky značnému počtu vstupních parametrů je možné tuto hodnotu počítat podle různých kritérií. Tato funkce je jednou z nejdůležitějších částí celého programu, je tedy nutné věnovat jí při návrhu experimentu značnou pozornost.

5.4.5 interp

Tato funkce provádí interpretaci programu, poskytuje mu potřebné vstupní hodnoty a ukládá výstupní hodnoty pro pozdější výpočet hodnoty fitness funkce. Možné návratové hodnoty této funkce naleznete v části A.1.

Kapitola 6

Experimenty

V této části je uvedeno několik úloh, pro něž je navrženo programové rozhraní. U každé úlohy je uveden cíl, použité mechanismy genetického programování, zjištěné výsledky a zhodnocení. Úloha je vždy rozdělen na několik experimentů od prvotního návrhu, až po nalezení vhodného řešení.

Během experimentů se hledá řešení dané úlohy. Toto řešení nemusí být nutně optimální, takže se v něm mohou vyskytnou sekvence příkazů, které v podstatě nic nedělají (například `+-` nebo `<>`).

6.1 Sčítání

6.1.1 Cíl

Řešením této úlohy je program, který načte ze vstupu dvě hodnoty a vypíše na výstup jejich součet.

6.1.2 Implementace

Interpret

Interpret použitý v této úloze splňuje všechny požadavky na přenositelnost uvedené v části 2.2.4. Paměťové buňky mají velikost 1 bajt, což je pro daný účel více než dostačující. Velikost paměti je 30 000 paměťových buněk. Doba běhu programu je omezena na 2000 zpracovaných instrukcí, aby nemohlo dojít k nekonečnému zacyklení. V chování interpretu byly v posledním experimentu provedeny určité změny, které jsou u tohoto experimentu uvedeny.

Trénovací množina

U této úlohy se jako trénovací množina využívá vektor vstupů a výstupů, kdy pro každý běh programu jsou náhodně generována dvě čísla sloužící jako vstupy a očekávaný výstup je vypočítám jako prostý součet těchto hodnot. Vstupní hodnoty jsou generovány v rozsahu 32 až 128, aby se předešlo případům, kdy součet nízkých čísel najde i algoritmus, který ve skutečnosti neprovádí sčítání, ale například pouze vytiskne nulu, což by například pro součet čísel 1 a 1 byl poměrně přijatelný výsledek. Horní hranicí 128 je zajištěno, aby součet nebyl větší než rozsah buňky, do které se ukládá a nedošlo k přetečení.

Generování nových jedinců

Tato část se během vývoje úlohy měnila. Jednotlivé úpravy jsou uvedeny v částech jednotlivých experimentů. Jediné, co se během experimentů neměnilo bylo použití minimální a maximální délky chromozomu, které byly ve všech experimentech striktně dodržovány.

Křížení

Křížení je v této úloze použito jednobodové, neboť délka chromozomů se pohybuje jen ve velmi malém rozsahu. Bylo by možné použít i křížení dvoubodové, avšak v tomto případě by to dle mého nepřineslo žádné výrazné zlepšení. Mechanismus křížení zůstal ve všech experimentech stejný.

Mutace

Mechanismus mutace se během experimentů měnil jen mírně, provedené změny jsou však u jednotlivých experimentů vypsány.

Fitness funkce

Přestože se fitness funkce měnila, základním kritériem pro určování hodnoty fitness byla vždy odchylka generovaného výstupu od očekávaného výstupu. Další kritéria hodnocení jsou uvedena u jednotlivých experimentů.

6.1.3 Experiment č. 1

Generování nových jedinců

Jedinci jsou generováni zcela náhodně ze všech osmi příkazů jazyka brainfuck. Jediným zásahem do chromozomu je spárování závorek, aby byl program syntakticky správný. U tohoto přístupu je zásadní nevýhodou, že nelze explicitně ovlivnit počet vstupů vyžadovaných chromozomem ani jím generovaný počet výstupů.

Mutace

V tomto experimentu je použita jednoduchá varianta mutace spočívající v nahrazení náhodně vybraného příkazu jiným příkazem. Opět je nutné provést kontrolu závorek, aby se předešlo případům, kdy je například nahrazena jedna ze závorek ohraničujících cyklus.

Fitness funkce

Ve fitness funkci se porovnává počet předpokládaných vstupů se skutečným počtem vstupů vyžadovaným programem. Dále se porovnává počet výstupů a odchylka generovaných výstupů od očekávaných výstupů. Posledním hodnotícím kritériem je návratová hodnota interpretu, díky čemuž lze znevýhodnit např. jedince, kteří se zacyklí nebo překročí hranice paměti. Jednotlivé složky fitness funkce mohou nabývat hodnoty od 0 do 100. Následně se z nich vypočítá celková hodnota fitness podle vzorce

```
fitness = 0,2 * vstupní fitness + 0,7 * výstupní fitness + 0,1 * fitness stavu
```

Tabulka 6.1: Sčítání - experiment č. 1

Max. počet generací	200
Počet jedinců	1000
Křížení	90%
Mutace	40%
Průměrná fitness hodnota nejlepšího řešení	28
Průměrný počet generací k nalezení řešení	200

Výsledky

V tomto experimentu nebylo dosaženo žádných výrazných výsledků. Správné řešení sice bylo párkrát nalezeno, ale pouze v případě, kdy populace obsahovala značně velké množství jedinců a navíc až po několika stovkách generací. Mechanismus hledání by se tedy dal považovat za poměrně náhodný. Vzhledem k tomuto neúspěchu tedy bylo nutné návrh pozměnit.

Jak je v tabulce 6.1 vidět, nebylo v žádném z provedených opakování nalezeno řešení, o čemž vypovídá hodnota průměrného počtu generací k nalezení řešení, která je v tomto případě shodná s maximálním počtem generací.

6.1.4 Experiment č. 2

Generování nových jedinců

V tomto experimentu byla provedena zcela zásadní úprava generování nových jedinců. Byl určen pevně daný začátek i konec chromozomu. Na začátku jsou příkazy pro načtení dvou vstupních hodnot a na konci příkaz pro výpis výsledku. Díky tomu je zajištěn správný počet vstupů i výstupů a není tedy nutné využívat při generování chromozomu příkazy pro vstup a výstup. Ke generování se tedy používá jen šest příkazů.

Mutace

Byly přidány dva nové typy mutací. Prvním typem je přidání nového příkazu do chromozomu. Tento typ se však aplikuje pouze za předpokladu, že délka chromozomu je menší než maximální povolená délka. Druhým typem je výměna příkazů na dvou různých pozicích. Výhodou obou těchto mutací je především to, že se pomocí nich může zvětšovat počet příkazů uzavřených v cyklu, což bylo dříve možné pouze u křížení.

Fitness funkce

Z fitness funkce byla vypuštěna část hodnotící rozdíl mezi očekávaným a skutečným počtem vstupů. To bylo umožněno změnou způsobu generování jedinců. Obdobně tomu je i u výstupů, kde se již nekontroluje jejich počet. Tím větší důležitost je však připsána hodnocení rozdílů výstupů. Nakonec byl přidán parametr hodnocení, který kontroluje, zda chromozom obsahuje cyklus. V případě, že cyklus obsahuje, je jedinci připsána vyšší hodnota fitness. Tato změna vychází z povahy jazyka brainfuck, neboť je zřejmé, že program pro sčítání musí alespoň jeden cyklus obsahovat. Fitness funkce je v tomto případě následující:

Tabulka 6.2: Sčítání - experiment č. 2

Max. počet generací	200
Počet jedinců	1000
Křížení	90%
Mutace	40%
Průměrná fitness hodnota nejlepšího řešení	42
Průměrný počet generací k nalezení řešení	192

```
fitness = 0,6 * výstupní fitness + 0,3 * fitness cyklu + 0,1 * fitness stavu
```

Výsledky

Výsledky dosažené v tomto experimentu byly uspokojivější než v experimentu předchozím. K nalezení řešení bylo oproti předchozímu experimentu třeba jen zlomku počtu jedinců v populaci. I po těchto úpravách se však stávalo, že bylo řešení nalezeno až po několika stech generací nebo dokonce vůbec.

Jak je z tabulky 6.2 patrné, bylo v některých případech nalezeno řešení, nicméně průměrný počet generací potřebných k nalezení řešení se stále velmi blíží zadané hranici.

6.1.5 Experiment č. 3

Interpret

Interpret byl upraven tak, aby se na konci kromě hodnoty aktuální buňky vypsaly i hodnoty buněk na pozicích 0 a 1. Důvod této úpravy je osvětlen v části pojednávající o změnách fitness funkce.

Fitness funkce

Jak bylo již napsáno výše, obsahuje řetězec výstupních hodnot navíc i hodnoty buněk s indexy 0 a 1. Kromě porovnání generovaného výstupu (tedy hodnoty buňky aktuální na konci programu) se s očekávaným výstupem porovnávají i hodnoty těchto buněk. Důvodem, proč se používají právě první dvě buňky je to, že hledáme nejjednodušší algoritmus sčítání. Takový algoritmus tedy zřejmě pracuje právě jen s prvními dvěma buňkami, ve kterých jsou uloženy, ale před výpisem výstupu se může posunout na buňku, která hodnotu součtu neobsahuje. Tato úprava má zcela zásadní dopad na hledání řešení, neboť zvýhodňuje jedince, kteří obsahují správný kód pro součet, vstupní hodnoty a provádí jejich destruktivní součet, ale vypisuje hodnotu nesprávné buňky. Protože však takové řešení není zcela správné, je fitness hodnota mírně snížena, aby poté byly zvýhodněny chromozomy, které již vypisují hodnotu správné buňky.

Výsledky

Díky výše uvedeným změnám bylo dosaženo velmi výrazného zlepšení. Řešení nyní bývá nalezeno poměrně brzo, i když v některých případech je k jeho nalezení třeba více generací. Jsou to však většinou případy, kdy jsou nesprávně nastaveny procentuální šance křížení a mutace a hledání se dostane do lokálního extrému. Při použití správných parametrů je

Tabulka 6.3: Sčítání - experiment č. 3

Max. počet generací	200
Počet jedinců	1000
Křížení	90%
Mutace	40%
Průměrná fitness hodnota nejlepšího řešení	98
Průměrný počet generací k nalezení řešení	63

však řešení nalezeno s poměrně vysokou pravděpodobností. Tyto výsledky lze považovat za uspokojivé a není tedy nutné pokračovat v dalším vývoji.

Z dat uvedených v tabulce 6.3 vyplývá, že výsledky tohoto experimentu jsou přijatelné, neboť řešení bylo nalezeno ve většině případů a navíc v poměrně malém počtu generací.

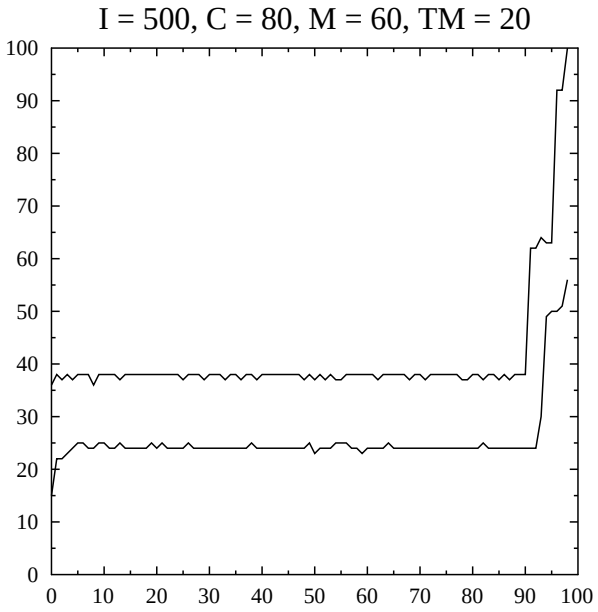
6.1.6 Shrnutí

Během tří experimentů jsem dospěl od značně náhodného hledání řešení až po řešení poměrně systematické, i když i zde hraje náhoda poměrně značnou roli. Důkazem může být například to, že správného řešení se dosáhne rychleji, když je procentuální šance mutace výrazně vyšší, než se u evolučních algoritmů běžně používá. Důvodem jsou především skokové změny fitness funkce, neboť lze rozlišit v zásadě dvě základní skupiny programů. Programy, které sčítání neprovádí a programy, které sčítání provádí, i když obsahují například odchylku o 1. Tato skoková změna byla zmírněna kontrolou prvních dvou buněk paměti zavedenou v posledním experimentu.

Dalším problémem je, že u jedinců s nízkou hodnotou fitness je tato hodnota dána především návratovou hodnotou interpretu a přítomností cyklu v programu. Neexistuje tedy žádné vodítko, kterým směrem by se měla evoluce ubírat a program může upadnout do lokálního extrému. Vysoká procentuální šance křížení a mutace tak zajistí, aby bylo nalezeno lepší řešení a evoluce se v tomto lokálním extrému nezdržela dlouho.

Podobným způsobem lze vygenerovat i program pro odčítání, jen je nutné upravit trénovací množinu. Vzhledem k tomu, že se programy na sčítání a odčítání liší jen v jednom příkazu, nepovažuji za nutné se odčítáním dále zabývat. Totéž platí i o přesunu hodnoty do jiné prázdné buňky, neboť ten probíhá jako součet přesouvaného čísla a hodnoty 0. Tento příklad zde zmiňuji proto, že je jeho výsledek použit v úloze řešící násobení.

Na následujícím grafu je uveden průběh hodnot fitness nejlepšího jedince v generaci a průměrné hodnoty fitness celé populace. V nadpisu grafu jsou uvedeny parametry experimentu, kde zkratky jsou I = počet jedinců, C = procentuální šance křížení, M = procentuální šance mutace, TM = počet členů turnaje.



Obrázek 6.1: Úloha řešící sčítání

Zde jsou nejlepší nalezená řešení této úlohy. Jak je vidět, ne vždy jsou optimální. Další výsledky lze nalézt na přiloženém CD v adresáři /vysledky.

```
,>,[-<+>]<.
,>,[-<+>]<-+-+.
,>,<>[-<+>]<-+.
```

6.2 Kopírování hodnoty buňky

6.2.1 Cíl

Cílem této úlohy je vygenerovat program, který načte ze vstupu jednu hodnotu a zkopíruje ji do následujících dvou buněk tak, že v původní buňce zůstane hodnota 0.

6.2.2 Implementace

Interpret

O interpretu platí totéž jako v předchozím experimentu. Rozdílem však je, že při volání příkazu výpisu výstupní hodnoty se vypíší hodnoty prvních tří buněk.

Trénovací množina

Trénovací množina by v podstatě mohla obsahovat jen jeden vstup a jeden výstup pro každý běh. Je však mnohem jednodušší generovat tři očekávané výstupy, protože je pak možné

Tabulka 6.4: Kopírování hodnoty buňky - experiment č. 1

Max. počet generací	200
Počet jedinců	1000
Křížení	90%
Mutace	40%
Průměrná fitness hodnota nejlepšího řešení	96
Průměrný počet generací k nalezení řešení	118

použít již vytvořený kód z předchozího experimentu. Očekávané výstupy tedy budou po řadě 0, vstup, vstup. Při generování vstupů je samozřejmě opět vhodné vyhnout se malým číslům, která by se mohla na výstupu výsledného programu objevit i jinou, než správnou cestou.

Generování nových jedinců

Noví jedinci jsou generováni stejným způsobem jako v posledním experimentu předchozí úlohy. Na levé straně je tedy pevná část programu načítající vstupní hodnotu a na pravé straně příkaz vypisující hodnoty patřičných buněk.

Křížení

Křížení je opět stejné jako v předchozím experimentu.

Mutace

Mechanismus mutace zůstal rovněž stejný jako u předchozího experimentu.

Fitness funkce

Fitness funkce je podobná jako v předchozím experimentu s tím rozdílem, že mezi kontrolovanými buňkami není aktivní buňka, ale první tři buňky.

6.2.3 Experiment č. 1

Výsledky

Vzhledem k tomu, že u této úlohy byly již v prvním experimentu využity optimalizace provedené v předchozí úloze, bylo řešení nalezeno v čase jen o něco delším. Delší čas je třeba z důvodu, že výsledný program musí obsahovat o tři příkazy více a ty musí být pochopitelně na správných pozicích. S výsledkem jsem tedy byl spokojen již po prvním experimentu a vzhledem k tomu, že tento výsledek je potřebný jen jako dílčí část další úlohy, jsem neviděl důvod provádět další optimalizace. Graf zde uvádět rovněž nebudu, protože byl velmi podobný grafu předchozí úlohy.

Z dat uvedených v tabulce 6.4 vyplývá, že výsledky tohoto experimentu jsou přijatelné, neboť řešení bylo nalezeno ve většině případů a navíc v poměrně malém počtu generací.

6.2.4 Shrnutí

Tuto úlohu lze považovat za úspěšnou a díky výsledkům v ní získaným můžeme postoupit k další, tentokrát již složitější, úloze.

Během řešení této úlohy byla nalezena například následující řešení.

```
,[->+>+<<].
,[->+>+<<]+-.
```

6.3 Násobení

6.3.1 Cíl

Cílem této úlohy je vygenerovat program, který načte ze vstupu dvě hodnoty a vypíše na výstup jejich součin.

6.3.2 Implementace

Interpret

Interpret použitý v této úloze je téměř shodný s interpretem použitým při sčítání. Byly však použity paměťové buňky o velikosti 4 bajty, aby bylo možné generovat do trénovací množiny i větší čísla a zpřesnit tak vyhodnocování fitness funkce. Další změnou je zvýšení maximálního počtu provedených instrukcí, neboť násobení vyžaduje více času.

Trénovací množina

Jak je popsáno výše, byla velikost paměťových buněk zvětšena, aby bylo možné násobit i větší čísla. Je však vhodné, aby čísla nebyla příliš vysoká, protože v takovém případě by interpretace programu trvala zbytečně dlouho. Vhodný je například rozsah činitelů od 8 do 32. Díky zvětšeným paměťovým buňkám není nutné řešit přetečení.

Generování nových jedinců

Tato část se během vývoje úlohy měnila. Jednotlivé úpravy jsou uvedeny v částech jednotlivých experimentů. Jediné, co se během řešení úlohy neměnilo bylo použití minimální a maximální délky chromozomu, které byly ve všech experimentech striktně dodržovány. Při generování chromozomu jedince je opět použita pevná levá a pravá strana pro načtení vstupů a výpis výsledku.

Křížení

Vzhledem k tomu, že v tomto případě je již délka programu výrazně vyšší, by se mohlo zdát, že je nutné použít dvoubodové křížení. Díky optimalizacím provedeným v druhém experimentu, které jsou popsány dále, to však nakonec nutné nebylo.

Mutace

Mechanismus mutace se během experimentů měnil jen mírně, provedené změny jsou však u jednotlivých experimentů vypsány.

Tabulka 6.5: Násobení - experiment č. 1

Max. počet generací	500
Počet jedinců	2000
Křížení	90%
Mutace	40%
Průměrná fitness hodnota nejlepšího řešení	40
Průměrný počet generací k nalezení řešení	500

Fitness funkce

Fitness funkce je téměř totožná s fitness funkcí u sčítání. U této úlohy však byl povolen větší rozdíl mezi porovnávanými výstupy, neboť například při nechtěném snížení hodnoty jednoho z činitelů o 1 se výsledek vygenerovaný jinak správným řešením bude lišit o hodnotu druhého činitele.

6.3.3 Experiment č. 1

Křížení

V tomto experimentu je použito opět jednobodové křížení, protože program do značné míry používá stejný kód jako sčítání. Původně jsem měl v plánu zkusit v dalším experimentu použít dvoubodové křížení, díky odlišnému přístupu generování chromozomu to však nakonec nebylo nutné.

Mutace

V tomto experimentu je použit stejný mechanismus mutace jako u sčítání.

Fitness funkce

Fitness funkce je opět téměř stejná jako u sčítání, jen je povolen větší rozdíl výstupních hodnot.

Výsledky

Výsledky tohoto experimentu byly ještě horší, než u prvního experimentu sčítání. Připisuji to především tomu, že kód programu pro násobení čísel je mnohem delší než pro sčítání, neboť kód pro sčítání je v tomto případě jen nevelkou částí celého kódu. I při použití populací o velmi vysokém počtu jedinců nebylo během stovek generací nalezeno řešení, které by se alespoň blížilo předpokládanému řešení. Bylo tedy nutné provést výrazné změny v návrhu.

Z dat uvedených v tabulce 6.5 je zcela zřejmé, že v průběhu experimentu nebylo nalezeno ani jedno řešení.

Tabulka 6.6: Násobení - experiment č. 2

Max. počet generací	500
Počet jedinců	2000
Křížení	90%
Mutace	40%
Průměrná fitness hodnota nejlepšího řešení	58
Průměrný počet generací k nalezení řešení	467

6.3.4 Experiment č. 2

Generování nových jedinců

V tomto experimentu byla provedena zcela zásadní úprava generování nových jedinců. Vyjdeme-li z povahy jazyka brainfuck, zjistíme, že násobení musí být nutně realizováno opakovaným sčítáním. Napadlo mě tedy, že by bylo vhodné použít již vygenerovaný kód pro sčítání jako nedělitelný celek. Problém však je, že toto sčítání je destruktivní a přišli bychom tedy o oba původní činitele. Jeden by zaniknul v důsledku průběžného ukládání výsledku, druhý díky tomu, že je využitý jako čítač cyklů.

Jako mnohem vhodnější se tedy jeví použití kódu pro kopírování hodnoty buňky, které plní funkci sčítání a navíc původní hodnotu zkopíruje do další buňky. V kombinaci s kódem pro přesun hodnoty mezi buňkami vygenerovaným rovněž pomocí genetického programování tedy dostáváme blok kódu, který provádí sčítání beze změny původního sčítance, což je pro násobení naprosto vhodné. Při generování jedince i následných evolučních mechanismech je tento blok kódu nahrazen znakem, který se liší od znaků znázorňujících běžné příkazy jazyka brainfuck. Před vstupem do interpretu je tento znak nahrazen příslušným blokem kódu a interpret tedy nemusí zpracovávat nový znak.

Výsledky

V tomto experimentu byly výsledky již poměrně uspokojivé. Řešení již bylo nalezeno, ovšem zdaleka ne vždy a opět jen při větším počtu jedinců v populaci a po mnoha generacích. Rozhodnul jsem se tedy proces ještě poněkud optimalizovat.

Z dat uvedených v tabulce 6.6 je zcela zřejmé, že v průběhu experimentu sice bylo několikrát nalezeno řešení, avšak až po mnoha generacích.

6.3.5 Etapa č. 3

Mutace

Při zkoumání výsledků předchozího experimentu jsem došel k závěru, že je nutné určitým způsobem změnit mechanismus mutace.

První změnou bylo přidání nového typu mutace, který měl vyřešit problém, kdy jinak správný program obsahuje v hlavním cyklu některé znaky navíc. Spoléhat se na to, že tento znak bude odstraněn během křížení by bylo poměrně naivní. Tento nový typ mutace tedy odstraní z chromozomu jeden znak, avšak jen v případě, že je délka chromozomu větší než minimální délka.

Tabulka 6.7: Násobení - experiment č. 3

Max. počet generací	500
Počet jedinců	2000
Křížení	90%
Mutace	40%
Průměrná fitness hodnota nejlepšího řešení	91
Průměrný počet generací k nalezení řešení	227

Další problém, který bylo nutné vyřešit, byl nedostatečně dlouhý hlavní cyklus. Délka tohoto cyklu se pochopitelně mění během křížení, ale když se mění i během mutace, je evoluce programu mnohem rychlejší. Tento typ mutace tedy najde pravou závorku cyklu a posune ji vpravo. Samozřejmě by bylo možné posunovat i levou závorku vlevo, ale ukázalo se, že posouvání jedné závorky je dostačující.

Posledním problémem, který jsem z výsledků předchozího experimentu identifikoval byla nesprávná pozice znaku zastupujícího blok kódu pro nedestruktivní sčítání. Pokud je totiž tento znak umístěn na nesprávném místě v jinak vhodném kódu, jsou takto dosažené výsledky poměrně špatné. Tato mutace tedy vyhledá znak zastupující blok kódu a posune jej na náhodně vygenerovanou pozici.

Výsledky

S výsledky tohoto experimentu jsem byl již spokojen, neboť řešení bylo nalezeno poměrně brzy a nebylo nutné, aby populace obsahovala tolik jedinců jako v předchozím experimentu. Doba hledání řešení je delší než u sčítání, to je však dáno vyšší složitostí násobení.

Z dat uvedených v tabulce 6.7 je zcela zřejmé, že nyní je již řešení nalezeno poměrně často a v rozumném počtu generací.

6.3.6 Shrnutí

Tato úloha prokázal, že je v jazyce brainfuck možné pomocí genetického programování vygenerovat i složitější aritmetické operace. Generování takových operací však pochopitelně vyžaduje vhodné omezení prohledávaného prostoru, jak bylo ukázáno v jednotlivých experimentech.

Graf průběhu fitness hodnot zde uvádět nebudu, neboť je téměř shodný s grafem sčítání. Jediným rozdílem je, že hledání řešení trvá více generací, ale graf opět obsahuje pouze dvě skokové změny.

Zde jsou uvedeny některé nalezené výsledky.

```
,>,<[>[->+>+<<]>>[-<<+>>]<<<-]>>.
,>,<[>+-[->+>+<<]>>[-<<+>>]<<<-]>>.
```

6.4 Stezka ze Santa Fe

6.4.1 Cíl

Cílem této úlohy je vygenerovat program, který řeší problém typu Santa Fe.

6.4.2 Popis problému

Tento problém se používá poměrně často pro prezentaci evolučních technik. Základem je dvourozměrné pole o určité velikosti (v tomto případě 32 x 32). Na tomto poli je umístěno několik kousků potravy (zde 89). Na toto pole je pak umístěn mravenec řízený programem, jehož úkolem je najít v omezeném čase co nejvíce kousků potravy.

6.4.3 Implementace

Prostředí

Jak již bylo napsáno výše, mravenec se v tomto případě pohybuje po poli o vlikosti 32 x 32, na kterém je umístěno 89 kousků potravy. Zbývá tedy vyřešit už jen, co se stane, když se mravenec pokusí překročit hranice pole. Zvolil jsem řešení, kdy je pole kulové, takže když se mravenec pokusí překročit hranice např. na pravém konci řádku, přesune se na lévý konec téhož řádku. Totéž platí i o pohybu směrem nahoru a dolů.

Mravenec má omezenou sadu činností. Může jít vpřed, otočit se vpravo či vlevo nebo se podívat, zda je na poli před ním jídlo.

Interpret

Interpret bylo v tomto případě pochopitelně nutné upravit, neboť musí uchovávat a měnit aktuální polohu mravence, směr, kterým je natočen, a stav potravy (které kousky jsou již zkonzumovány). Další změnou je, že příkaz pro vstup slouží jako indikátor, zda se před mravencem nachází potrava. Pokud ne, uloží do aktuální buňky hodnotu 0, pokud ano, uloží hodnotu 1. Poslední úpravou je, že doba trvání programu již není omezena jen počtem vykonaných instrukcí, ale i počtem pohybů mravence. Jako pohyb se počítá posun vpřed a otočení vpravo nebo vlevo.

Trénovací množina

Trénovací množina v tomto případě obsahuje jen jedno pole s náhodně rozmístěnou potravou.

Generování nových jedinců

Způsob generování nových jedinců probíhá ve všech experimentech náhodně, avšak liší se reprezentace jedinců. Ve všech experimentech však platí, že je vygenerována v podstatě jen část chování jedince, která se poté uzavře do nekonečné smyčky, neboť program, který by popisoval všech 400 pohybů, by byl příliš dlouhý a výpočet by byl značně náročný na systémové prostředky.

Křížení

Křížení se během experimentů vývoje měnilo. Podrobnosti jsou opět rozepsány v jednotlivých experimentech.

Mutace

Pro mutaci platí totéž, co pro křížení.

Tabulka 6.8: Santa Fe - experiment č. 1

Max. počet generací	50
Počet jedinců	500
Křížení	90%
Mutace	5%
Průměrná fitness hodnota nejlepšího řešení	9

Fitness funkce

Fitness funkce je zde naprosto jednoduchá, neboť hodnotu fitness určuje počet zkonzumovaných kousků potravy.

6.4.4 Experiment č. 1

Generování nových jedinců

Noví jedinci jsou generování jako řetězce příkazů, stejně jako v předchozích úlohách. Po vygenerování chromozomu je opět volána funkce na úpravu závorek.

Křížení

Křížení probíhá opět jednobodově, protože je využit kód z předchozích úloh. Dvoubodové křížení jsem ani nezaváděl, protože jsem již měl v úmyslu zcela zásadně změnit návrh.

Mutace

V tomto experimentu je použit stejný mechanismus mutace jako u násobení.

Výsledky

Mravenec v tomto případě během 400 pohybů většinou neposbírá více než 10 z 89 kousků potravy. Tato hodnota není příliš uspokojivá a je tedy nutné provést další vylepšení. Mnohem lepší je však skutečnost, že oproti předchozím úlohám je zde dobře vidět postupný vývoj populace. Je tomu tak právě díky tomu, že zde nedochází k velkým skokovým změnám fitness funkce.

Z dat uvedených v tabulce 6.8 je zcela zřejmé, že není ani zdaleka nalezeno optimální řešení, neboť při použití stejných parametrů se u tohoto problému dosahuje po 50 generací většinou řešení, které má fitness hodnotu vyšší než 40.

6.4.5 Experiment č. 2

Interpret

Při procházení výsledků předchozího experimentu jsem dospěl k závěru, že některá poměrně kvalitní řešení nejsou vybrána kvůli překročením rozsahu paměti směrem vlevo. Rozhodnul jsem se tedy změnit interpret tak, aby se paměťové buňky nacházely i vlevo od počátečního ukazatele. Nejjednodušším způsobem, jak toho dosáhnout bylo umístit na začátku interpretu ukazatel doprostřed pole paměťových buněk.

Tabulka 6.9: Santa Fe - experiment č. 2

Max. počet generací	50
Počet jedinců	500
Křížení	90%
Mutace	5%
Průměrná fitness hodnota nejlepšího řešení	11

Výsledky

Výsledky jsou lepší, než v předchozím experimentu, avšak jen zanedbatelně. Vzhledem k tomu, že toto vylepšení příliš nepomohlo, jsem se rozhodnul provést zásadní změny.

Jak je v tabulce 6.9 vidět, mírně se zvýšila průměrná fitness hodnota nejlepšího řešení, avšak stále ne dostatečně.

6.4.6 Experiment č. 3

Generování nových jedinců

Vzhledem k nepříznivým výsledkům předchozích experimentů jsem se rozhodnul změnit radikálně reprezentaci jedince. Konkrétně přejít od lineární reprezentace na reprezentaci stromovou V této stromové reprezentaci obsahuje jedinec místo řetězce sloužícího jako chromozom odkaz na kořen stromu znázorňujícího chromozom. Struktura tohoto stromu je popsána v A.4.

Této změně jsem již od začátku přisuzoval velký význam, neboť se jedná o podobnou změnu jako vkládání nedělitelného bloku kódu u násobení. Pro jednotlivé příkazy mravence jsem tedy vytvořil odpovídající úseky kódu v jazyce brainfuck. Další výhodou přístupu je také to, že program takto pracuje jen s jednou paměťovou buňkou.

Křížení

Při křížení si dva jedinci vzájemně vymění větve ze stromů reprezentujících jejich chromozom.

Mutace

Při mutaci se vymění dvě větve v rámci jednoho jedince. U této výměny je však třeba kontrolovat, zda jedna z větví není podmnožinou větve druhé, neboť by tak vzniknul neplatný strom.

Výsledky

Výsledky tohoto experimentu jsou mnohem lepší než v experimentech předchozích. Mravenec nyní najde ve většině případů více než polovinu potravy. Pokud potravu nerozmístíme náhodně, ale například tak, aby tvořila souvislou cestu, nalezne se bez problémů i řešení, které najde veškerou potravu.

Jak je vidět v tabulce 6.10, je v tomto případě již průměrná hodnota fitness nejlepšího řešení natolik vysoká, že je srovnatelná s obdobnými experimenty řešícími tento problém.

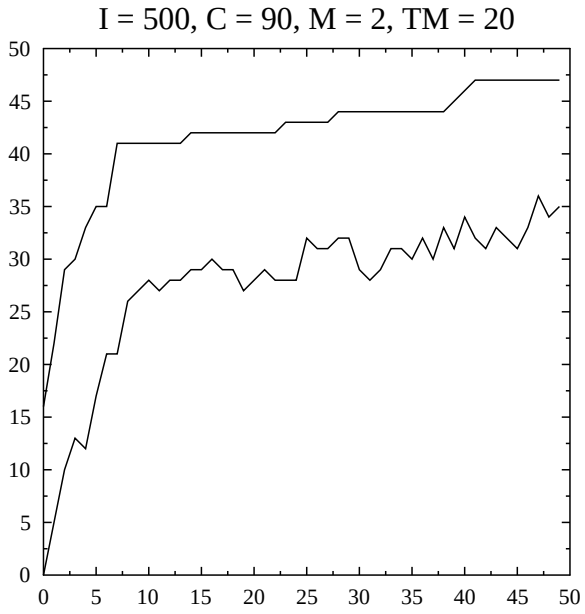
Tabulka 6.10: Santa Fe - experiment č. 3

Max. počet generací	50
Počet jedinců	500
Křížení	90%
Mutace	5%
Průměrná fitness hodnota nejlepšího řešení	45

6.4.7 Shrnutí

V tomto experimentu bylo dokázáno, že v jazyku brainfuck se dají řešit klasické úlohy pro genetické programování s výsledky, které jsou srovnatelné s použitím jiných jazyků. Kvůli syntaxi jazyka brainfuck je však nutné výrazně redukovat prohledávaný prostor.

Na následujícím grafu je znázorněn opět průběh fitness hodnot jako v předchozích úlohách. Značení v nadpisu je rovněž stejné.



Obrázek 6.2: Úloha řešící sčítání

Kapitola 7

Závěr

V této práci jsem dokázal, že jazyk brainfuck lze využít ke genetickému programování při dosažení poměrně kvalitních výsledků. Zároveň se však ukázalo, že k dosažení takových výsledků jsou nutné velké zásahy ze strany programátora ve formě optimalizací omezujících prohledávaný prostor. Pro složitější aplikace genetického programování je tedy dle mého lepší zvolit jiný jazyk.

Nevýhody jazyka brainfuck pro genetické programování se projevily především v experimentech generujících aritmetické operace. Je tomu tak především proto, že tento jazyk nepoužívá pojmenované proměnné, a v chromozomu se tedy mohou jednotlivé proměnné snadno zaměnit, což vede k nežádoucím výsledkům. Tento problém by se dal pochopitelně kompenzovat používáním různých bloků kódu, které by kopírovaly proměnné uložené na pevně daných místech do jiných umístění a naopak. Tím by se dosáhlo jistého ekvivalentu pojmenování proměnných. Tento přístup mi však přijde zbytečně komplikovaný.

Dalším důvodem, proč je jazyk brainfuck nevhodný pro genetické programování je to, že i drobná změna kódu, například jen jediného příkazu, může změnit program natolik, že bude provádět zcela jinou činnost. To odporuje předpokladu evolučních algoritmů, dle kterého by měli mít jedinci, kteří se liší jen minimálně podobnou hodnotu fitness. V tomto případě tomu tak však zdaleka není.

Další možnosti pokračování v tomto tématu vidím především ve vytvoření bloků kódu, které by umožňovaly vytvářet i složitější algoritmy. Spojením takových bloků a stromové reprezentace chromozomu by mohlo být možné dosáhnout výsledků například i u symbolické regrese apod.

Druhou možností by bylo porovnat jazyk brainfuck s dalšími ezoterickými jazyky a pomocí různých experimentů určit jazyky vhodné pro konkrétní aplikace.

Literatura

[1] Vladimír Mařík. *Umělá inteligence (1-4)*. Academia, 2003. ISBN 80-200-1044-0.

[2] WWW stránky. Brainfuck - esolang.
`http://esoteric.voxelperfect.net/wiki/Brainfuck`.

[3] WWW stránky. Brainfuck - wikipedia, the free encyclopedia.
`http://en.wikipedia.org/wiki/Brainfuck`.

[4] WWW stránky. Evolution strategy - wikipedia, the free encyclopedia.
`http://en.wikipedia.org/wiki/Evolution_strategy`.

[5] WWW stránky. Evolutionary algorithm - wikipedia, the free encyclopedia.
`http://en.wikipedia.org/wiki/Evolutionary_algorithm`.

[6] WWW stránky. Evolutionary programming - wikipedia, the free encyclopedia.
`http://en.wikipedia.org/wiki/Evolutionary_programming`.

[7] WWW stránky. Genetic algorithm - wikipedia, the free encyclopedia.
`http://en.wikipedia.org/wiki/Genetic_algorithm`.

[8] WWW stránky. Genetic programming - wikipedia, the free encyclopedia.
`http://en.wikipedia.org/wiki/Genetic_programming`.

[9] WWW stránky. humancompetitive.
`http://www.genetic-programming.com/humancompetitive.html`.

[10] Irena Váňová. Úvod do genetických algoritmů.
`http://vanova.org/texty/geneticke_algoritmy.pdf`.

Dodatek A

Implementace

Programová část je implementována v jazyce C++. V následující částech budou uvedeny používané třídy spolu s popisem jejích základních funkcí a rovněž použité mechanismy selekce, křížení a mutace.

A.1 Výčtový typ INTERP_RET

```
typedef enum
{
  OK = 0,
  OUT_OF_BOUNDS = 1,
  MAX_ITER = 2,
  SYNTAX_ERR = 3,
} INTERP_RET;
```

Tento výčtový typ slouží k uložení návratové hodnoty z interpretu. Význam jednotlivých položek výčtového typu je následující:

OK	Program byl interpretem vykonán bez problémů.
OUT_OF_BOUNDS	Kód programu způsobil posun ukazatele mimo hranice paměti.
MAX_ITER	Bylo dosaženo maximálního počtu iterací interpretu.
SYNTAX_ERR	V kódu programu byla nalezena syntaktická chyba.

Hodnotu SYNTAX_ERR interpret vůbec nevrací, neboť jsou do něj předávány jen syntakticky správné programy. V tomto výčtovém typu je spíše pro případnou změnu návrhu, kdy by se předávaly i syntakticky nesprávné programy.

A.2 Struktura inOut

```
typedef struct inOut
{
string inString;
string outString;
} inOut;
```

Do této struktury se ukládá řetězec vstupních znaků spolu s odpovídajícím řetězcem očekávaných výstupních znaků.

A.3 Třída insOuts

```
class insOuts
{
private:
vector<inOut> inOutArr;
public:
insOuts(unsigned int experCount);
string *getExpOut(unsigned int pos);
string *getIns(unsigned int pos);
unsigned int getCount();
};
```

Tato třída obsahuje vektor struktur inOut (viz A.2). Tento vektor obsahuje vstupy a očekávané výstupy pro jednotlivé běhy programu.

insOuts	Slouží k naplnění trénovací množiny. Parametr experCount udává počet generovaných položek vektoru, který je shodný s počtem samostatných běhů programu.
getExpOut	Vrací ukazatel na řetězec očekávaných výstupních znaků. Parametr pos určuje pozici ve vektoru vstupně/výstupních hodnot.
getIns	Vrací ukazatel na řetězec vstupních znaků.
getCount	Vrátí počet položek vektoru trénovacích řetězců.

Funkci insOuts je možné přepsat podle potřeb aktuálního experimentu. Bylo by pochopitelně možné, a z programátorského hlediska čistší, aby parametrem této funkce byl odkaz na funkci generující tento vektor, avšak cílem této práce není vytvoření obecných tříd pro libovolnou implementaci, ale jen pro konkrétní případ.

A.4 Třída treeNode

```
class treeNode
{
private:
  treeNode *parent;
  treeNode *left;
  treeNode *middle;
  treeNode *right;
  char value;
public:
  treeNode(treeNode *parentNode, unsigned int maxDepth);
  treeNode(treeNode *parentNode);
  treeNode *getParent();
  treeNode *getLeft();
  treeNode *getMiddle();
  treeNode *getRight();
  char getValue();
```

```
  treeNode *getTop();
  void setParent(treeNode *newParent);
  void setLeft(treeNode *newLeft);
  void setMiddle(treeNode *newMiddle);
  void setRight(treeNode *newRight);
  void setValue(char newValue);
  unsigned int getDepth();
  void getString(string &nodeChrom);
  void print();
  void allNodes(vector<treeNode *> *treeNodes);
  void switchChild(treeNode *oldChild, treeNode *newChild);
  treeNode * copyTree(treeNode *parentNode);
  bool isParent(treeNode *secondNode);
};
```

Tato třída implementuje uzel stromu reprezentujícího chromozom programu.

parent	Odkaz na rodičovský uzel. U kořenového uzlu má hodnotu NULL.
left	Levý potomek. U listových uzlů má hodnotu NULL.
middle	Prostřední potomek
right	Pravý potomek
value	Hodnota uzlu. Může nabývat hodnot L (otočení vlevo), R (otočení vpravo), M (posun dopředu), I (pokud je před mravencem jídlo), 2 (blok dvou pohybů) nebo 3 (blok tří pohybů).
getParent	Vrací ukazatel na rodiče. Funkce getLeft, getMiddle a getRight jsou podobné.
getValue	Vrací hodnotu uzlu.
getTop	Vrací ukazatel na kořenový uzel.
getDepth	Vrací hloubku uzlu v rámci stromu.
getString	Vrací chromozom větve s kořenem v tomto uzlu v řetězci předaném do funkce.
print	Tiskne obsah větve s kořenem v tomto uzlu.
allNodes	Naplní vektor všemi uzly větve s kořenem v tomto uzlu.
switchChild	Nahradí potomka tohoto uzlu jiným potomkem. Využívá se u křížení a mutace.
copyTree	Vytvoří kopii větve s kořenem v tomto uzlu a jako rodiče nastaví uzel určený parametrem parentNode.
isParent	Zjistí, zda je uzel předaný jako parametr rodičem (ne nutně přímým) tohoto uzlu. Tato funkce je velmi důležitá kvůli zachování správné struktury stromu při mutaci.

Funkce, jejichž názvy začínají na set jsou odbobami funkcí začínajících na get, jen slouží k nastavení

A.5 Třída individual

Třída individual reprezentuje jedince v populaci.

```
class individual
{
private:
  string chromosome;
  unsigned int fitness;
public:
  individual(unsigned int minLength, unsigned int maxLength);
  void setChromosome(string &newChromosome);
  string getChromosome();
  void setFitness(unsigned int newFitness);
  unsigned int getFitness();
  void execute(insOuts &trainSet);
  void print();
  void fixBrack(unsigned int fixedLeft, unsigned int maxLength);
};
```

chromosome	Řetězec, který obsahuje kód jedince.
fitness	Celočíselná fitness hodnota jedince
individual	Konstruktor třídy. Parametry minLength a maxLength určují minimální, resp. maximální délku programu jedince.
setChromosome	Nastavení chromozomu (kódu programu) jedince
getChromosome	Vrací chromozom jedince.
setFitness	Nastavení fitness hodnoty
getFitness	Vrací fitness hodnotu.
execute	Spustí chromozom v interpretu a následně jedinci přiřadí fitness hodnotu. Parametr trainSet obsahuje vstupy a výstupy jednotlivých běhů programu (viz A.3).
print	Vytiskne chromozom a fitness hodnotu.
fixBrack	Upraví kód programu tak, aby byly všechny závorky spárované. Parametr fixedLeft určuje počet neměnných znaků na levé straně chromozomu.

A.6 Třída generation

Třída generation zastupuje generaci jedinců.

```
class generation
{
private:
  vector<individual> individuals;
public:
  generation(%dodělat
bool firstGen, unsigned int indivCount, unsigned int minLength,
             unsigned int maxLength);
  void evalIndividuals(insOuts &trainSet);
  individual * selectIndiv(unsigned int tourMembers);
  void pushIndiv(individual &newIndiv);
```

```
  unsigned int getIndivCount();
  void crossOver(unsigned int crossPercent, unsigned int fixedLeft,
                 unsigned int fixedRight, unsigned int maxLength);
  void mutation(unsigned int mutatePercent, unsigned int fixedLeft,
                unsigned int fixedRight, unsigned int maxLength);
  void printStat();
  individual & getBestIndiv();
};
```

individuals	Vektor jedinců, kteří patří do dané generace.
generation	Konstruktor generace. Parametr indivCount určuje počet jedinců v generaci. Hodnoty minLength a maxLength udávají minimální a maximální délku chromozomu jedince.
evalIndividuals	Odešle do interpretu postupně všechny jedince v generaci a následně je vyhodnotí pomocí fitness funkce.
selectIndiv	Vybere jedince pomocí turnajové metody. Parametr tourMembers určuje počet účastníků turnaje.
pushIndiv	Přidá jedince do vektoru jedinců aktuální generace.
getIndivCount	Vrací počet jedinců v generaci.
crossOver	Provede křížení jedinců v generaci. Parametr crossPercent určuje procentuální šanci křížení. Ostatní parametry mají stejný význam jako v předchozích případech.
mutation	Provede mutaci jedinců v generaci. Parametry mají stejný význam jako u křížení.
printStat	Vytiskne průměrnou fitness hodnotu celé generace a nejlepšího jedince.
getBestIndiv	Vrací jedince v generaci, který má podle fitness funkce nejvyšší ohodnocení.

A.7 Třída experiment

Třída generation zastupuje generaci jedinců.

```
class experiment
{
private:
  unsigned int chMaxLength;
  generation *actGen;
  generation *nextGen;
  unsigned int mutation;
  unsigned int cross;
  unsigned int tourMembers;
  insOuts *trainSet;
  void nextGener();
public:
```

```
  experiment(unsigned int indivCount, unsigned int trainSets,
             unsigned int minLength, unsigned int maxLength,
             unsigned int initCross, unsigned int initMutation,
             unsigned int initTourMembers);
  void start();
};
```

chMaxLength	Maximální délka chromozomu.
actGen	Ukazatel na aktuální generaci
nextGen	Ukazatel na následující generaci
mutation	Procentuální pravděpodobnost mutace
cross	Procentuální pravděpodobnost křížení
tourMembers	Počet účastníku turnaje při selekci
trainSet	Ukazatel na trénovací množinu pro experiment (viz A.3).
nextGener	Pomocí mechanismů selekce, křížení a mutace vytvoří následující generaci.
experiment	Konstruktor experimentu. Význam parametrů je stejný jako v předchozích případech.
start	Spustí experiment.

A.8 Funkce generateChromosome

```
void generateChromosome(string &chromosome, unsigned int minLength,
                        unsigned int maxLength);
```

Tato funkce slouží ke generování chromozomů. Je volána z konstruktoru třídy individual (viz A.5). Tuto funkci lze tedy přepsat tak, aby byla pro řešený problém co nejvhodnější. Například je možné určit příkazy na levé a pravé straně používané k načtení a výpisu hodnot, které budou obsahovat všechny chromozomy a omezit tak prohledávaný prostor.

A.8.1 Parametry

chromosome	Chromozom jedince
minLength	Minimální délka chromozomu
maxLength	Maximální délka chromozomu

A.9 Funkce fixBrackets

V této funkci se chromozom upravuje tak, aby byly všechny závorky spárovány. Chromozom obsahující levé nebo pravé hranaté závorky bez příslušných protějšků totiž není syntakticky správný a není tedy platným chromozomem.

A.9.1 Parametry

```
void fixBrackets(string &chromosome, unsigned int fixedLeft,
                 unsigned int maxLength);
```

chromosome	Chromozom jedince
fixedLeft	Počet pevných znaků na levé straně jedince
maxLength	Maximální délka chromozomu

A.10 Funkce expInsOuts

Účelem této funkce je vygenerování vstupů a očekávaných výstupů pro jeden běh programu. Opakovaným voláním této funkce lze tedy vytvořit celou trénovací množinu uloženou v objektu třídy insOuts (viz A.3).

```
void expInsOuts(string &ins, string &outs);
```

A.10.1 Parametry

ins	Řetězec vstupů
outs	Řetězec výstupů

A.11 Funkce evalFitness

Tato funkce počítá fitness hodnotu jedince. Díky značnému počtu vstupních parametrů je možné tuto hodnotu počítat podle různých kritérií.

```
int evalFitness(string &genOut, string &expOut, int inputsDiff,
                INTERP_RET status, string &chromosome);
```

A.11.1 Parametry

genOut	Řetězec obsahující výstup generovaný programem jedince
expOut	Očekávaný výstup
inputsDiff	Rozdíl mezi počtem vstupů vyžadovaných programem jedince a skutečným počtem vstupů
status	Návratový stav interpretu
chromosome	Chromozom jedince

A.12 Funkce interp

Tato funkce provádí interpretaci programu, poskytuje mu potřebné vstupní hodnoty a ukládá výstupní hodnoty pro pozdější výpočet hodnoty fitness funkce. Možné návratové hodnoty této funkce naleznete v části A.1.

```
INTERP_RET interp(string &prog, string &inarr, string &genOut, int *inDiff);
```

A.12.1 Parametry

prog	Řetězec obsahující program, který se má interpretovat.
inarr	Řetězec vstupních znaků
genOut	Řetězec pro uložení výstupních znaků
inDiff	Proměnná pro uložení rozdílu požadovaných vstupů a skutečných vstupů