VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

GRAMATICKÁ EVOLUCE V OPTIMALIZACI SOFTWARE

GRAMMATICAL EVOLUTION IN SOFTWARE OPTIMIZATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. ZDENĚK PEČÍNKA**
AUTHOR

VEDOUCÍ PRÁCE **Prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
SUPERVISOR

BRNO 2017

Abstrakt

Tato diplomová práce nabízí stručný úvod do evolučního počítání. Popisuje a porovnává genetické programování a gramatickou evoluci a jejich možné využití v problematice automatické opravy software. Podrobně studuje možnosti aplikace gramatické evoluce v problému automatické opravy softwaru. Na základě získaných poznatků byla navržena a implementována nová metoda pro automatickou opravu softwaru, založená na gramatické evoluci. Její experimentální ověření proběhlo na řadě testovacích programů.

Abstract

This master's thesis offers a brief introduction to evolutionary computation. It describes and compares the genetic programming and grammar based genetic programming and their potential use in automatic software repair. It studies possible applications of grammar based genetic programming on automatic software repair. Grammar based genetic programming is then used in design and implementation of a new method for automatic software repair. Experimental evaluation of the implemented automatic repair was performed on set of test programs.

Klíčová slova

evoluční počítání, genetické programování, gramatická evoluce, soft-computing, gramatika, softwarové inženýrství, oprava softwaru, lokalizace chyby, gcc, Clang, Python, symbolická regrese, umělá inteligence

Keywords

evolutionary computing, genetic programming, grammar based genetic programming, soft-computing, grammar, software engineering, software repair, fault localization, gcc, Clang, Python, symbolic regression, artificial intelligence

Citace

PEČÍNKA, Zdeněk. *Gramatická evoluce v optimalizaci software*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Sekanina Lukáš.

Gramatická evoluce v optimalizaci software

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Lukáše Sekaniny, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Zdeněk Pečínka
23. května 2017

Poděkování

Tímto bych chtěl poděkovat mému vedoucímu prof. Ing. Lukáši Sekaninovi, Ph.D., za cenné rady a konzultace, které mi pomohly s dokončením práce.

Obsah

Kapitola 1

Úvod

Význam automatické optimalizace a opravy softwaru neustále narůstá. Kvůli vysokým finančním a časovým nákladům se nevyplatí ruční ladění a testování velkých aplikací. Rostoucí složitost programů a nemožnost formalizovat řešené problémy nám navíc nedovoluje použít tradiční deterministické algoritmy, je tedy nutné vymyslet efektivnější postup. Díky své robustnosti se nabízí využití metod inspirovaných přírodou. Evoluční algoritmy byly již v minulosti použity v řadě složitých problémů a často dosáhly velice dobrých výsledků.

Cílem této práce je analyzovat, jaké výhody nabízí evoluční počítání v dané problematice. Na základě získaných poznatků potom navrhnout, implementovat a otestovat metodu pro automatickou opravu sémantických chyb v jednoduchém zdrojovém kódu, který bude využívat gramatickou evoluci.

Druhá kapitola poslouží jako stručný úvod do problematiky evolučního počítání. V jejím rámci nastíníme, jak probíhá výpočet pomocí evolučního algoritmu a popíšeme některé z nejčastěji používaných variant a jejich vlastnosti.

V třetí kapitole se zmíníme o hlavních důvodech, proč se vlastně automatickou opravou softwaru zabývat. Připomeneme některé z tradičních přístupů k opravě chyb a nakonec popíšeme některé z úspěchů, jakých bylo na tomto poli dosaženo využitím metod založených na evolučním počítání.

Čtvrtá kapitola popisuje základní principy automatické opravy softwaru využívající gramatické evoluce. Důležitou součástí je porovnání vlastností opravy využívající gramatické evoluce s lineárním chromozomem, oproti automatické opravě implementované aplikací GenProg. Ta je hlavní inspirací této práce, ačkoliv je postavená na genetickém programování se stromově strukturovaným chromozomem.

Pátá kapitola prezentuje strukturu aplikace. Zabývá se volbou implementačních nástrojů a detailně popisuje implementaci součástí automatické opravy pomocí gramatické evoluce.

Šestá kapitola ukáže kroky provedené pro ověření správné funkce implementované GE. Popíše programy použité pro testování funkčnosti automatické opravy a statistické vyhodnocení jim příslušejících testů. V poslední části nabídne srovnání dosažených výsledků oproti výsledkům dosažených jinými existujícími metodami.

Kapitola 2

Evoluční počítání

Algoritmy založené na evolučním počítání můžeme popsat jako netradiční optimalizační metody používané pro řešení obtížných úkolů. Od tradičních přístupů se liší zejména využíváním nepřesnosti a náhodnosti při hledání optima. Jejich výpočty jsou charakteristické svou robustností a díky tomu, že pracují s tzv. populací kandidátních řešení, je méně pravděpodobné, že uváznou v lokálním optimu. Jak vyplývá z názvu kapitoly, je hlavní inspirací těchto algoritmů biologická evoluce, ze které převzaly zejména základní myšlenky diverzifikace a selekce.

2.1 Biologická evoluce

Biologická evoluce je samovolný dlouhodobý proces, jehož působení můžeme pozorovat jako diverzifikaci a vývoj života na Zemi. Její vliv můžeme pozorovat jak na úrovni druhů, tak na úrovni organismů. Princip biologické evoluce vysvětlují evoluční teorie jako je například Darwinova evoluční teorie nebo teorie neodarwinismu. Podle [14, 9] platí, že:

1. Na všechny organismy působí selekční tlak, ten vzniká kombinací všech vnějších faktorů (např. množství potravy v okolí, schopnost zaujmout sexuálního partnera, přírodní katastrofy, apod.).
2. Jedinci, kteří jsou vůči selekčnímu tlaku lépe přizpůsobeni, mají vyšší šanci se reprodukovat a přenést tak svoje geny do další generace.
3. Při reprodukci dochází ke smíšení genetické informace dvou jedinců a vznikají tak nové organismy, které v sobě kombinují vlastnosti svých rodičů. Tento proces jinak nazýváme křížení.
4. Díky křížení a náhodným genetickým mutacím mohou vzniknout úplně nové znaky jedinců.

Můžeme říct, že biologická evoluce může být chápána jako forma optimalizačního procesu. Řešeným problémem je zde nalezení nejlépe přizpůsobeného organismu, vzhledem k podmínkám daným selekčním tlakem.

Zdá se, že biologická evoluce nemá žádný cíl. Pokud ji využijeme jako zdroj inspirace pro vytvoření optimalizačního algoritmu, je nejprve potřeba definovat cíl. V praxi hledáme extrém tzv. účelové funkce. Lépe přizpůsobení jedinci se pak rozmnožují častěji, než ti ostatní. Díky tomu můžeme předpokládat, že bude průměrná kvalita jedinců v populaci během času stoupat a výpočet tak bude směrovat k žádoucímu řešení.

2.2 Základní pojmy evolučního počítání

Význam některých pojmů používaných v evolučním počítání se může lišit od jejich významu v biologické evoluci. Pro pořádek tedy základní pojmy nadefinujeme [14, 16]:

gen je základní jednotka genetické informace, v evolučním počítání bývá často reprezentován číselnou hodnotou, jak v binární tak dekadické podobě, výsledný tvar závisí na konkrétních požadavcích daného algoritmu

alela je konkrétní hodnota jednoho genu

genotyp je způsob zakódování kandidátního řešení. Nejčastěji se můžeme setkat s tím, že geny jsou uspořádané lineárně jako řetězec, nebo ve stromové struktuře. V rámci evolučních algoritmů se termín genotyp často zaměňuje s označením chromozom

fenotyp je výsledné řešení získané dekódováním chromozomu jedince

jedinec je označení pro jedno kandidátní řešení, definované svým genotypem

populace je množina všech jedinců v daném prostoru

generace je populace v konkrétním časovém okamžiku, resp. iteraci výpočtu

fitness funkce je označení pro účelovou funkci, reprezentující "prostředí, ve kterém se jedinci vyvíjejí". Funkce vrací číselnou hodnotu vyjadřující úroveň přizpůsobenosti jedince vzhledem k "prostředí", tuto hodnotu nazýváme **fitness hodnota**.

selekce je algoritmus výběru jedinců určených pro reprodukci

selekční tlak v biologické evoluci označuje souhrnný vliv všech vnějších faktorů, v rámci evolučního počítání popisuje, jak rychle evoluční algoritmus konverguje

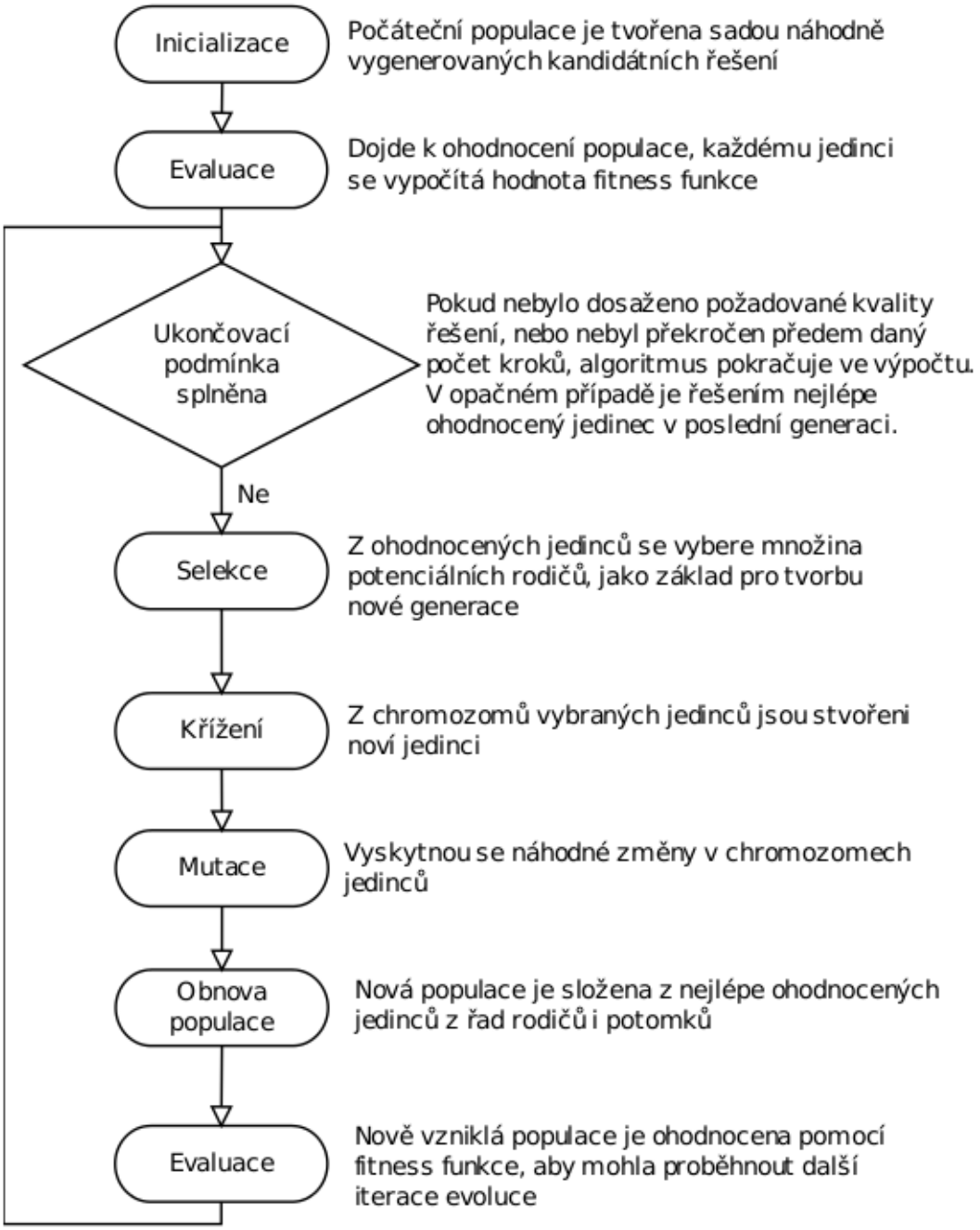
křížení je postup vzniku nových jedinců-potomků, díky kombinaci genotypů dvou nebo více vybraných rodičů

mutace je označení pro náhodnou změnu genotypu, která nastává s velice malou pravděpodobností

2.3 Princip evolučního výpočtu

Pro praktické využití evoluce v optimalizačních úlohách je nutné vyřešit řadu problémů. Jedná se zejména o potřebu vyvinout vhodnou reprezentaci řešeného úkolu a navrhnout snadno vyčíslitelnou fitness funkci, tak aby bylo možné evoluční výpočet efektivně použít. Vzhledem ke komplexnosti tohoto postupu se automaticky předpokládá jeho využití zejména v obtížně řešitelných úlohách. Každá varianta evolučního algoritmu řeší jednotlivé problémy odlišně a udržuje různou úroveň podobnosti s principy biologické evoluce. Základní princip je ale pokaždé stejný [9]. Zobecněná podoba evolučního algoritmu je znázorněna v obrázku 2.1.

- Algoritmus pracuje nad populací jedinců. Velikost populace je většinou pevně daná a dodržuje se ve všech generacích.



Obrázek 2.1: Evoluční algoritmus

- Každý jedinec je charakterizován svým chromozomem a kóduje v sobě jedno z kandidátních řešení problému.
- Prvním krokem je vytvoření počáteční populace. Ta obsahuje pevně daný počet jedinců, jejichž chromozomy jsou vygenerovány náhodně nebo vhodnou heuristikou.
- Populace je ohodnocena pomocí fitness funkce. Každému jedinci je přiřazena číselná hodnota, která říká, do jaké míry je přizpůsoben, vzhledem k řešení daného problému. Často bývá fitness funkce definována jako velikost chyby kandidátního řešení oproti

tomu referenčnímu. Potom je nejlepší možná fitness, jaké lze dosáhnout, rovna nule. Pokud některý jedinec dosáhne takového ohodnocení, algoritmus můžeme ukončit, protože jsme nalezli řešení úlohy.

- Výpočet končí, pokud bylo nalezeno optimální řešení. Ne vždy ale můžeme takový výsledek zaručit, proto existuje druhá podmínka, která omezuje maximální počet iterací nebo dobu výpočtu. Dokud ani jedna z ukončovacích podmínek není splněna, pak se opakovaně provádí následující kroky:

  1. Selekce zajistí výběr rodičů určených pro křížení. Výběr je závislý na ohodnocení jedinců, čím lepší je, tím vyšší je šance na výběr. Selekce tak generuje selekční tlak, který dlohodobě vynucuje zlepšování populace jako celku. Selekční tlak je možné, a často i výhodné, různě modifikovat. Místo proporciální selekce, ve které je šance na reprodukci přímo úměrná hodnotě fitness funkce, můžeme použít normalizovanou fitness. Normalizovaná fitness je užitečným nástrojem, který umožňuje snadno ladit diverzitu a rychlost konvergence výpočtu.
  2. Nad dvěma nebo více vybranými rodiči je provedena operace křížení. Vznikají tak noví potomci, jejichž chromozom je kombinací genotypu rodičů. Selekce a křížení se opakuje tak dlouho, než je dosaženo požadované velikosti nové populace.
  3. Populace je vystavena vlivu náhodné mutace. S nízkou pravděpodobností dojde k náhodným změnám genů v chromozomech jedinců. Tento mechanismus zajišťuje vyšší diverzitu prohledávaných řešení. Další výhodou je možnost vzniku úplně nového genetického materiálu.
  4. Dojde k sestavení nové generace jedinců. Existuje více přístupů jak tento proces implementovat. Obecně můžeme říct, že řešení se pohybuje mezi dvěma extrémy. První z nich nastává v situaci, když je nová generace složena pouze z potomků a všichni jedinci z řad rodičů jsou odstraněni. V druhém případě je nová populace tvořena rodiči i potomky.
  5. Nová populace je znovu ohodnocena stejným způsobem, jako tomu bylo ve fázi inicializace, a pokud není splněna některá z ukončovacích podmínek, tak algoritmus pokračuje další iterací výpočtu.

2.4 Genetické programování

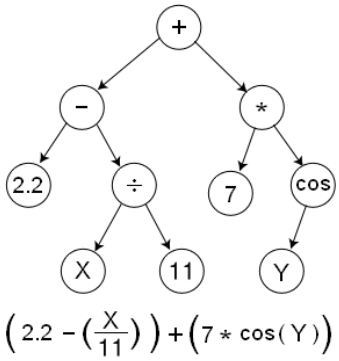
Genetické programování je metoda strojového učení založená na evolučním počítání. Z názvu můžeme odvodit hlavní cíl této metody, kterým je automatická tvorba a modifikace programů. Je vhodné se nad touto metodou pozastavit, protože hlavní inspirací této práce je právě publikace [17], která GP aplikuje v problematice opravování chyb.

Genetické programování pracuje, stejně jako ostatní evoluční algoritmy, s populací jedinců. Dále je nutné definovat následující pojmy [15]:

- **Množina terminálů**, složená z proměnných, konstant a funkcí bez argumentů s vedlejším účinkem.
- **Množina funkcí**, která může obsahovat aritmetické a logické funkce a řadu konstrukcí z programovacích jazyků. V GP je nutné použít tzv. chráněné varianty některých funkcí, ty ošetřují případy, že by funkce nebyla definována pro některou kombinaci vstupů. Příkladem můžou být základní aritmetické nebo logické operace.

- **Způsob výpočtu fitness** - účelová funkce souvisí s řešeným problémem a její hodnota by měla reflektovat úroveň přizpůsobenosti jedince vzhledem k danému problému
- **Parametry GP** jako je například velikost populace, počet generací, apod.
- **Způsob ukončení výpočtu** - například nalezení jedince, u kterého je suma chyb oproti trénovací množině menší než předem daný práh.

Každý jedinec reprezentuje počítačový program, zakódovaný nejčastěji ve stromové struktuře. Uzly stromu nabývají hodnot z dané množiny funkcí. Listy obsahují zase prvky z množiny terminálů. Ukázku programu můžeme vidět na obrázku 2.2.



Obrázek 2.2: Příklad reprezentace programu pomocí stromu [18].

Každého jedince je možné evaluovat vykonáním programu, který reprezentuje, na zadaných datech. Vzhledem ke stromové reprezentaci je GP obecně nejvhodnější pro logické nebo funkcionální jazyky. Hodnota fitness funkce bývá tradičně určena podle odchylky výstupů programu jedince od očekávaných výstupů, vzhledem k předem dané trénovací množině.

Stromová struktura s sebou přináší jeden zásadní problém - tím je implementace křížení a mutace. Bohužel není možné použít klasické operátory používané u lineární reprezentace.

Obrázek 2.3 znázorňuje jednu z možných variant křížení ve stromové struktuře. V každém z rodičovských stromů je náhodně zvolen podstrom, který je následně nahrazen podstromem z druhého rodiče.

Mutace obvykle probíhá tak, že se v mutovaném jedinci určí podstrom, který je posléze nahrazen náhodně vygenerovaným podstromem, viz 2.4.

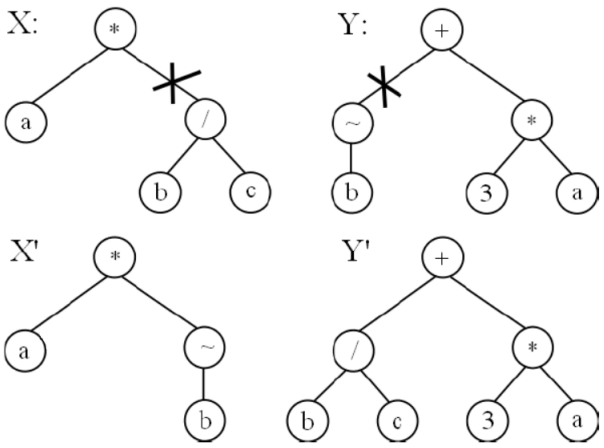
2.5 Gramatická evoluce

Gramatická evoluce je jednou z variant evolučního algoritmu. Tato metoda do problematiky zavádí novou úroveň abstrakce, protože omezuje prostor možných řešení pomocí formální gramatiky. Systémy založené na GE mají kořeny zejména v dříve používaném GP a poprvé se objevily kolem roku 1990 [10]. V této sekci popíšeme základní principy GE.

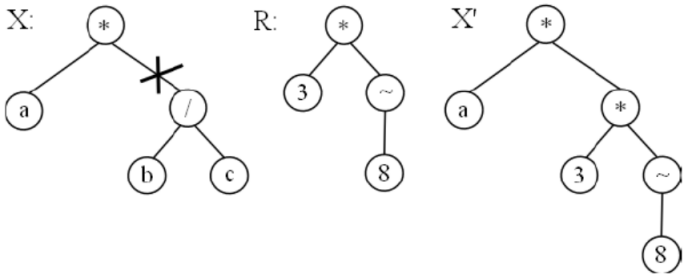
2.5.1 Kontextová gramatika

V informatice termín formální gramatika označuje strukturu, která popisuje formální jazyk, resp. jej dokáže generovat. Tato práce bude zaměřena na opravu zdrojového kódu v podmnožině jazyka C, jehož gramatika je kontextová.

Obecně gramatiku můžeme definovat následovně[19]:



Obrázek 2.3: Ukázka křížení v GP [16].



Obrázek 2.4: Ukázka mutace v GP [16].

$$G = (N,\Sigma,P,S)$$

Gramatika G je čtveřice, kde:

- N je množina neterminálních symbolů
- Σ je množina terminálních symbolů
- P je množina přepisovacích pravidel
- S je počáteční neterminální symbol z množiny N

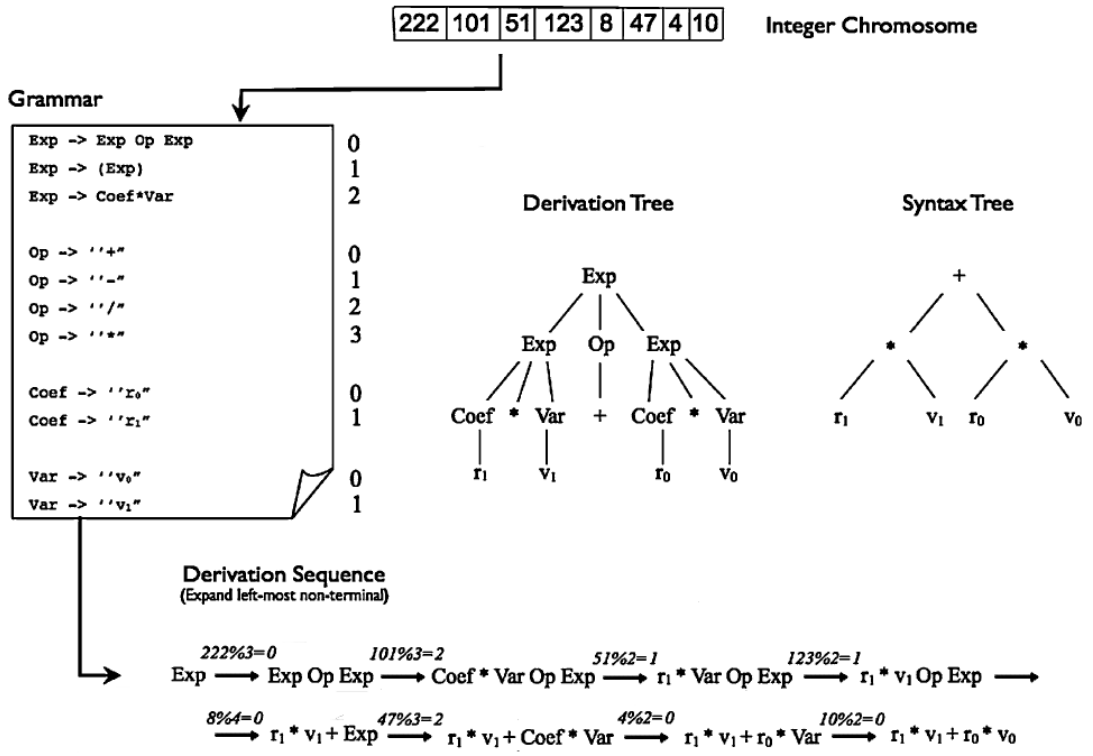
Pokud je gramatika kontextová, potom jsou všechna její pravidla z množiny P ve tvaru:

$$\alpha \mathrm{A} \beta \rightarrow \alpha \gamma \beta, \text{ kde } \mathrm{A} \in \mathrm{N},\ \alpha, \beta \in (\Sigma \cup \mathrm{N})^*,\ \gamma \in (\Sigma \cup \mathrm{N})^+$$
$$\text{nebo } \mathrm{S} \rightarrow \varepsilon, \text{ pakliže se S neobjevuje na pravé straně žádného pravidla}$$

2.5.2 Reprezentace jedince

Každý jedinec je definován řetězcem tvořeným z celočíselných hodnot v intervalu $[0, n-1]$, kde n odpovídá maximálnímu počtu přepisovacích pravidel, použitelných pro jeden neterminální symbol gramatiky. Z praktických důvodů se ale v programové implementaci často využívá pevně dané hodnoty 255, protože n obecně nedosahuje vysokých hodnot.

Chromozom je tedy lineární, GE ale umožňuje vytvořit jedince, jehož fenotyp je strukturován jako strom, podobně jako tomu je v GP. Na obrázku 2.5 můžeme vidět, jak mapování lineárního chromozomu na stromově strukturovaný fenotyp funguje.



Obrázek 2.5: Tvorba stromově strukturovaného fenotypu z lineárního chromozomu s využitím gramatiky [10].

Na začátku je nutné označit přepisovací pravidla, asociovaná ke každému neterminálu, číslem z intervalu $[0, m-1]$, kde m je celkový počet pravidel vycházejících z daného neterminálu. V průběhu algoritmu zpracováváme chromozom zleva. Aktuálně používanou hodnotu genu označíme x. Princip tvorby fenotypu (viz obrázek 2.5) potom můžeme popsat následovně:

1. Rozvíjenou větnou formu inicializujeme tak, že bude obsahovat pouze počáteční neterminál určený v definici gramatiky.
2. Nalezneme nejlevější neterminál větné formy a rozvineme jej pomocí přepisovacího pravidla s číslem $x \ \% \ m$.[1]
3. Pokud větná forma obsahuje pouze terminální symboly, ukončíme výpočet. Jinak pokračujeme bodem 2.

Při výpočtu je kvůli redundantnímu kódování chromozomu využívána operace modulo. To umožňuje použití gramatiky, ve které mají neterminály různý počet přepisovacích pravidel.

Jednou z žádaných vlastností genotypu je pevně daná délka. V rámci GE ale tuto vlastnost nelze jednoduše garantovat, je nutné nejprve vyřešit několik problémů.

[1] % značí operaci modulo

Pokud dojde k vygenerování fenotypu, zatímco chromozom ještě nebyl zpracován, jsou nepoužité geny chápány jako nekódující sekvence a jsou jednoduše ignorovány. Podobně tomu je i v DNA, kde jsou úseky zvané *introny* záměrně odstraňovány.

Hlavní komplikací je ale to, že při použití chromozomu pevné délky je vysoká šance vygenerování invalidního fenotypu[2]. Z toho důvodu je do postupu zavedeno nové pravidlo. Pokud fenotyp není validní a algoritmus došel na konec chromozomu, aplikuje se pravidlo zvané *wrap* [10]. To v podstatě říká, aby algoritmus začal chromozom zpracovávat opět od začátku.

Tento postup minimalizuje výskyt invalidních fenotypů. Přináší s sebou ovšem novou komplikaci. Opakované zpracování chromozomu může způsobit nekonečný růst fenotypu, což pochopitelně není vítané. Tento problém je ovšem snadno řešitelný. Pro operaci *wrap* stačí zavést maximální počet opakování k. Pokud ani po k přečteních chromozomu nebude vygenerován validní jedinec, tak k němu bude přiřazena hodnota fitness, která minimalizuje pravděpodobnost jeho reprodukce.

2.5.3 Genetické operátory

Díky tvaru chromozomu je možné využívat standardní genetické operátory pracující nad řetězci, používané i v jiných variantách evolučních algoritmů. Přestože aplikace křížení a mutace je prakticky stejná, dochází u GE k jistým odlišnostem ve výsledné podobě fenotypu. Ty vznikají z toho důvodu, že fenotyp je strukturován ve stromě narozdíl od genotypu.

Křížení

Standardní křížení pracuje nad dvěma chromozomy. Existuje více možností, jak jej implementovat. Nejvíce používanou verzí je pravděpodobně jednobodové křížení. Jeho princip je znázorněn v příkladu 2.6.

Rodičovské chromozomy

48	16	120	38	51	230	79	17	84	63	49	122	165	213
56	80	71	168	214	147	31	3	91	112	67	135	8	29

↓

Chromozomy vzniklé křížením

48	16	120	38	214	147	31	3	91	112	67	135	8	29
56	80	71	168	51	230	79	17	84	63	49	122	165	213

Obrázek 2.6: Ukázka jednobodového křížení nad lineárním chromozomem.

Bod křížení se volí náhodně. V příkladu 2.6 je jeho pozice rovna čtyřem. Z rodičů A a B vzniknou dva noví potomci, $A´$ a $B´$. Jedinec $A´$(resp. $B´$) potom má první čtyři alely shodné s rodičem A (resp. B) a zbytek chromozomu zase s rodičem B (resp. A).

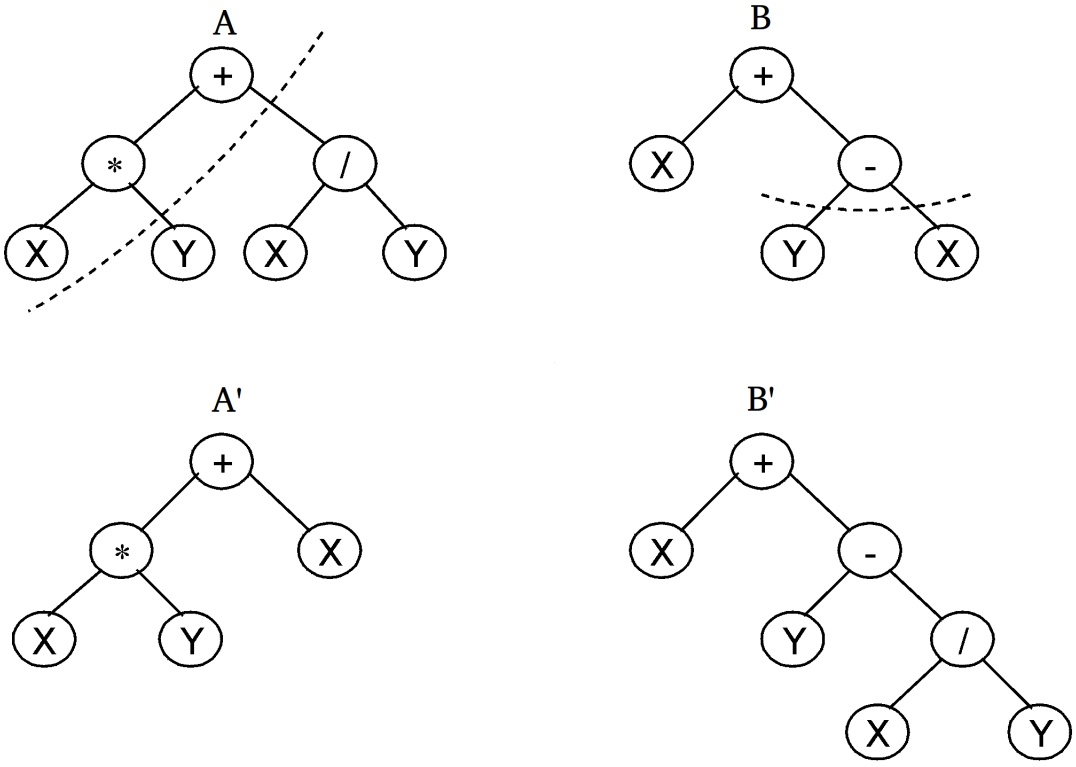
[2]Validní fenotyp obsahuje pouze terminální symboly.

Pro znázornění vlivu křížení na fenotyp zavedeme následující přepisovací pravidla 2.1:

Tabulka 2.1: Přepisovací pravidla pro znázornění vlivu genetických operátorů na fenotyp

expr	→	op expr expr	(0)
	→	var	(1)
op	→	+	(0)
	→	-	(1)
	→	*	(2)
	→	/	(3)
var	→	X	(0)
	→	Y	(1)

Na obrázku 2.7 můžeme vidět, jaký účinek na fenotyp má křížení z ukázky 2.6, s použitím pravidel z tabulky 2.1.



Obrázek 2.7: Vliv operace křížení na fenotyp v GE [6].

Je patrné, že vliv křížení v GE má mnohem větší dopad, než je tomu například v GP (viz 2.3). Kvůli reprezentaci gramatikou může jednobodová změna chromozomu způsobit vícenásobné změny v fenotypu.

Ukázalo se, že tento způsob křížení zajišťuje velice efektivní mixování stavebních bloků, a zároveň má velkou generativní schopnost. Díky tomu se GE stala účinnou metodou, která mnohdy může předčit klasicky využívané postupy, jako je třeba GP.

Mutace

Mutace v lineárním chromozomu pracuje tak, že na náhodně vybraném genu dojde k náhodné změně hodnoty.

Původní chromozom

					↓								
48	16	120	38	51	230	79	17	84	63	49	122	165	213

Zmutovaný chromozom

					↓								
48	16	120	38	51	29	79	17	84	63	49	122	165	213

Změna jedné pozice v chromozomu může zapůsobit na více míst cílového fenotypu. V případě změny v prvním genu může vzniknout naprosto odlišný jedinec. Mutace má na formování fenotypu velký vliv a je vhodné udržet pravděpodobnost jejího výskytu co nejnižší.

Na druhou stranu princip redundantního kódování genů sám o sobě zajišťuje, že mutace často bude neutrální povahy. Předpokládejme, že mutace modifukuje gen s hodnotou *8* na hodnotu *16*, a aktuálně zpracovávaný neterminál je možné rozgenerovat právě dvěma způsoby. Potom se výsledné přepisovací pravidlo vybere pomocí *8 % 2* resp. *16 % 2*, je zřejmé, že v obou případech zvítězí pravidlo s číslem *0*. Popsaná mutace tedy je neutrální.

2.6 Nastavení parametrů

Jak již bylo zmíněno, evoluční algoritmy jsou spíše heuristického charakteru a jejich kvalita je z velké části dána vhodným nastavením parametrů.

Mezi základní parametry EA patří:

- **Velikost populace**, která je většinou pevně daná. Výpočet na úrovni populace umožňuje souběžný rozvoj více potenciálních řešení, nemluvě o případné paralelizaci výpočtu.
- **Počet generací** určuje počet iterací evolučního výpočtu. Počet generací slouží zejména k omezení délky výpočtu, který by jinak teoreticky mohl probíhat nekonečně dlouho.
- **Metoda selekce** může přímo ovlivňovat selekční tlak. Ten potom podává informaci o očekávané rychlosti konvergence daného EA. Vyšší selekční tlak může urychlit nalezení řešení, ale také zvyšuje pravděpodobnost předčasné konvergence k lokálnímu optimu. Nižší selekční tlak naopak prohledávání prostoru zpomaluje, ve snaze nalézt globální optimum.
- **Způsob obnovy populace** - příkladem může být například pevně zvolený poměr počtu rodičů a potomků v nové populaci.

2.7 Vyhodnocení experimentů

Evoluční algoritmy při výpočtu využívají náhodně vygenerovaných hodnot řídících parametrů. Vzhledem k tomu je pro získání relevantních a statisticky významných výsledků nutné provést řadu běhů.

Sám princip EA je založený na iterativním počítání a postupném přibližování k hledanému řešení. Vzhledem k nutnosti opakovaně provádět výpočet můžeme očekávat, že EA budou relativně časově náročné. Můžeme předpokládat, že řešení problému využitím EA bude všeobecně pomalejší, než případný deterministický přístup.

Výsledné řešení bude obsaženo v populaci po posledním kroku evoluce. Nejkvalitnější řešení bude reprezentováno jedincem dané populace, jehož fitness hodnota převyšuje fitness hodnoty ostatních.

Kapitola 3

Automatická oprava softwaru

Složitost nových aplikací je stále větší a s tím se zvyšují i náklady na údržbu. Úměrně tomu narůstá důležitost automatické opravy softwaru. Tradiční metody jsou často stavěné pouze jako podpůrné systémy opravy a stále vyžadují konstruktivní přínos programátora. Plně automatizovaná oprava chyb má tedy potenciál značně přispět k rozvoji softwarového inženýrství.

V současnosti dochází k rychlému rozvoji experimentálních výpočetních metod, jako jsou například evoluční algoritmy. Jejich využití často umožňuje řešení značně komplikovaných problémů. Dalo by se říct, že aplikace těchto moderních postupů na automatickou opravu chyb se vyloženě nabízí.

3.1 Motivace v číslech

V obrázku 3.1 je možné vyčíst poměrné náklady na jednotlivé fáze života softwaru.

Automatická oprava softwaru může být aplikována, pokud existuje možnost vzniku chyb ve zdrojovém kódu. Z toho můžeme odvodit, že automatická oprava softwaru má vliv zejména na fáze implementace, testování a údržby. Je patrné, že zmíněné fáze zabírají více než 80% nákladů vynaložených na celý životní cyklus softwaru (viz 3.1). Z hlediska šetření nákladů je tedy v automatické opravě softwaru velký potenciál.

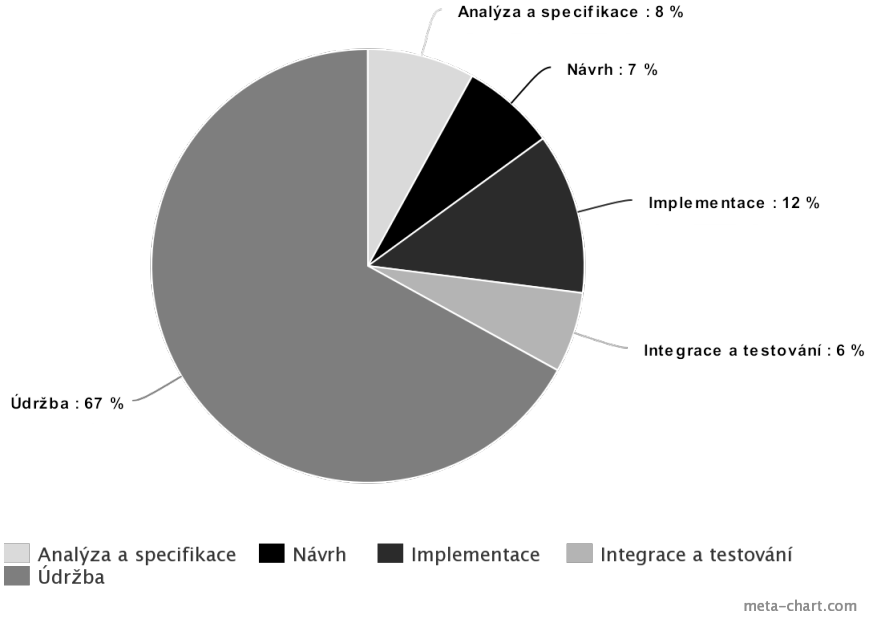
Logickým krokem je tedy právě snaha automatizovat opravu softwaru. Díky složitosti problému se jako vhodné jeví právě použití evolučních algoritmů.

3.2 Konvenční oprava softwaru

Tradiční metody opravy softwaru jsou často zaměřené spíše na usnadnění odhalení chyby a následné ruční provedení opravy programátorem, než na její plně automatizované vygenerování.

Existuje řada nástrojů umožňující *profiling* nebo ladění programů založených na statické nebo dynamické analýze zdrojového kódu. Jako příklad můžeme uvést *Valgrind* nebo *GNU Debugger*. S jejich pomocí je odhalení chyb popř. optimalizace kódu mnohem efektivnější. Ačkoliv mohou být tyto nástroje značně nápomocné, hlavní práci stále musí odvést programátor.

Kvůli složitosti zdrojových kódů je algoritmické nalezení opravy často nemožné. Jedná se tedy o problém, který je vhodné řešit experimentálními metodami, jako jsou například evoluční algoritmy.



Obrázek 3.1: Poměrné náklady spojené s jednotlivými životními fázemi softwaru. Hodnoty převzaty z přednášek předmětu IUS [5].

3.3 Automatická oprava softwaru

Pro automatické generování opravy softwaru byla navržena řada odlišných metod. Uvedeme několik příkladů, jak je možné k její realizaci přistupovat. Údaje v této sekci jsou převzaty z [11].

Aplikace pro automatickou opravu může klasicky pracovat nad zdrojovým nebo binárním kódem. Nás zajímá především oprava prováděná nad zdrojovým kódem programu. Taková oprava často využívá výhod AST – abstraktního syntaktického stromu, který slouží k reprezentaci abstraktní syntaktické struktury zdrojového kódu.

Prvním krokem automatické opravy je automatické generování modifikovaných verzí programů. Je možné jej realizovat na více úrovních:

- **Modifikace příkazů** – je možné využít tzv. operátory opravy. Jedná se o předem definované úseky kódu, které je možné vložit do původního programu a docílit tak změny chování. Příkladem může být vložení podmínky nebo příkazu přiřazení. Stejně tak je možné existující příkazy odstranit nebo změnit jejich pořadí. Tento přístup je možné implementovat na úrovni AST nebo zdrojového kódu.

- **Modifikace uzlů AST** – tento přístup pracuje čistě nad AST a umožňuje vznik jemnějších modifikací, jako je například změna datového typu nebo operátoru.

- **Modifikace zakódovaného AST** – za účelem změny generování modifikací je možné stromovou strukturu AST zakódovat do jiné podoby. Způsob generování modifikací je dán právě typem zmíněného kódování. Tato práce využívá právě tohoto principu a jeho podrobný popis je v následujících kapitolách.

V okamžiku, kdy aplikace dokáže generovat modifikované verze původního programu, je nutné mezi nimi najít tu, které bude reprezentovat výslednou opravu. K tomuto kroku se často používají experimentální metody jako jsou evoluční algoritmy.

3.4 Příklad automatické opravy softwaru

Jako ukázku funkčnosti automatické opravy software uvedeme příklad z textu [17]. Tato oprava byla vygenerována pomocí aplikace GenProg založené na využití genetického programování. Aplikace GenProg je hlavní inspirací této práce, princip její opravy je nastíněn v sekci 4.3.

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365) {
4        if (isLeapYear (year) ){
5        if (days > 366) {
6          days -= 366;
7          year += 1;
8        }
9        else {
10       }
11    }
12      else {
13        days -= 365;
14        year += 1;
15      }
16   }
17   printf("the year is \%d\n", year);
18 }
```

Obrázek 3.2: Úsek zdrojového kódu s chybou [17]

V obrázku 3.2 je popsán program, který by měl v závislosti na parametru *days* vypsat jaký bude rok po uběhnutí přesně *days* dní od roku 1980. Tabulka 3.1 potom obsahuje testovací případy.

	Input	Output
1	1000	1982
2	2000	1985
3	3000	1988
4	4000	1990
5	5000	1993
6	366	1980
7	10593	2008

Tabulka 3.1: Množina testovacích případů. Případy 1-5 jsou pozitivní. 6 a 7 negativní [17].

Při vykonávání testovacích případů 6 a 7 dojde k nekončné smyčce na řádcích 3,4,8 a 11. Docházelo k chybě při zpracování posledního dne přechodného roku. S pomocí aplikace GenProg [17] bylo nalezeno následující korektní řešení 3.3.

Kromě opravy samotné je důležité si povšimnout, jakým způsobem byla provedena. Je zřejmé, že aplikace GenProg dodržuje základní princip minimalizace opravy - většina kódu zůstala nezměněna a ve výsledném řešení je jasně vyznačeno, které části byly modifikovány.

```
1  void zunebug_repair (int days) {
2    int year = 1980;
3    while (days > 365) {
4      if (isLeapYear (year) ) {
5        if (days > 366) {
6          // days -= 366;//deleted
7          year += 1;
8        }
9      else {
10     }
11     days -= 366;//inserted
12     } else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf ("the year is \%d\n", year);
18 }
```

Obrázek 3.3: Opravený úsek kódu s značkami o změně [17]

Pomocí GP autoři práce dosáhli kvalitních výsledků při automatizovaném opravování chyb v projektech, jejichž zdrojové kódy obsahovaly mnohdy až desetitisíce řádků. V obrázku 3.4 jsou zobrazeny informace o některých opravách automatizovaně provedených aplikací GenProg.

Obrázek 3.4 obsahuje základní informace o testovaném programu jako je popis funkce nebo počet řádků zdrojového kódu. Nejzajímavější částí jsou údaje o typu opravené chyby, době hledání a výsledném rozsahu opravy.

Program	Lines of Code	Weighted Path	Description	Fault	Time (s)	Fitness Evals	Repair Size
gcd	22	1.3	Euclid's algorithm	Infinite loop	153	45.0	2
zune	28	2.9	MS Zune excerpt	Infinite loop	42	203.5	4
uniq utx 4.3	1146	81.5	Duplicate filtering	Segmentation fault	34	15.5	4
look utx 4.3	1169	213.0	Dictionary lookup	Segmentation fault	45	20.1	11
look svr 4.0	1363	32.4	Dictionary lookup	Infinite loop	55	13.5	3
units svr 4.0	1504	2159.7	Metric conversion	Segmentation fault	109	61.7	4
deroff utx 4.3	2236	251.4	Document processing	Segmentation fault	131	28.6	3
nullhttpd 0.5.0	5575	768.5	Webserver	Heap buffer overrun	578	95.1	5
indent 1.9.1	9906	1435.9	Source code formatting	Infinite loop	546	108.6	2
flex 2.5.4a	18775	3836.6	Lexical analyzer generator	Segmentation fault	230	39.4	3
atris 1.0.6	21553	34.0	Graphical tetris game	Stack buffer overrun	80	20.2	3
Total	63277	8817.2			2003	651.2	39

Obrázek 3.4: Ukázka oprav uskutečněných aplikací GenProg [17].

Kapitola 4

Gramatická evoluce a oprava softwaru

V této kapitole si popíšeme, jak je možné postupovat při opravě softwaru s využitím GE. Nastíníme jednotlivé činnosti, které je nutné implementovat pro dosažení automatické opravy softwaru. Dále se budeme zabývat rolí gramatická evoluce. Nabízí se srovnání principu opravy pomocí GE oproti opravě využívající GP, protože právě pomocí porovnávání jsme schopni nejsnáze odhalit, v čem je řešení problémů pomocí GE specifické oproti jiným přístupům.

4.1 Popis problému

Předpokládaným vstupem je soubor se zdrojovým kódem obsahující chybu. Samotná oprava bude zaměřena na hledání chyb v rámci vybrané funkce.

Základní myšlenka automatické opravy je následující:

- Vstupem aplikace je zdrojový kód, obsahující chyby.
- Pro činnost algoritmu je nutná sada testovacích případů, které jsou rozdělené do dvou skupin:
  - Pozitivní testovací případy, s jejichž pomocí je možné konstruovat požadovanou funkci.
  - Negativní testovací případy reprezentující takové vstupní hodnoty, při kterých program vrací nesprávné výsledky. Jejich použití umožňuje pomocí dynamické analýzy programu lokalizovat části kódu, které jsou potenciálně zodpovědné za chybné chování.
- Výstupem je potom zdrojový kód, ve kterém byly opraveny všechny chyby.

4.2 Princip opravy

Princip opravy můžeme v obecnosti popsat pomocí následujících kroků:

1. Proto, aby mohl systém automatické opravy s vstupním programem pracovat, je nutné napřed jeho zdrojový kód převést do podoby abstraktního syntaktického stromu(AST).

S ním pak můžeme pracovat podobně jako s derivačním stromem, který popisuje gramatikou vygenerovanou větu.

2. Nyní můžeme využít množinu testovacích případů, a s její pomocí odhalit části programu, které jsou pravděpodobně zodpovědné za chybné chování. Tento krok je důležitý hlavně ze dvou důvodů:
   - Čím menší úsek programu opravujeme, tím se typicky stává problém jednodušší z pohledu automatické opravy.
   - Vzhledem k snaze o zachování čitelnosti zdrojového kódu je žádoucí, aby oblast automatizovaně vygenerované změny byla co nejmenší.
3. Poté, co izolujeme úseky zdrojového kódu určené k opravě, můžeme vynegerovat počáteční chromozom. Zkoumaný podstrom AST v podstatě reprezentuje fenotyp jedince. GE ale pracuje s lineárním genotypem, je tedy nutné nejprve fenotyp v podobě podstromu AST zakódovat pomocí sekvence genů.
4. V okamžiku, kdy získáme genotyp počátečního jedince reprezentujícího opravovanou část původního programu, můžeme inicializovat počáteční generaci vygenerováním jedinců s genotypem podobným genotypu počátečního jedince.
5. Nyní můžeme zahájit samotnou opravu, využívající gramatickou evoluci. Jejím cílem je nalezení nového podstromu AST, který vrací správné výsledky pro všechny testovací případy.

4.3 Lokalizace chyb

Pro odhalení sekcí zdrojového kódu, které mohou obsahovat chyby, je nejprve nutné provést dynamickou analýzu programu s využitím sady testovacích případů. Výstupem je posloupnost příkazů, které byly vykonány v běhu, počas kterého došlo k chybě.

V článku *Automatic Program Repair with Evolutionary Computation* [17] byla prezentována elegantní metoda vyžívající tzv. váhovaných cest. Každý jedinec byl reprezentován následující dvojicí:

- AST obsahující všechny příkazy programu.
- Váhovanou cestu, tvořenou posloupností dvojic (s,w_s), kde

  s označuje příkaz

  $\mathbf{w}_s$ označuje váhu daného příkazu

  Váha příkazu je potom rovna:

  0, pokud byl navštíven pouze v rámci vykonávání pozitivního případu,

  1, pokud byl příkaz navštíven pouze při vykonávání negativního testovacího případu,

  1/10, pokud byl příkaz navštíven při vykonávání obou druhů testovacích případů.

Takto váhované cesty potom modifikovaly pravděpodobnosti výběru uzlů AST pro aplikování genetických operátorů. Díky tomu došlo k efektivnímu omezení prohledávaného prostoru a současně zachování zbytku AST, který by jinak bylo nutné opakovaně sestavovat kvůli výpočtu fitness hodnoty.

Další nezanedbatelnou výhodou byla možnost snadno a libovolně upravit váhy příkazů tak, aby se prohledávací prostor rozšířil resp. zúžil, podle konkrétních požadavků dané aplikace. Takový algoritmus je umožněn díky tomu, že genetické programování pracuje s stromovým chromozomem, který je možné přímo namapovat na AST.

Nevýhodou této metody je, že je možné ji využít pouze v případě, že automatická oprava interně reprezentuje chromozom v podobě stromu.

4.4 Lokalizace chyb při automatické opravě pomocí gramatické evoluce

Postup popsaný v předchozí kapitole bohužel nemůžeme použít, jestliže chceme pro automatickou opravu využívat gramatickou evoluci. GE interně pracuje nad lineárním chromozomem, jehož vliv na stromově strukturovaný fenotyp není snadné předpovědět.

Pokud je chromozom reprezentován v podobě stromu, může evoluce probíhat nad celým stromem, od kořene po listové uzly. Lokalizace spočívá jednoduše v tom, že nikdy nebudeme modifikovat uzly, které nebyly navštíveny během negativního testovacího případu. Můžeme říct, že tato implementace lokalizace využívá principu vylučovací metody.

Oproti tomu lineární chromozom tento přístup neumožňuje, protože není možné limitovat vliv genetických operátorů na jednotlivé uzly výsledného AST. Je tedy nutné znát konkrétní uzel, ve kterém se nachází chyba, ještě před spuštěním evoluce a poté modifikovat pouze podstrom daného uzlu. Implementace takové lokalizace je komplikovaná, pravděpodobně by musela využívat heuristického přístupu.

Proto, aby bylo dosaženo plně automatizované opravy chyb, je tedy nutné nejprve implementovat kvalitní lokalizaci. V tomto ohledu se GP využívající chromozomu v podobě stromu jeví jako výhodnější metoda.

4.5 Oprava chyb

Za opravu chyb je zodpovědná gramatická evoluce. Výpočet pracuje s podstromem AST, který byl izolován ve fázi lokalizace chyby. Výpočet probíhá obdobně, jako tomu je u obecného popisu evolučního algoritmu 2.3 a je založen na principu symbolické regrese.

1. Počáteční populace může být inicializována zcela náhodně, nebo na základě chromozomu originálního podstromu. V takovém případě jedinci nové populace vznikají aplikováním mutace na původního jedince.
2. Proběhne evaluace populace.
   (a) Chromozom každého jedince je přeložen na spustitelný program.
   (b) Dojde k otestování všech testovacích případů.
   (c) Jedincům, jejichž chromozom neumožňuje sestavení programu je přiřazena fitness hodnota MAX.
   (d) Jedincům, jejichž program nelze spustit, je přiřazena fitness hodnota MAX-1.
   (e) Výsledná fitness hodnota je vypočítána jako průměrná odchylka mezi požadovanými a očekávanými výstupy. Cílem evoluce je minimalizovat fitness hodnotu.

3. Pokud v této fázi existuje jedinec, který splňuje stanovené ukončovací podmínky, nebo již došlo k předem danému počtu iterací nebo překročení časového limitu, je algoritmus ukončen.

4. Během selekce dojde k výběru části jedinců, jedinci s lepší fitness hodnotou mají větší šanci na výběr.

5. Pomocí křížení je vygenerována část nové generace.

6. Nová generace se skládá z předem určeného poměru nových jedinců vzniklých křížením a nejlépe hodnocených jedinců předchozí generace.

7. Na všechny jedince je aplikována operace mutace a algoritmus pokračuje krokem 2.

Kapitola 5

Návrh aplikace

V této kapitole se budeme věnovat postupu výběru jednotlivých implementačních nástrojů potřebných pro realizaci aplikace umožňující automaticky opravovat jednoduchý zdrojový kód. Detailně popíšeme všechny podsystémy, které bylo pro zkompletování aplikace nutné navrhnout a naprogramovat. Hlavní důraz přitom bude kladen zejména na popis problémů, jejichž řešení se liší od tradičního přístupu a které jsou pro aplikaci jako celek nejdůležitější.

5.1 Implementační jazyk

Implementace systému zajišťujícího automatickou opravu programů je značně komplikovaná a zasahuje současně do více odvětví informatiky. Jako implementační jazyk pro tvorbu aplikace proto byl zvolen Python, jehož použití zajistí potřebnou úroveň abstrakce.

Jelikož cílovým jazykem automatické opravy je podmnožina jazyka C, je další výhodou pythonu také jednoduchý přístup k Linuxové příkazové řádce a možnost snadno implementovat výpočty probíhající ve více vláknech. Tyto vlastnosti budou stěžejní při sestavování a testování vygenerovaných zdrojových kódů.

5.2 Volba parseru

Jako parsovací nástroj pro účely implementace byl zvolen pycparser [3]. K funkcionalitě, kterou nabízí, patří jak generování AST ze zdrojového kódu, tak generování zdrojového kódu z AST. Z našeho pohledu bude výhodou i to, že pycparser je implementovaný čistě pomocí pythonu. Díky tomu bude snadné integrovat jej do aplikace.

V rámci dekódování chromozomu a generování nového AST je nutné tento AST sestavovat "ručně". Další výhodou pycparseru je tudíž i to, že umožňuje přímo využít v něm definované struktury AST, které je možné použít pro generování zdrojového kódu.

5.3 Volba překladače

Při běhu evolučního algoritmu bude nutné opakovaně kompilovat a testovat programy popsané chromozomy jedinců. V rámci evolučního algoritmu je zrychlení v každém bodě vítané a z toho důvodu bylo potřeba rozhodnout, jaký překladač bude pro naše účely nejvhodnější.

Hlavními kandidáty byly překladače GCC [2] a Clang [1]. Pro ohodnocení jejich rychlosti byl sestaven jednoduchý skript *compilerTester.py*, který umožňuje automatické zahájení daného množství překladů nebo spuštění zvoleného programu.

Měření probíhalo nad ukázkovými zdrojovými kódy, o kterých se ještě podrobněji zmíníme v kapitole o testování aplikace. Ačkoliv mezi rychlostí výpočtu programu nebyly velké rozdíly, ukázalo se, že GCC je výrazně rychlejší co se týká doby překladu. Z toho důvodu aplikace využívá překladač GCC.

5.4 Cílový jazyk pro automatickou opravu

Aplikace je implementovaná tak, aby podporovala automatickou opravu programů psaných v podmnožině jazyka C. Mezi povolené konstrukce patří podmínky, všechny typy cyklů a volání známých funkcí. Konstrukce *switch* a práce s poli a ukazateli není podporována.

Nevýhodou *pycparseru* je komplikované načítání deklarací z vložených knihoven. Z toho důvodu je možné volat pouze funkce, které jsou deklarované v rámci zpracovávaného souboru.

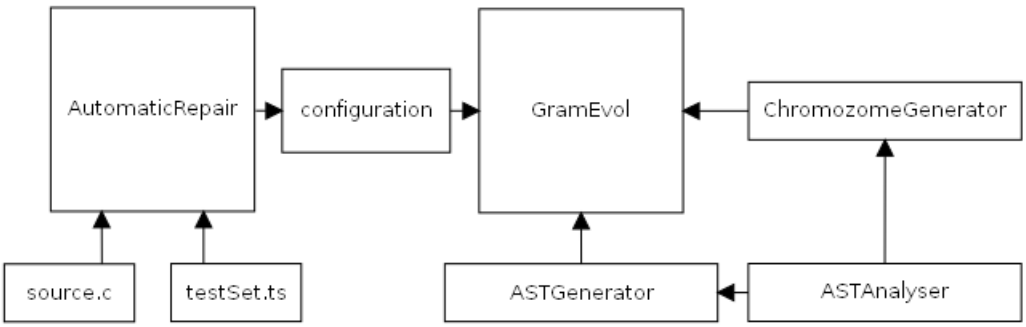
Ačkoliv automatická oprava může libovolně využívat všechny globální i lokální proměnné, deklarace nových proměnných není povolena. Implementace deklarací v generátoru AST by byla značně komplikovaná, protože by bylo nutné držet v paměti seznamy lokálních proměnných pro všechny jedince a za běhu evoluce je dynamicky upravovat, nemluvě o tom, že geny kódující proměnné by byly u různých jedinců interpretovány odlišně a pravděpodobně by docházelo k umělému nafukování kódu. Jak jsou vyřešeny deklarace v rámci původního kódu popíšeme v sekci o generování počátečního chromozomu.

Všechny unární i binární operátory a operátory přiřazení jsou podporovány, stejně tak všechny základní celočíselné datové typy i typy *float* a *double*. V sekci věnované konfiguraci opravy popíšeme, jak je možné cílový jazyk dále redukovat, za účelem zmenšení prostoru prohledávaných řešení.

Použitá gramatika včetně očíslování jednotlivých přepisovacích pravidel je k dispozici v souboru *grammar.txt* přiloženém k odevzdanému řešení.

5.5 Kostra aplikace

Na obrázku 5.1 můžete vidět základní moduly, ze kterých se skládá aplikace.



Obrázek 5.1: Znázornění modulární stavby aplikace.

AutomaticRepair - modul zodpovídá za načtení všech požadovaných vstupů - zdrojového kódu určeného k opravě a sady testovacích případů, a následné vytvoření konfiguračního objektu *GramEvolConfig*.

GramEvol – objekt řídící průběh automatické opravy a gramatické evoluce.

ASTAnalyser – implementace analýzy vstupního AST.

ChromozomeGenerator – vygenerování bázového chromozomu z podstromu AST, bázový chromozom slouží jako předloha pro inicializaci počáteční populace.

ASTGenerator – modul používaný pro evaluaci jedince, umožňuje dekódování chromozomu a vygenerování nového AST.

Automatická oprava se skládá z následujících kroků:

1. Modul *AutomaticRepair* načte ze souboru sadu testovacích případů a zdrojový kód určený k opravě.
2. *AutomaticRepair* vygeneruje konfigurační objekt a předá řízení modulu *GramEvol*.
3. *GramEvol* lokalizuje podstrom AST určený k opravě, na základě zadaných údajů.
4. Pomocí *ASTAnalyser* je zvolený podstrom AST analyzován a jsou zjištěny všechny dostupné funkce a proměnné.
5. *ChromozomeGenerator* s pomocí informací získaných z analýzy v předchozím kroku vygeneruje z podstromu AST bázový chromozom.
6. Počáteční populace je inicializována na základě bázového chromozomu.
7. *GramEvol* s využitím gramatické evoluce vygeneruje opravu. Jedinci generovaní během evoluce jsou ukládání v adresářové struktuře.

5.6 Lokalizace chyby

Jak bylo zmíněno v teoretické části, automatická lokalizace chyby je v případě použití lineárního chromozomu značne komplikovaná. Tato práce je zaměřená primárně na aplikaci evolučních algoritmů na automatickou opravu chyb a nikoliv na teorii překladačů. Z toho důvodu bylo rozhodnuto, že automatická lokalizace nebude implementována.

Aplikace umožňuje ruční lokalizaci zadáním jména funkce určené k opravě. Další kroky opravy jsou již plně automatické.

Z předchozího rozhodnutí plyne jedna změna specifikace. Oproti obecnému návrhu využívajícímu sadu negativních a pozitivních testovacích případů nyní stačí použít pouze sadu pozitivních testovacích případů k zakódování funkcionality. Negativní testovací případy jsou totiž primárně určené k automacké lokalizaci chyby a k dalšímu běhu aplikace již nejsou zapotřebí.

5.7 Reprezentace jedince

Za účelem sjednocení všech informací o jedinci, potřebných k běhu evoluce, byla implementována třída *Individual*. Obsahuje následující položky:

chrom – obsahuje pole hodnot reprezentující chromozom jedince

fitness – fitness hodnota jedince

buildCmd – objekt pro správu překladu jedince

fileName – vygenerované jméno jedince, obecně dané jako "individualN", kde N reprezentuje pořadové číslo jedince

path – kompletní cesta k souborům jedince, zohledňující číslo generace i číslo jedince

K usnadnění práce s chromozomem existuje třída *ChromozomeHandler*. Jejím úkolem je zajištění korektního čtení jednotlivých genů a kontrola počtu operací *wrap* použitých na chromozom jedince.

Kódování AST na chromozom detailně popíšeme v následující sekci.

5.8 Generování počátečního chromozomu

Základním předpokladem tohoto kroku je, že se vstupní podstrom AST od výstupního korektního řešení liší jen v drobných změnách, zatímco kostra implementace je téměř totožná. Díky tomu můžeme pomocí analýzy vstupního programu vygenerovat bázový chromozom, na jehož základě se inicializuje počáteční populace.

Pokud tento předpoklad platí, dojde k usměrnění evoluce správným směrem a značnému urychlení opravy. Dopady tohoto kroku budou detailně popsány v kapitole o testování aplikace.

V této sekci popíšeme princip kódování podstromu AST na lineární chromozom reprezentující genotyp jedince.

5.8.1 Generování chromozomu

Jak již bylo řečeno, chromozom je reprezentován lineárně – jako pole hodnot jednotlivých genů. Kódování se drží postupu popsaného v kapitole 2.5. Jeho vygenerování je realizováno pomocí rekurzivního průchodu přes AST, který byl získán pomocí parseru nástroje *pycparser*.

Pro každý typ uzlu AST je implementovaná samostatná funkce, která analyzuje své poduzly a na základě přepisovacích pravidel gramatiky rozhoduje o hodnotě dalšího genu.

5.8.2 Kódování proměnných

Kvůli kódování proměnných je napřed nutné provést statickou analýzu zdrojového kódu. K tomu slouží modul *ASTAnalyser*, který umožní z AST vstupního programu detekovat všechny dostupné globální i lokální proměnné, relativně k zadanému podstromu AST.

Ještě před generováním počátečního chromozomu je tedy nutné provést analýzu a během ní sestavit seznamy globálních a lokálních delkarací. Aplikace interně ukládá všechny proměnné v jednom poli *usableVariables*, které vzniklo spojením seznamu lokálních proměnných s seznamem globálních proměnných.

V případě výskytu libovolné proměnné během kódování stačí nahlédnout do seznamu deklarací a do chromozomu poté vložit gen, jehož hodnota odpovídá pozici dané proměnné. Implementace dekódování chromozomu zpět na AST využívá stejného principu.

5.8.3 Kódování funkcí

Kódování funkcí pracuje na stejném principu jako je tomu u proměnných. I zde je nutné v počátku vygenerovat seznam dostupných funkcí, které jsou také detekovány pomocí modulu *ASTAnalyser*.

V seznamu jsou uloženy kompletní deklarace funkcí. Tento postup je nutný, protože v případě volání funkce je potřeba vygenerovat korektní počet vstupních parametrů.

5.8.4 Kódování konstant

Kódování konstant pracuje podobně jako kódování proměnných. V současnosti aplikace povoluje pouze použití celočíselných konstant typu *int*. Seznam konstant je tvořen kombinací dvou metod.

1. První metodou je využití defaultních konstant definovaných v konfiguračním objektu. Ačkoliv jsou v konfiguračním objektu uloženy jako celočíselné hodnoty, do seznamu konstant je automaticky vygenerován seznam odpovídajících deklarací, tak aby bylo možné je přímo vložit do výsledného AST.

2. Druhou metodou je čtení existujících konstant, použitých ve vstupním zdrojovém kódu. Tato metoda staví na myšlence, že již využité konstanty budou pravděpodobně použity i v korektním řešení. Zapamatování těchto konstant tedy usnadní práci gramatické evoluci, která by jinak číselnou hodnotu musela vytvořit kombinací operátorů a defaultních konstant.

5.8.5 Kódování deklarací

Vzhledem k tomu, že cílový jazyk opravy nepodporuje deklarace nových proměnných, bylo nutné nějak vyřešit kódování deklarací v původním zdrojovém kódu.

Opět se držíme základní myšlenky, že vstupní kód je značně podobný výslednému řešení. Konkrétně předpokládáme, že k dosažení korektní implementace jsou plně dostačující proměnné, které jsou deklarované v rámci opravovaného bloku.

V aplikaci byl problém vyřešen pomocí následujících kroků:

- Deklarované proměnné jsou detekovány a uloženy do seznamu lokálních proměnných.
- Deklarace bez inicializace během generování chromozomu ignorujeme.
- Deklarace s inicializací během generování chromozomu zakódujeme jako přiřazení inicializační hodnoty do odpovídající lokální proměnné.
- Při generování zdrojového kódu vložíme deklarace proměnných na začátek opravovaného bloku, před kód reprezentovaný chromozomem.

Použití této metody významně zjednoduší kódování chromozomu a implementaci jeho mapování na AST.

5.9 Generování počáteční populace

V okamžiku, kdy známe bázový chromozom, je možné vygenerovat počáteční populaci. Prvním krokem ale je určení délky chromozomu.

5.9.1 Určení délky chromozomu

Pomocí konfiguračního objektu je možné délku chromozomu nastavit ručně. Nastavení délky chromozomu je ale vhodnější ponechat na aplikaci. Délku chromozomu je totiž nutné přizpůsobit následujícím podmínkám:

- Základní podmínkou je, aby chromozom byl dostatečně dlouhý, aby bylo pomocí něj možné reprezentovat výsledné řešení. Minimální délku můžeme odhadnout na základě předpokladu, že vstupní a výstupní kódy budou podobné. Můžeme tedy očekávat, že i délka jejich chromozomů bude podobná, potom je minimální délka rovna délce bázového chromozomu. Je tedy vhodné ponechat několik genů navíc jako rezervu pro případ, že by výsledné řešení bylo složitější a chromozom jej reprezentující byl delší.
- Současně je potřeba věnovat pozornost tomu, že čím delší je chromozom, tím menší je efektivita křížení a mutace. Tento fakt plyne z toho, že lineární chromozom je čten zleva a pravděpodobnost, že bude gen využitý při tvorbě AST, klesá se vzdáleností od začátku. Pokud by tedy chromozom byl mnohonásobně delší, než je délka bázového chromozomu, tak by se mohlo stát, že by většina genetických operací působila pouze neutrální změny genotypu. Z toho důvodu je naopak vhodné používat nejkratší možný chromozom.

S přihlédnutím k dvěma zmíněným podmínkám aplikace řeší problém tak, že minimální povolenou délku chromozomu nastavuje na dvojnásobek délky bázového chromozomu. Díky tomuto přístupu je zajištěna dostatečná délka chromozomu a současně je dosaženo vysoké efektivity genetických operátorů.

5.9.2 Generování jedinců

Když je určena vhodná délka chromozomu, stačí vygenerovat jedince do počáteční populace. Cílem je vytvořit populaci jedinců reprezentujících programy podobné programu vstupnímu. Tohoto cíle je dosaženo následovně:

- Chromozomy všech jedinců začínají bázovým chromozomem.
- Jeden náhodný gen každého jedince je modifikován aplikováním mutace.
- Konec chromozomu je prodloužen náhodnými geny, aby dosahoval minimální délky chromozomu.

Tímto postupem jsou vygenerování jedinci s novým genetickým materiálem, kteří si přesto zachovávají značnou podobnost s bázovým jedincem. Díky tomu je evoluce již usměrněna k hledání minimálně odlišných řešení.

5.10 Generování AST z chromozomu

Generování AST z chromozomu je, jak název napovídá, inverzní operací k generování chromozomu z AST. Opět je pro každý typ uzlu implementována speciální funkce, která na základě aktuálního genu rekurzivně rozgenerovává podstromy AST.

Pro usnadnění práce s chromozomem je využito pomocného objektu *ChromozomeHandler*, implementujícího funkci *GetNextGene(N)*, kde N je počet možných způsobů interpretace genu v daném okamžiku.

5.11 Generování zdrojového kódu

V předešlém textu již bylo zmíněno, že v rámci evaluace jedince je nutné sestavit jeho kód a otestovat návratové hodnoty vzniklého programu oproti očekávaným hodnotám v testovací sadě.

Kompletní postup generování nového zdrojového kódu je popsán následujícími kroky:

- Vstupní zdrojový kód musí být v jednom souboru a musí obsahovat funkci, jejíž jméno je dané konfiguračním objektem.
- Nevýhodou *pycparseru* je komplikované načítání vložených knihoven a komentářů. Z toho důvodu je nutné komentáře odstranit a příkazy *#include<..>* knihoven vyjmout a uložit pro pozdější rekonstrukci kódu.
- Modul *GramEvol* zajistí lokalizaci chyby. Lokalizací se v tomto případě rozumí to, že je v stromu AST celého programu nalezen uzel reprezentující funkci určenou k opravě. Tento uzel je interně uložen v proměnné *changedNode*.
- V případě, že zdrojový kód obsahuje funkci *main*, je nutné ji dočasně odstranit, aby bylo možné pro testování opravované funkce využít funkci *main* vygenerovanou aplikací. Alternativním řešením, které bylo použito v této implementaci, je původní funkci *main* místo mazání pouze přejmenovat.
- Pokud v původním kódu funkce *main* nebyla, je možné předchozí krok přeskočit a do kódu přímo vložit automaticky vygenerovanou funkci *main*, která zařídí pouze volání opravované funkce, předání vstupních parametrů z testovací sady a nakonec výpis návratové hodnoty funkce do souboru.
- Tělo opravované funkce v uzlu *changedNode* je nahrazeno podstromem AST vygenerovaným z chromozomu odpovídajícího jedince.
- Z kompletního AST je vygenerován zdrojový kód řešení. Před překladem je ještě nutné vložit na začátek souboru dříve vyjmuté příkazy *#include<..>* zodpovídající za import knihoven.
- Zdrojový kód jedince je tímto připravený k překladu a testování.

5.12 Gramatická evoluce

Aplikace využívá vlastní implementaci gramatické evoluce. Tento přístup zajišťuje velkou volnost při ladění algoritmu.

Základní kostra GE je postavená na teorii popsané v druhé kapitole a drží se principů běhu obecného evolučního algoritmu. Grafické znázornění můžeme vidět na obrázku 2.1. V rámci této sekce projdeme zejména odlišnosti jednotlivých částí GE od tradiční implementace.

5.12.1 Inicializace

Ve fázi inicializace algoritmu musí proběhnout čtyři zásadní kroky:

1. V první řadě je potřeba provést inicializaci pomocných modulů *chromozomeGenerator*, *ASTAnalyser* a *ASTGenerator*. Během inicializace modulů dojde k nastavení používaných unárních a binárních operátorů a operátorů přiřazení. Je nutné, aby hodnoty ve všech modulech využívaly stejná data, jinak by došlo k odchylkám při kódování a dekódování chromozomu. Důsledkem by bylo zpomalení evoluce.

2. Poté je potřeba provést statickou analýzu vstupního kódu za účelem detekce dostupných proměnných, konstant a funkcí. I zde platí stejné pravidlo jako v předchozím kroku. Pokud by například během kódování chromozomu chyběla jedna konstanta, narozdíl od seznamu konstant použitého během dekódování, došlo by k chybné interpretaci některých genů. Problém by se projevil nevhodně vygenerovanou počáteční populací a následným zpomalením evoluce.

3. Pomocí správné konfigurace modulů a korektní statické analýzy vstupního zdrojového kódu je možné zahájit generování počátečního chromozomu. Výhody tohoto kroku jsme již popsali v sekci 5.8.

4. Na základě bázového chromozomu dojde k vygenerování vhodně inicializované počáteční populace a tím ke značnému urychlení výpočtu evoluce. Více informací v sekci 5.9.

5.12.2 Evaluace

Gramatická evoluce je implementována tak, aby minimalizovala fitness funkci jedinců. Základní kroky evaluace můžeme popsat následovně:

1. Modul *ASTGenerator* se pokusí vygenerovat podstrom AST na základě lineárního chromozomu jedince.

2. Pokud bylo generování neúspěšné, je jedinci přiřazena nejhorší možná fitness hodnota, v našem případě je to hodnota *sys.maxint*. Takové ohodnocení minimalizuje výskyt vadných jedinců v budoucích generacích.

3. Pokud se generování podstromu AST podařilo, je podstrom vložen do uzlu *changedNode* v původním zdrojovém kódu. Výsledný kód je současně obohacený o automaticky vygenerovanou funkci *main*, zajišťující volání opravované funkce během testu.

4. Je zahájena kompilace vygenerovaného kódu.

5. Pokud kompilace selhala, je jedinci přiřazena druhá nejhorší fitness hodnota - *sys.maxint-1*. Díky tomu je budoucí výskyt zmíněných jedinců omezem, ale stále dojde k jejich upřednostňování před jedinci, jejichž chromozom je úplně nepoužitelný.

6. Správně přeložený program jedince je otestován na vstupech z testovací sady. V případě, že program pro některý ze vstupů nevrátí výsledek, je jedinci přiřazena fitness hodnota *sys.maxint-2*. V rámci cílového jazyka tento problém může nastat primárně z důvodu zacyklení nebo uváznutí v rekurzi. Detekce tohoto problému je popsána v 5.12.4.

7. Pokud přeložený program odpoví na všechny testovací případy, je jeho fitness hodnota vypočítána jako průměrná odchylka od požadovaného výstupu, definovaného v odpovídajícím testovacím případu.

Během výpočtu je třeba generovat řadu souborů nutných pro ohodnocení jedince. Všechna data jsou uložena v jednotné adresářové struktuře, organizované v první úrovni podle čísla generace a v druhé úrovni podle čísla jedince.

Pro každého správně ohodnoceného jedince v dané generaci N jsou vygenerovány následující tři soubory:

```
./_repairData/generationN/individualM.c
                         /individualM
                         /individualM.txt
```

Obrázek 5.2: Adresářová struktura se soubory vygenerovanými během evaluace jedince.

individualM.c – vygenerovaný zdrojový kód jedince M.

individualM – binární soubor s přeloženým programem jedince M.

individual.txt – textový soubor, ve kterém jsou uloženy výsledky jednotlivých volání programu *individualM*, oddělené mezerou.

5.12.3 Urychlení evaluace

Evaluace využívající generování, kompilaci a spouštění programů v reálném čase, je výpočetně i časově velmi náročná.

Značnou optimalizací bylo provádění překladů a spouštění programů paralelně ve více procesech. Implementace výpočtů ve více procesech s využitím Pythonové modulu *subprocess* [4] byla v rámci projektu realizována.

K ověření zrychlení byl použit již dříve zmíněný skript *compilerTester.py*, který, mimo jiné, umožňuje testování sériového zpracování oproti paralelnímu. Ukázalo se, že již od paralelního překladu v řádech desítek procesů je rychlost oproti sériovému výpočtu přibližně dvojnásobná.

Hlavním úskalím tohoto přístupu je potenciálně obrovské množství souběžně pracujících procesů, které by teoreticky mohlo požadovat více výpočetních prostředků, než nabízí aktuálně používaný hardware. Z toho důvodu bylo implementováno dávkové zpracování, které umožňuje maximálně N souběžně pracujících procesů.

5.12.4 Zacyklení a uváznutí v rekurzi

Pokud bychom chtěli úplně zamezit vzniku zacyklení a nekonečné rekurzi v automaticky generovaných programech, bylo by bezpodmínečně nutné naprosto vyloučit generování cyklů a volání funkcí. Toto řešení ale nepřipadá v úvahu, kvůli důležitosti cyklů a funkčních volání. Je tedy nutné problém ošetřit jiným způsobem.

Nejjednodušším řešením je omezení výpočetního času programu. V implementované aplikaci je možné maximální výpočetní čas programu nastavit s využitím konfiguračního objektu.

Pokud navíc vezmeme v úvahu paralelní testování přeložených programů, tak teoreticky můžeme oproti sériové evaluaci dosáhnout zrychlení mnohonásobně vyššího, než jaké je u kompilace. Plyne to z toho, že v případě paralelního výpočtu zacyklený program nebrzdí evaluaci ostatních.

5.12.5 Konec evoluce

Ukončovací podmínka evoluce se drží tradičního postupu. Výpočet může být zastaven v případě, že byl nalezen jedinec s fitness hodnotou *0*. Tento stav odpovídá nalezení řešení, které vrací správné výsledky pro všechny testovací případy testovací sady.

Druhou podmínkou je prosté překročení maximálního počtu generací. V případě takového ukončení se nepodařilo nalézt kompletní řešení, uživateli je tedy zobrazena alespoň informace o nejlepších výsledcích, kterých bylo dosaženo během poslední generace.

5.12.6 Selekce

Cílem selekce je vybrat kandidátní jedince, kteří budou dále použiti ve fázi křížení pro vytvoření nové populace. Selekce následuje hned za evaluací jedinců. Dá se tedy říct, že jejich fitness hodnoty v tuto chvíli reprezentují jejich pomyslnou vzdálenost od hledaného řešení. Vzhledem k tomuto faktu nemůžeme přímo použít ruletového výběru, protože základní verze rulety vybírá jedince s pravděpodobností úměrnou jejich fitness. Tím pádem by docházelo tedy právě k propagaci nejhorších řešení do další generace.

Je tedy nutné v první řadě přepočítat fitness tak, aby nejkvalitnějším jedincům byla přiřazena nejvyšší hodnota. Pro dosažení tohoto efektu byl použit následující algoritmus:

1. Dojde k sestupnému seřazení populace, což odpovídá seřazení od nejhoršího k nejlepšímu.

2. Nejhůře ohodnocený jedinec, který je na první pozici v seřazeném poli, získá základní fitness hodnotu *b=1*.

3. Algoritmus přejde na dalšího jedince.

   (a) Pokud je jeho fitness hodnota stejná, jako původní ohodnocení předchůdce, je aktuální jedinec též ohodnocen předchozí hodnotou *b*.

   (b) V případě, že je jeho fitness hodnota nižší než u fitness hodnota předchůdce, můžeme říct, že je jeho kvalita o krok lepší. Jeho nové ohodnocení tedy bude rovno inkrementované předchozí hodnotě, programově zapsáno *b+1*.

4. Dokud existuje v seřazené populaci jedinec, jehož ohodnocení nebylo přepočítáno, pokračuje výpočet krokem 3.

Předchozí výpočet vlastně realizuje invertovanou lineární normalizaci fitness hodnot. Výhodou linearizace je zajištění větší diverzity v prohledávaném prostoru. V případě, že původní fitness hodnoty byly již před normalizací uspořádány rovnoměrně, nedojde k žádné změně.

Pokud by ovšem původní populace obsahovala například jedince, jehož fitness hodnota je v řádu tisíců, zatímco ohodnocení ostatních by bylo v řádech jednotek, došlo by s nejvyšší pravděpodobností k přemnožení prvního jedince a genetický materiál ostatních kandidátů by byl nenávratně ztracen, přestože teoreticky může být potřebný k objevení řešení.

Jak již bylo zmíněno v druhé kapitole, použití normalizované fitness hodnoty nabízí řadu možností pro ovlivňování výsledného selekčního tlaku. V implementované aplikaci je k jeho ovlivňování možné využít parametr konfiguračního objektu zvaný *minMaxFitnessRatio*.

Bezprostředně po normalizaci fitness hodnot je populace ohodnocena následovně:

- Nejhůře ohodnocený jedinec má přiřazenu hodnotu 1.

- Nejlépe ohodnocený jedinec je ohodnocen číslem N, kde N reprezentuje počet odlišných fitness hodnot, které se vyskytly během evaluace.

Poměr mezi ohodnocením nejlepšího a nejhoršího jedince je tedy roven N. Zmíněný parametr *minMaxFitnessRatio* obsahuje uživatelem zadanou hodnotu M, reprezentující požadovaný přibližný poměr mezi fitness hodnotou nejlepšího a nejhoršího jedince. Definiční obor hodnot M můžeme popsat intervalem $(2, N)$.

Pro dosažení uživatelem požadovaného poměru mezi fitness hodnotami je nutné provést posun všech fitness hodnot. Velikost tohoto posunu vypočtena pomocí následujícího vzorce $s = 1/(M-1)$. Výsledná hodnota s je posléze přičtena k normalizovaným fitness hodnotám všech jedinců.

Pomocí této úpravy je možné shora pevně omezit rozdíly mezi nejlepšími a nejhoršími fitness hodnotami. Důsledkem čehož dochází k potenciálnímu snížení selekčního tlaku a zvýšení diverzity jedinců při prohledávání prostoru možných řešení.

Pro dosažení většího poměru než N by bylo potřeba použít jinou metodu normalizace, jako je třeba kvadratická normalizace.

Po všech předchozích úpravách je nyní možné jedince do nové generace volit pomocí ruletového výběru nad modifikovanými fitness hodnotami.

5.12.7 Křížení

Křížení plně využívá výhody gramatické evoluce, kterou je lineární reprezentace chromozomu. Je tedy možné používat klasické jednobodové křížení.

Princip jednobodového křížení dvou jedinců A a B můžeme popsat jednoduchým postupem:

1. Dojde k náhodnému zvolení bodu křížení C, tento bod je reprezentován hodnotou z intervalu $(0, L-1)$, kde L reprezentuje délku chromozomu.

2. Nově vzniklý chromozom potom obsahuje geny z chromozomu jedince A od pozice *0* po C a geny jedince B od *(C+1)* do L.

5.12.8 Mutace

Realizace mutace je založena na tradiční implementaci mutace nad lineárním chromozomem. Pravděpodobnost její aplikace je dána hodnotou *mutationChance* nastavenou v konfiguračním objektu.

5.12.9 Obnova

Obnova je důležitou součástí evolučního algoritmu. Její implementace určuje postup vzniku nové generace. Hlavním parametrem ovlivňujícím obnovu je proměnná konfiguračního objektu *renewalOldIndividualsRatio*. Tento parametr obsahuje hodnotu z intervalu $(0, 1)$ určující procentuální zastoupení jedinců z předchozí generace v generaci nové.

Zachování nejlepších jedinců předchozí generace, často označované jako elitismus, je důležitou strategií při implementaci EA. Jejím hlavním přínosem je zejména to, že jakmile algoritmus objeví jedince relativně vysoké kvality, není již možné během dalšího výpočtu ztratit jeho genetický materiál. Elitismus je v implementované aplikaci podporován.

5.13 Konfigurace opravy

Pro konfiguraci automatické opravy slouží konfigurační objekt *GramEvolConfig*. Jeho modifikace je aktuálně založena na úpravě hodnot odpovídajících proměnných ve zdrojovém souboru *main.py*. Automatická oprava zapouzdřuje řadu problémů od lokalizace, přes analýzu vstupního kódu až po nastavení gramatické evoluce. Z toho důvodu je konfigurační objekt poměrně rozsáhlý. V této kapitole proto popíšeme jeho jednotlivé oddíly.

5.13.1 Lokalizace chyby a nastavení výsledné adresářové struktury

Před automatickou opravou chyby je nutné ručně zadat jméno cílené funkce. V rámci konfiguračního objektu je jméno uloženo v proměnné *funcname*.

Další informace o konfiguraci v rámci tohoto oddílu jsou již nastavovány automaticky. Patří k nim:

filename – jméno vstupního souboru se zdrojovým kódem. Hlavním důvodem uložení této informace je to, že *pycparser* vyžaduje jméno analyzovaného souboru během generování AST pro korektní vypisování chybových zpráv. Jméno souboru je tedy uloženo primárně kvůli účelům ladění aplikace, na samotný běh opravy jinak nemá vliv.

code – uložený zdrojový kód načtený ze vstupního souboru *filename*. Původní zdroj je ve fázi inicializace opravy analyzován a na jeho základě je vygenerován bázový chromozom a poté počáteční populace.

header – již zmíněnou nevýhodou *pycparseru* je komplikované načítání vložených knihoven, z toho důvodu je potřeba z původního kódu vyjmout odpovídající příkazy *#include<..>*. Později během rekonstrukce kódu je nutné zmíněné příkazy opět vložit zpět. Z toho důvodu jsou uloženy v konfigračním objektu v proměnné *header*.

functionType – typ opravované funkce, jeho znalost je důležitá zejména pro korektní převod typovou konverzi při evaluaci jedinců, a také během generování nové funkce *main* určené k řízení testování.

inputTypes – seznam typů vstupních parametrů opravované funkce. Uložení tohoto seznamu je důležité ze stejného důvodu, jako je tomu u *functionType*.

5.13.2 Redukce cílového jazyka

Cílový jazyk pro automatickou opravu podporuje všechny binární i unární operátory a operátory přiřazení, které jsou normálně dostupné v jazyce *C*. V některých případech ale může být rozumné množinu dostupných operátorů redukovat, s cílem omezit generování některých řešení.

Jako demonstrace této myšlenky může posloužit jednoduchá funkce pro výpočet obsahu nějakého geometrického obrazce. Je zřejmé, že pro výpočet budou nutné operace jako je sčítání, odečítání, násobení apod. Oproti tomu například binární operátor pro bitový posun pravděpodobně vhodný nebude. Redukce počtu opetátorů za předpokladu, že je stále dostačující k implementaci řešení, má pozitivní vliv na běh evoluce, jelikož díky ní dochází k zmenšení prohledávaného prostoru.

Popsanou redukci je možné provést ručním vytvořením množiny operátorů a jejím uložení do vhodného seznamu v konfiguračním objektu. Jména dostupných seznamů jsou následující:

unaryOperators – seznam unárních operátorů.

binaryOperators – seznam binárních operátorů.

assignmentOperators – seznam operátorů přiřazení.

Aplikace ošetřuje případy, ve kterých by uživatelem zadaná množina operátorů neobsahovala některý z operátorů vyskytujících se v rámci opravované funkce. V opačném případě by došlo k selhání během generování chromozomu. Uživatelem nastavený seznam je tedy použit pouze jako minimální množina operátorů, která může během analýzy vstupního kódu být ještě automaticky rozšířena.

Příklad redukce operátorů je uveden na obrázku 5.3.

```
config.unaryOperators = ['++','p++','--','p--']
config.binaryOperators = ['*','/','%','+','-']
config.assignmentOperators = ['=','*=','/=','%=','+=','-=']
```

Obrázek 5.3: Ukázka volby operátorů cílového jazyka.

5.13.3 Nastavení gramatické evoluce

Většina parametrů implementované gramatické evoluce již byla podrobně popsána v příslušných sekcích návrhu. V této části je přehled kompletního nastavení GE.

useBaseChromozome – booleovská proměnná, pokud je nastavena na *True*, je počáteční populace inicializována na základě bázového chromozomu. V opačném případě je počáteční populace vygenerována zcela náhodně.

chromLen – celočíselná hodnota, určující délku chromozomu. Nastavení této hodnoty je důležité zejména v okamžiku, kdy není použita inicializace bázovým chromozomem a aplikace tedy nemá podle čeho délku vypočítat. V opačném případě je vhodnější nastavení délky ponechat na aplikaci.

maxWrapCount – počet operací *wrap* použitých na chromozom. Více informací je možné najít v sekci 2.5.2.

populationSize – celočíselná hodnota udávající počet jedinců v populaci.

generationCount – celočíselná hodnota udávající maximální počet generací.

mutationChance – celočíselná hodnota určující pravděpodobnost mutace, hodnota *5* znamená pravděpodobnost 5%.

renewalOldIndividualsRatio – číselná hodnota v intervalue $(0, 1)$, popisující procentuální podíl jedinců předchozí populace v populaci nové.

minMaxFitnessRatio – číselná hodnota popisující požadovaný poměr fitness hodnot mezi nejhorším a nejhorším jedincem, podle zadané hodnoty dojde k modifikaci normalizovaných fitness hodnot jedinců.

maxWaitTime – čas v sekundách určující maximální délku čekání na dokončení výpočtu procesu.

generationHistoryLength – celočíselná hodnota udávající maximální počet generací současně uložených v adresářové struktuře.

evaluationBlockSize – celočíselná hodnota určující maximální počet souběžně pracujících procesů.

5.14 Tvorba testovacích případů

Vzhledem k absenci automatizované lokalizace je pro správný chod aplikace potřebná pouze pozitivní testovací sada. Pozitivní testovací případy existují primárně za účelem kódování správné funkcionality. Pro jejich tvorbu se v našem případě nabízí dva základní postupy. Prvním je jejich ruční tvorba, druhou možností je vygenerovat je programově.

5.14.1 Ruční tvorba testovací sady

Ruční sestavení testovací sady s sebou přináší jednu důležitou výhodu. V případě, že testy tvoří člověk, který rozumí cílovému výpočtu, může funkcionalitu zakódovat velice úsporně - pomocí relativně malého počtu případů. Aby toho dosáhl, stačí se zaměřit na oblasti definičního oboru, které jsou pro daný výpočet kritické, a testovací případy koncentrovat do jejich blízkosti.

Slabinou tohoto přístupu je, že sestavení kvalitní testovací sady zabere nemalé množství času. Navíc stále existuje riziko, že se evoluční algoritmus vyvine takovým způsobem, aby řešil pouze příklady z testovací sady a nedojde ke správné generalizaci řešení.

5.14.2 Automatické generování testovací sady

Problémem automatického generování je to, že není možné jej aplikovat vždy. Plyne to z toho, že testovací případy jsou tvořené pro kódování funkcionality, jejíž správnou programovou reprezentaci v danou chvíli neznáme. Kdybychom hledanou implementaci předem znali, proč bychom se ji potom snažili hledat pomocí evolučního výpočtu?

V našem případě je automatické generování naštěstí použitelné. U všech programů, které byly použity při testování aplikace, je správná implementace známá a opravované chyby jsou vloženy uměle.

Pro tvorbu testovacích případů tedy byl sestaven pomocný skript *testSetGen.py*, který umožňuje automatické vygenerování testovacích případů pro programově zakódovanou hledanou funkcionalitu. Všechny testovací sady použité při testování byly vygenerovány pomocí tohoto skriptu. Podrobněji se testovacím sadám a testování budeme věnovat v následující kapitole.

5.14.3 Formát testovací sady

Testovací sady jsou uloženy v textových souborech s příponou *.ts*. Každý řádek testovací sady reprezentuje jeden testovací případ.

Každý testovací případ je složen z odpovídajícího množství vstupních hodnot a jednou hodnotou reprezentující očekávaný výstup. Pro zápis do souboru je použit následující formát:

- Všechny vstupní hodnoty jsou zapsány za sebou, oddělené mezerou.

- Očekávaná výstupní hodnota je zapsána jako poslední hodnota sekvence, od ostatních oddělená také mezerou.

Jednoduchým příkladem může být testovací případ kódující součet dvou vstupů:

29 13 42

V příkladu můžete vidět hodnoty vstupů 29 a 13, následované očekávaným výstupem, jehož hodnota je 42.

Kapitola 6

Experimentální ověření funkčnosti aplikace

Důležitou součástí této práce je experimentální ověření dosažených výsledků. V této kapitole se tedy budeme věnovat právě zmíněné tématice.

V první části popíšeme zdrojové kódy použité při testování. Poté se budeme věnovat postupu, kterým byla ověřena činnost implementované gramatické evoluce. V poslední části prezentujeme výsledky automatické opravy na konkrétních testovacích programech.

Veškeré testování aplikace probíhalo na školních serverech *edesign1-4*.

6.1 Programy použité při testování

Cílem práce bylo dosáhnout automatické opravy chyb v jednoduchém kódu. Programy, na kterých bylo prováděno testování, byly speciálně navrženy se zaměřením na demonstraci opravy chyby v různých typech programovacích konstrukcí.

6.1.1 Návrh testovacích programů

Obecný princip návrhu těchto ukázkových zdrojových kódu je jednoduchý. V první řadě bylo potřeba vymyslet a implementovat jednoduchý program využívající cílenou programovací konstrukci. Jako příklad cílené programovací konstrukce můžeme uvést zaměření na aritmetické operace, podmínky, cykly, nebo volání funkcí. Omezení při implementaci těchto programů je dáno pouze dříve zmíněnou specifikací cílového jazyka.

Druhým krokem je umělé vytvoření chyby, určené k opravě. Jelikož se oprava zaměřuje na sémantické chyby, je podmínkou, aby testovací kód byl i po vložení chyby syntakticky správný. Během testování jednotlivých programů většinou docházelo k opravování více odlišných typů chyb. Detaily tedy popíšeme až ve specializovaných sekcích.

6.1.2 Návrh a vkládání chyb

Při umělém vytváření chyb bylo cílem co největší přiblížení k reálným situacím, při kterých programátor nedopatřením zanesl do zdrojového kódu chybu. K takovým případům může patřit zaměnění operátoru nebo proměnné. V případě cyklů půjde nejčastěji o chybný výpočet počtu opakování cyklu.

6.1.3 Triviální chyba

Z důvodu usnadnění popisu výsledků v následujících sekcích zavedeme pojem *triviální chyba*. Triviální chybou se rozumí taková chyba, jejíž opravy je možné docílit pomocí vhodné změny jediného genu. Jako příklad uvedeme záměnu jedné proměnné nebo záměnu jednoho operátoru.

Oproti tomu záměna konkrétní konstanty za konkrétní proměnnou již za triviální chybu ve většině případů považovat nelze. Plyne to ze způsobu kódování chromozomu v GE. Pokud vezmeme v úvahu například příkaz přiřazení, tak přiřazovanou hodnotu kódujeme minimálně pomocí dvou genů. První gen definuje druh hodnoty - konstantní nebo proměnná. Druhý gen potom reprezentuje adresu v seznamu konstant resp. proměnných.

6.1.4 Součet obsahů obdélníků

rectangleSurfaceSum je jednoduchá funkce, která implementuje součet obsahů dvou obdélníků, zadaných pomocí jejich výšky a šířky. Jejím primárním cílem je ověřit, že námi vyvinutá aplikace dokáže opravit chyby v jednoduchém programu, využívajícím převážně aritmetické operace. Implementaci funkce v jazyce C můžeme vidět na obrázku 6.1.

```
int rectangleSurfaceSum(int w1, int h1, int w2, int h2){
    return w1*h1+w2*h2;
}
```

Obrázek 6.1: Implementace testovací funkce *rectangleSurfaceSum*.

6.1.5 Maximum z tří

Pro ověření schopnosti opravovat chyby v programu, složeném ze sítě podmínek, byla implementována funkce *maxFromThree*. Jak z názvu vyplývá, funkce realizuje hledání maxima mezi třemi vstupními hodnotami. Implementace funkce v jazyce C je znázorněna v obrázku 6.2.

```
int maxFromThree(int a,int b,int c){
    if(a>b){
        if(a>c){
            return a;
        }
        else {
            return c;
        }
    }
    else {
        if(b>c){
            return b;
        }
        else {
            return c;
        }
    }
}
```

Obrázek 6.2: Implementace testovací funkce *maxFromThree*.

6.1.6 Násobení pomocí cyklu

Funkce *forMul* implementuje matematickou funkci $a * (b + c)$ s využitím cyklu *for*. Účelem této funkce je zjistit, zda aplikace dokáže automaticky opravit sémantické chyby, které mohou nastat při cyklickém výpočtu.

Implementace funkce v jazyce C je znázorněna v obrázku 6.3.

```
int forMul(int a, int b, int c){
    for(int i = 0;i<a;i++){
       sum += b+c;
    }
    return sum;
}
```

Obrázek 6.3: Implementace testovací funkce *forMul*.

6.2 Ověření funkčnosti gramatické evoluce

Před zahájením ověřování automatické opravy bylo nejprve nutné zjistit, zda je implementovaná gramatická evoluce schopná hledání řešení. Současně je na místě vyzkoušet, zda je GE naprogramovaná dostatečně univerzálně a je možné ji využít i na jiných problémech.

6.2.1 Jednoduchá symbolická regrese

Pro ověření bylo nutné zvolit jednoduchý problém, který by byl řešitelný využitím GE. Zvolenou metodou byla jednoduchá symbolická regrese, zkráceně SR. Navržená SR používá pouze binární aritmetické operátory +,-,*,/ a %.

V okamžiku, kdy chceme aplikovat implementovanou GE na jiný problém, než je automatická oprava chyb, je nutné implementovat novou evaluační funkci a v našem případě i novou funkci pro inicializaci. Ostatní kroky evolučního algoritmu jsou navržené obecně a není potřeba na nich provádět žádné změny.

Evaluace jedince, ve funkcích *SREvaluation*, *SRBuildPopulation* a *SREvaluatePopulation*, je postavena na vygenerování binárního stromu operátorů reprezentujícího řešení. Pro získání návratové hodnoty poté stačí rekurzivní průchod vytvořeným stromem. Za účelem tvorby binárního stromu byla vytvořena třída *SRNode*, reprezentující uzel binárního stromu, která obsahuje následující vlastnosti:

nodeType – typ uzlu, který může nabývat tří hodnot:

- operátor,
- proměnná a
- konstanta.

val – hodnota uzlu, její reprezentace je závislá na hodnotě *nodeType*:

operátor – celé číslo odkazující na jeden z dostupných operátorů.

proměnná – celé číslo odkazující na jednu z dostupných proměnných.

konstanta – celé číslo odkazující na jednu z předdefinovaných konstant.

left – levý podstrom.

right – pravý podstrom.

Implementace nové inicializační funkce v *SRInit* je v podstatě jen zjednodušením původní funkce *Init*. Hlavním krokem je odstranění analýzy vstupního podstromu AST a nastavení náhodného generování počáteční populace.

6.2.2 Postup testování

Testování probíhalo převážně na stejném výpočtu, jaký realizuje testovací funkce *rectangleSurfaceSum*. Pro zvolení tohoto výpočtu existují dva důvody.

- Automatické vygenerování matematické funkce $w1 * h1 + w2 * h2$ z testovacích případů není zcela triviální. GE musí v tomto případě dokázat objevit správné operátory, aplikovat je na správné proměnné, a aplikace operátorů musí proběhnout ve správném pořadí. Oproti tomu třeba funkce $a + b + c + d$ postrádá potřebu pro zachování pořadí operátorů, její vygenerování je tedy značně jednodušší.
- Díky použití stejného výpočtu můžeme provést srovnání dvou odlišných přístupů při hledání stejného řešení.
  - Výhodou SR je výrazně menší prohledávaný prostor řešení.

– Výhodou implementované automatické opravy je možnost inicializace počáteční populace, která svým způsobem umožňuje vhodné nasměrování evoluce.

V případě, že testování v obou případech provedeme nad stejným výpočtem, můžeme velice konkrétně srovnat dosažených výsledků a porovnat, který z přístupů má větší vliv na rychlost hledání řešení.

Během testování byla použita stejná testovací sada jako pro testování funkce *rectangleSurfaceSum*. Ta bude podrobněji popsána v následující sekci.

6.2.3 Výsledky ověřování GE

Během ověřování GE bylo zafixováno nastavení, které můžeme shlédnout v tabulce 6.1.

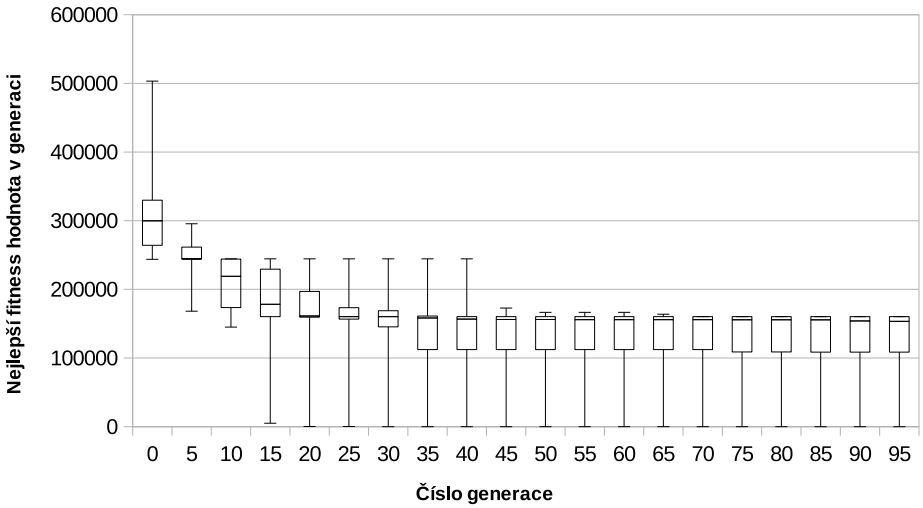
Tabulka 6.1: Nastavení gramatické evoluce použité při testování SR.

chromLen	100
generationCount	100
renewalOldIndividualsRatio	0.05
minMaxFitnessRatio	10
mutationChance	5
maxWrapCount	2

Kromě ověření konvergence GE bylo cílem zjistit, jak velká populace je nutná pro objevení řešení při daném limitu 100 generací. V grafech s výsledky je znázorněn vývoj nejlepších fitness hodnot v generaci, vždy z dvaceti testovacích běhů.

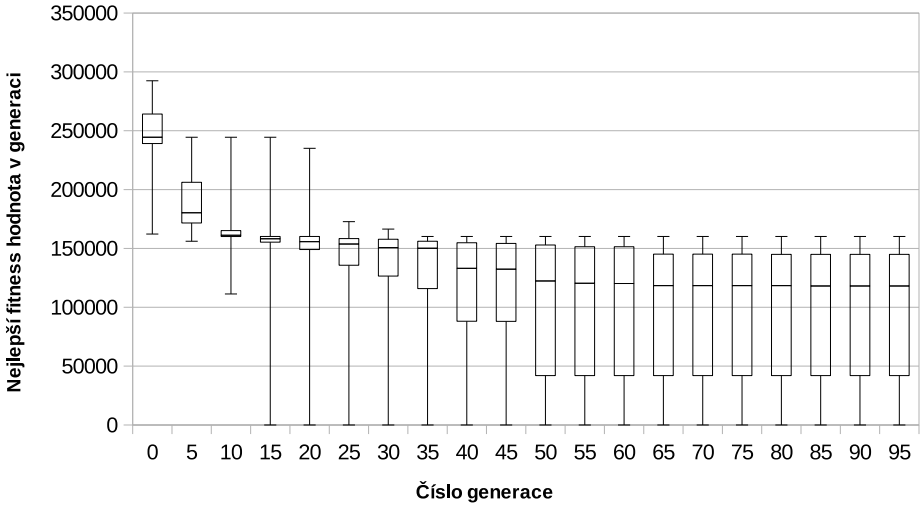
Testování proběhlo pro následující velikosti populace:

100 - výsledky ukazuje obrázek 6.4. Většina běhů nedosáhla hledaného řešení. Ve skutečnosti byly pouze čtyři běhy z 20 úspěšné. Dle neměnných hodnot na konci grafu můžeme předpokládat, že neúspěšní jedinci se ustálili v lokálních optimech a jejich konvergence k hledanému řešení je nepravděpodobná.



Obrázek 6.4: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 100, při testování SR.

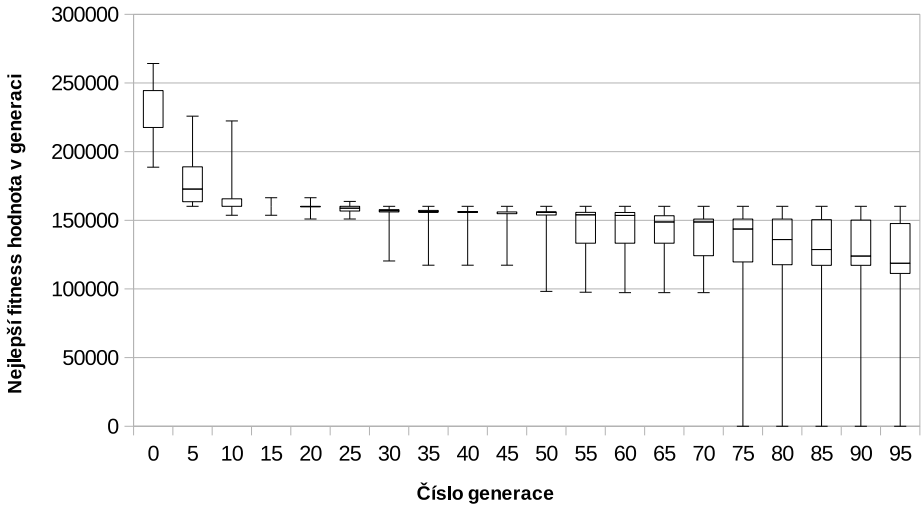
500 - výsledky znázorněny v obrázku 6.5. Oproti předchozímu testu je vidět určité zlepšení. Úspěšnost běhů ale stále není dostačující, při velikosti populace 500 bylo řešení nalezeno pouze pěti jedinci.



Obrázek 6.5: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 500, při testování SR.

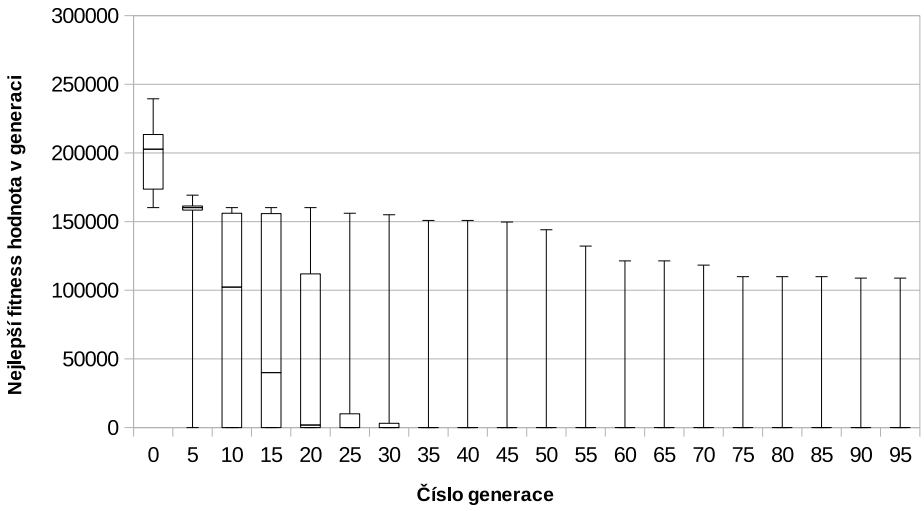
1000 - výsledky znázorněny v obrázku 6.6. Očekávaným jevem při zvětšování populace bylo kontinuální zlepšování výsledku. Výsledky získané při testováním při populaci velikosti 1000 se ale vymykají normě. Jednou z možností je, že GE nepracuje správně. Druhou možností je, že daná velikost populace je k řešení daného problému stále nedostačující. To by znamenalo, že hodnoty v předchozích dvou grafech jsou i přes vícenásobné měření silně

ovlivněny náhodou. Pro získání definitivní odpovědi je nutné provést měření na ještě vyšším počtu jedinců.



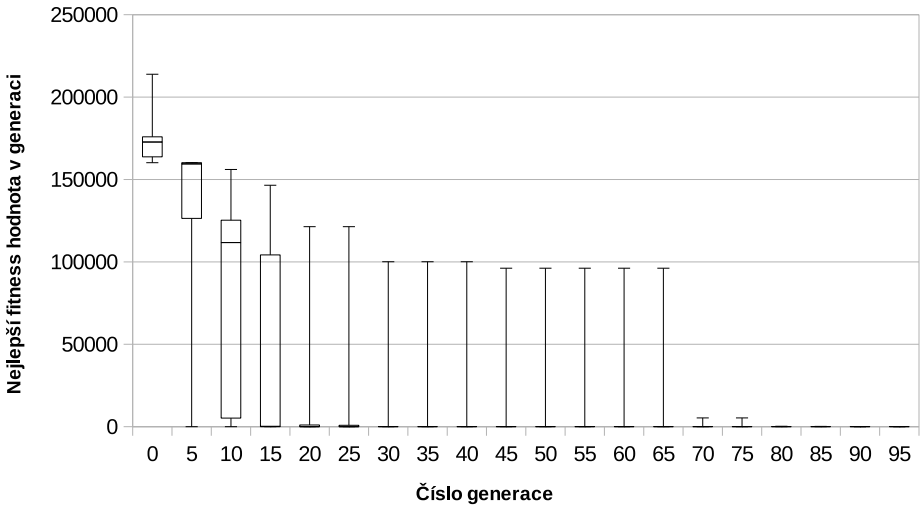
Obrázek 6.6: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 1000, při testování SR.

5000 - výsledky znázorněny v obrázku 6.7. Ukázalo se, že při populaci 5000 je SR již schopná vygenerovat korektní řešení ve většině případů. Jedná se o významné zlepšení oproti předchozímu měření.



Obrázek 6.7: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 5000, při testování SR.

10000 - výsledky znázorněny v obrázku 6.8. Poslední měření potvrzuje, že vyšší počet jedinců implikuje větší úspěšnost při hledání řešení. V případě populace velikosti 10000 je úspěšnost po dvaceti bězích stále 100%.



Obrázek 6.8: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 10000, při testování SR.

Můžeme říct, že ověření funkcionality gramatické evoluce proběhlo úspěšně. Z výsledků posledních dvou měření jasně vyplývá, že výpočet konverguje k hledanému řešení.

6.3 Součet obsahu obdélníků

Během testování byla použita testovací sada, která obsahovala celkem 200 testovacích případů a byla automaticky vygenerována na základě požadované funkcionality - výpočtu funkce:

$$w1 * h1 + w2 * h2$$

Během testování bylo zafixováno základní nastavení GE, stejně jako tomu je v předchozí sekci v tabulce 6.1. Limit počtu generací byl opět nastaven na hodnotu 100.

6.3.1 Opravené chyby

V počáteční fázi byla otestována automatická oprava řady různých triviálních chyb. Patřily k nim záměny proměnných jako například:

$$w1 * \underline{w1} + w2 * h2$$

nebo záměny operátorů jako:

$$w1 * h1 + w2 \underline{+} h2$$

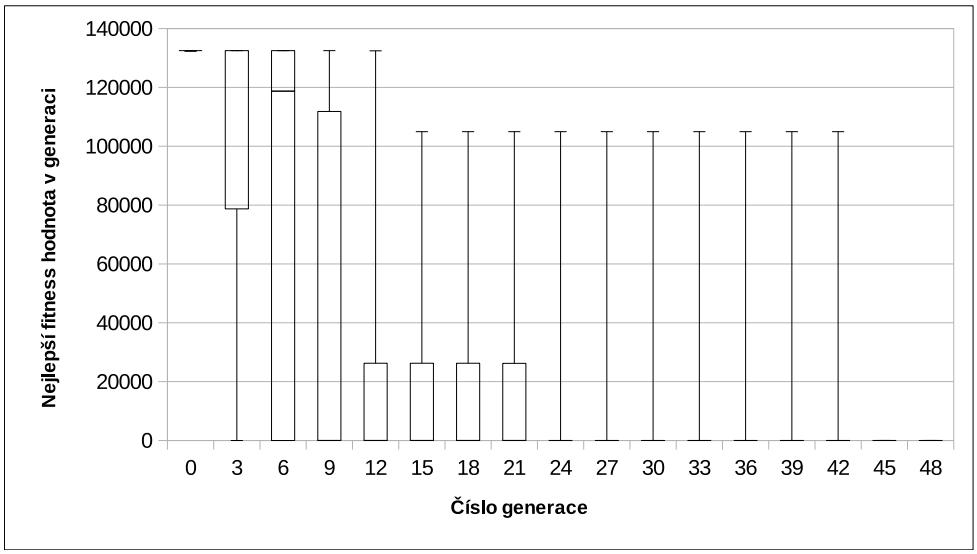
Ukázalo se, že aplikace všechny druhy triviálních chyb opravuje velice efektivně. I v případě malé počáteční populace o 100 jedincích bylo řešení v drtivé většině případů nalezeno již při první evaluaci. Toto chování bylo způsobeno vhodnou inicializací počáteční populace.

Dalším krokem byla automatická oprava složitějších chyb. V tomto případě probíhalo testování na zdrojovém kódu s dvěma nezávislými triviálními chybami:

$$w1 * \underline{h2} + w2 * \underline{h1}$$

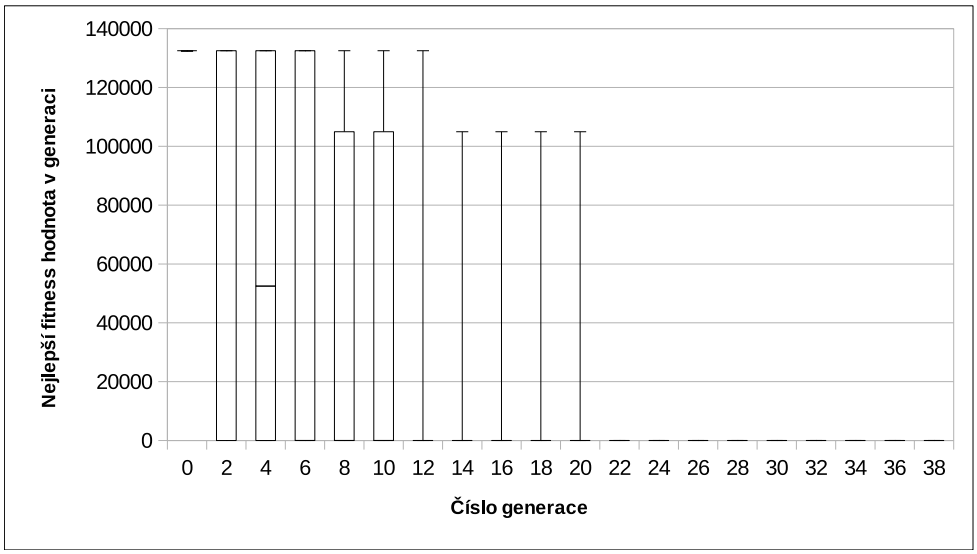
Můžeme vidět, že v opravovaném programu došlo k dvojité záměně proměnných. Oprava dvojité chyby je značně náročnější, protože je nutné dosáhnout dvou korektních modifikací na různých místech chromozomu.

Výsledky z dvaceti běhů při populaci 100 můžeme vidět na obrázku 6.9.



Obrázek 6.9: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 100, při opravování dvojité chyby v funkci *rectangleSurfaceSum*.

Výsledky z dvaceti běhů při populaci 200 znázorňuje obrázek 6.10.



Obrázek 6.10: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 200, při opravování dvojité chyby v funkci *rectangleSurfaceSum*.

Z obrázků 6.9 a 6.10 je možné vyčíst několik zajímavých informací. Nejpodstatnější z nich je samozřejmě fakt, že oprava dvojité chyby byla úspěšná.

Důležitý je ale i samotný průběh hledání opravy. Z obrázku jde jasně rozpoznat, že vylepšování fitness hodnot nejkvalitnějších jedinců probíhalo převážně skokově. Oproti tomu optimalizace fitness hodnot při testování SR probíhala mnohem plynuleji, viz. 6.7.

Důvodem tohoto rozdílného chování je zejména způsob hledání korektního řešení. V případě SR, která byla inicializována zcela náhodně, dochází k prohledávání rozsáhlého prostoru řešení. Přímým důsledkem je to, že dochází častěji k menším změnám fitness.

Implementovaná automatická oprava narozdíl od toho prohledává omezený prostor kolem pevně daného bodu, určeného původním programem. Množství změn, které vedou ke zlepšení fitness hodnoty, je značně redukované, z toho důvodu dochází ke zmíněným skokům.

V tuto chvíli je vhodné připomenout výsledky dosažené při generování stejného výpočtu pomocí jednoduché SR. Ke spolehlivému nalezení správného řešení stačil v případě vhodně inicializované automatické opravy pouze zlomek počtu generací, i přesto, že zmíněná SR povolovala pouze pět binárních operátorů a složitost jejího kódování oproti kódování cílového jazyka atomatické opravy je značně nižší.

Vliv inicializace počáteční populace pomocí bázového chromozomu je tedy jednoznačně pozitivní.

6.4 Maximum z tří

Funkce *maxFromtThree* je mnohem komplikovanější než *rectangleSurfaceSum*. Kódování podmínek je poměrně složité a je pravděpodobné, že aplikace genetických operátorů na chromozom naruší existující strukturu programu. Rozdíl složitosti programů můžeme dokonce do jisté míry kvantifikovat porovnáním délek bázových chromozomů.

Délka bázového chromozomu funkce *rectangleSurfaceSum* je 16 genů. Oproti tomu odpovídající počet genů u *maxFromThree* je roven 46. Můžeme tedy předpokládat, že oprava v tomto případě nebude probíhat tak hladce.

Při testování bylo použito stejné nastavení GE jako v předchozích příkladech. Pro zakódování hledané funkcionality byla použita automaticky vygenerovaná testovací sada čítající 200 testovacích případů.

6.4.1 Opravené chyby

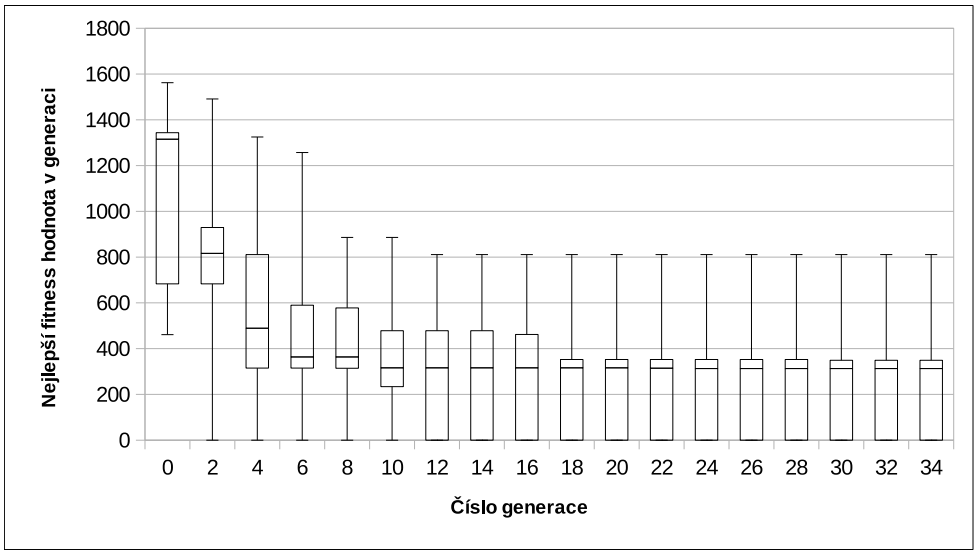
Během testování opravy triviálních chyb byla populace o 100 jedincích dostačující k dosažení 100% úspěšnosti. Některé z běhů ale pro nalezení řešení již vyžadovaly desítky generací. Výsledky tedy stále byly pozitivní, nicméně už nebyly natolik jednoznačné, jako tomu bylo v případě *rectangleSurfaceSum*.

Při opravování dvojité chyby byl použit následující vstupní program z obrázku 6.11:

```
int maxFromThree(int a,int b,int c){
    if(a>b){
        if(a>c){
            return b;                    <<< 'a' nahrazeno za 'b'
        }
        else {
            return c;
        }
    }
    else {
        if(b>c){
            return b;
        }
        else {
            return a;                    <<< 'c' nahrazeno za 'a'
        }
    }
}
```

Obrázek 6.11: Funkce *maxFromThree* s dvojitou chybou.

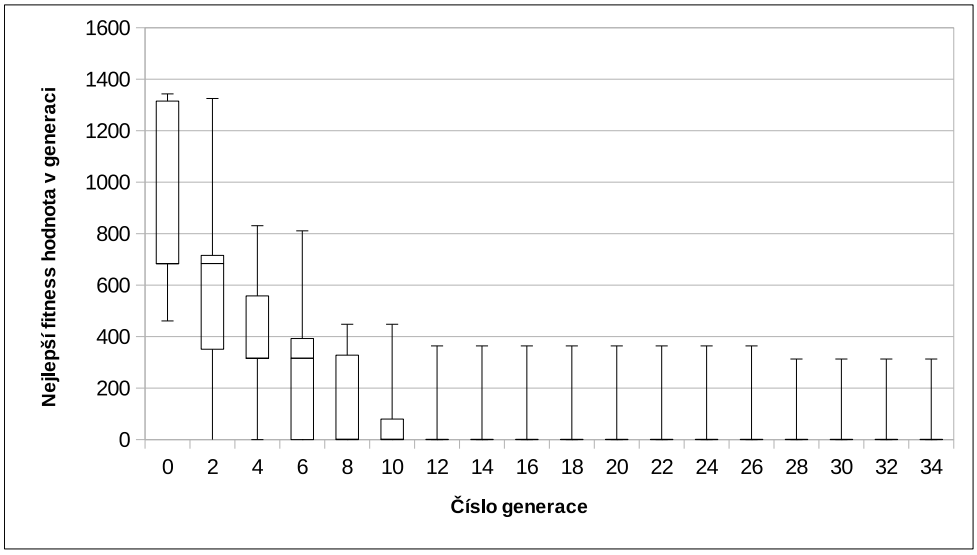
V obrázku 6.12 je zobrazen vývoj nejlepších fitness hodnot z dvaceti testovacích běhů při populaci velikosti 100.



Obrázek 6.12: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 100, při opravování dvojité chyby v funkci *maxFromThree*.

Z výsledků měření je zřejmé, že 100 jedinců je pro opravu této funkce málo. Řešení odhalilo pouze 9 z 20 běhů, všechny ostatní se přibližně mezi 12. a 30. generací ustálily v lokálním optimu a až do konce u nich nedošlo k další změně.

V obrázku 6.13 je zobrazen vývoj nejlepších fitness hodnot z dvaceti testovacích běhů při populaci velikosti 200.



Obrázek 6.13: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 200, při opravování dvojité chyby v funkci *maxFromThree*.

Z grafu měření, při populaci o velikosti 200 jedinců, jde snadno přečíst výrazné zlepšení úspěšnosti opravy. V tomto případě bylo řešení nalezeno při 18 z 20 běhů. Opravu dvojité chyby v programu založeném na síti podmínek můžeme tedy prohlásit za úspěšnou.

V době testování bohužel nebyly zaznamenávány údaje o počtech invalidních programů v populaci. Dodatečné spuštění evoluce ale jednoznačně potvrdilo přepoklad o častém narušování struktury původního programu. Přibližně polovina jedinců v každé generaci vykazovala chybu při překladu nebo již při sestavování AST z chromozomu.

Oproti tomu při testování funkce *rectangleSurfaceSum* bylo množství vadných jedinců pouze v rozmezí 10-20%.

6.5 Násobení pomocí cyklu

Teoretická složitost funkce *forMul* je nižší než je tomu u *maxFromThree*. To můžeme odvodit z délky jejího bázového chromozomu, která je rovna 36. Přesto je potřeba vzít v úvahu přítomnost cyklu v implementaci.

Složitost kódování cyklu je ještě větší, než je tomu v případě podmínky. Iterativní výpočet v průběhu evaluace jedince je další komplikací, která může znesnadnit konvergenci k globálnímu optimu. Očekávaným chování je opět časté narušování původní struktury funkce a tím pádem vyšší poměr vadných jedinců v populaci.

Při testování bylo použito stejné nastavení GE jako v předchozích příkladech. Pro zakódování hledané funkcionality byla použita automaticky vygenerovaná testovací sada čítající 200 testovacích případů.

6.5.1 Opravené chyby

Automatická oprava triviálních chyb při populaci o 100 jedincích vykazovala podobné výsledky jako v případě funkce *maxFromThree*. Řešení triviální chyby bylo nalezeno převážně během první evaluace. Toto chování můžeme odvodit z principu opravy triviální chyby. Ta je

závislá zejména na vhodné mutaci a pravděpodobnost této vhodné mutace klesá s rostoucí délkou chromozomu.

Pro další ověřování byla použita následující modifikace funkce *forMul* zobrazená v 6.14:

```
int forMul(int a, int b, int c){
    for(int i = 0;i<=a;i++){          <<< '<' nahrazeno za '<='
       sum += a+c;                    <<< 'b' nahrazeno za 'a'
    }
    return sum;
}
```

Obrázek 6.14: Funkce *forMul* s dvojitou chybou.

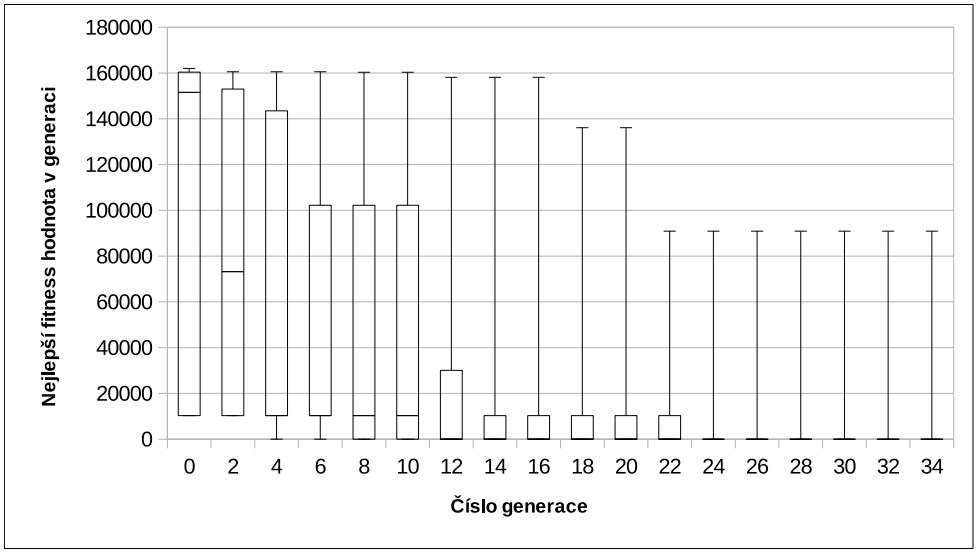
V programu byl nahrazen operátor $<$ za $<=$. Většina programátorů se určitě shodne v tom, že podobnou chybu v ukončovací podmínce cyklu *for* zažili již mnohokrát. Druhou modifikací byla změna výpočtu v těle cyklu z původního:

$$sum+ = b + c$$

na chybnou verzi:

$$sum+ = a + c$$

První test probíhal s populací čítající 100 jedinců, viz 6.15.



Obrázek 6.15: Krabicový graf zobrazující nejlepší fitness hodnoty generace při velikosti populace 100, při opravování dvojité chyby v funkci *forMul*.

Výsledky měření prokázaly, že oprava popsané dvojité chyby je úspěšná, i v případě, že opravovaná funkce je impementovaná využitím cyklu. Pouze dva z dvaceti běhů nedokázaly odhalit opravu před dosažením daném limitu 100 generací.

Podobně jako u *maxFromThree*, i v tomto případě bohužel nebyly zaznamenávány údaje o počtech vadných jedinců. Z dodatečného měření ale můžeme říct, že chybu při překladu nebo při dekódování chromozomu na AST ve většině případů vykazovalo 30-50% jedinců.

Poměr vadných jedinců byl tedy znatelně menší než tomu bylo v případě *maxFromThree*. Je ale potřeba brát v úvahu rozdíl v délce bázových chromozomů zmíněných funkcí a také to, že v rámci *forMul* byl implementován pouze jeden cyklus, zatímco v *maxFromThree* byly tři různé podmínky.

Pro to, abychom potvrdili předpoklad o citlivosti cyklického výpočtu vůči aplikaci genetických operátorů, by bohužel bylo nutné provést měření na funkcích, jejichž složitost by byla více vyrovnaná.

Kapitola 7

Závěr

Teoretická část této práce nabídla obecný úvod do evolučního počítání. Byly popsány některé ze současných přístupů k opravě softwaru a principy, na kterých je oprava nejčastěji postavena, a důvody, proč je vhodné se automatickou opravou softwaru zabývat.

Pomocí porovnání s existující aplikací GenProg byl podrobně popsán jeden z možných přístupů k automatické opravě softwaru využívající gramatickou evoluci. Mimo jiné byly vysvětleny pro nás nejdůležitější rozdíly v použití lineárního chromozomu oproti stromově strukturovanému chromozomu.

Pátá kapitola byla zaměřená na popis návrhu a implementace aplikace využívající dříve zmíněných principů. Její začátek se věnuje výběru implementačních nástrojů. Ostatní sekce popisují hrubý návrh aplikace a poté konkrétní programové řešení jednotlivých problémů.

Poslední kapitola se věnovala experimentálnímu ověření dosažených výsledků. Nejdříve byl zdokumentován postup ověřování funkcionality implementované gramatické evoluce. Na příkladu hledání jednoduchých matematických výrazů s využitím symbolické regrese prokazatelně konverguje ke globálnímu optimu. Můžeme tedy prohlásit, že GE pracuje správně.

Další oddíly se zaměřily na popis testování automatické opravy chyb v rámci vybraných programů. Naměřené výsledky ukázaly, že aplikace je schopná efektivně opravovat jednoduché chyby v různých typech programů.

V případě, že se jednalo o chyby triviálního charakteru, byla oprava ve většině případů nalezena již během první evaluace. Ostatní testy byly zaměřené na různé druhy dvojitých chyb. Aplikace prokázala schopnost opravovat takové chyby v relativně nízkém počtu generací.

Poznatky z testů automatické opravy potvrdily očekávané chování aplikace. Projevil se velký vliv inicializace počáteční populace založené na použití bázového chromozomu. Pomocí zmíněné inicializace bylo docíleno vhodného "nasměrování"evoluce a značného urychlení hledání opravy. Další výhodou byla výrazně menší velikost populace nutná pro nalezení opravy.

Implementovaná aplikace bohužel nepodporuje plně automatizovanou opravu chyb. K dosažení plně automatizované opravy je nutné napřed realizovat automatizovanou lokalizaci chyb. Ve čtvrté kapitole bylo popsáno, proč je lokalizace chyby v AST komplikovaná, pokud je využito lineární kódování chromozomu. Z těchto důvodů implementovaná automatická oprava problém lokalizace ponechává na uživateli.

Logickým krokem při dalším vývoji aplikace by tedy byla právě implementace automatické lokalizace chyb. Testy popsané v šesté kapitole probíhaly na krátkých úsecích zdrojového kódu. Předpokladem automatické lokalizace je totiž právě to, že ve vstupním programu

odhalí pokud možno co nejkratší úsek chybného kódu, nad kterým bude posléze probíhat implementovaná automatická oprava.

Další prostor ke zlepšení nabízí optimalizace rychlosti evaluace. Například by bylo vhodné změnit způsob formátování výstupů testovaných programů. Princip, na kterém je použité formátování založeno, totiž neumožňuje evaluaci více testovacích případu současně. Paralelizace je tedy povolena pouze mezi jedinci populace.

Literatura

[1] clang: a C language family frontend for LLVM. http://clang.llvm.org/, accessed: 2017-01-10.

[2] GCC, the GNU Compiler Collection. https://gcc.gnu.org/, accessed: 2017-01-10.

[3] pycparser, parser for the C language. https://github.com/eliben/pycparser, accessed: 2017-05-15.

[4] subprocess, python module for subprocess management. https://docs.python.org/2/library/subprocess.html, accessed: 2017-05-15.

[5] Kočí, R.; Křena, B.: *Úvod do softwarového inženýrství*. FIT VUT v Brně, 2013.

[6] Kubalík, J.: *Gramatická evoluce: Studijní materiál k předmětu XP33ECD*. FELK CVUT v Praze, 2016, accessed: 2017-01-10.
URL http://labe.felk.cvut.cz/vyuka/XP33ECD/Gramatick%e1%20evoluce.doc

[7] Langdon, W. B.; Harman, M.: Optimizing Existing Software With Genetic Programming. *IEEE Transactions on Evolutionary Computation*, ročník 19, č. 1, Feb 2015: s. 118–135, ISSN 1089-778X, doi:10.1109/TEVC.2013.2281544.

[8] Lourenço, N.; Pereira, F. B.; Costa, E.: Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines*, ročník 17, č. 3, 2016: s. 251–289, ISSN 1573-7632, doi:10.1007/s10710-015-9262-4.
URL http://dx.doi.org/10.1007/s10710-015-9262-4

[9] Lukáš Sekanina, e. a.: *Evoluční hardware*. Praha: Academia, 2009, ISBN 978-80-200-1729-1.

[10] McKay, R. I.; Hoai, N. X.; Whigham, P. A.; aj.: Grammar-based Genetic Programming: a survey. *Genetic Programming and Evolvable Machines*, ročník 11, č. 3, 2010: s. 365–396, ISSN 1573-7632, doi:10.1007/s10710-010-9109-y.
URL http://dx.doi.org/10.1007/s10710-010-9109-y

[11] Monperrus, M.: Automatic Software Repair: a Bibliography. 2015.
URL http://www.monperrus.net/martin/survey-automatic-repair.pdf

[12] Noorian, F.; de Silva, A. M.; Leong, P. H.: Grammatical Evolution: A Tutorial using gramEvol. 2015.

[13] O'Neill, M.; Ryan, C.; Keijzer, M.; aj.: Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*, ročník 4, č. 1, 2003: s. 67–93, ISSN 1573-7632,

doi:10.1023/A:1021877127167.
URL http://dx.doi.org/10.1023/A:1021877127167

[14] Schwarz, J.: *Aplikované evoluční algoritmy*. FIT VUT v Brně, 2014.

[15] Schwarz, J.; Sekanina, L.: Aplikované evoluční algoritmy EVO. 2006.

[16] Sekanina, L.: *Biologií inspirované počítače - Evoluční design*. FIT VUT v Brně, 2014.

[17] Weimer, W.; Forrest, S.; Le Goues, C.; aj.: Automatic Program Repair with Evolutionary Computation. *Commun. ACM*, ročník 53, č. 5, Květen 2010: s. 109–116, ISSN 0001-0782, doi:10.1145/1735223.1735249.
URL http://doi.acm.org/10.1145/1735223.1735249

[18] Wikipedia, the free encyclopedia: Genetic programming program example. 2007, [Online; accessed January 8, 2017].
URL https://en.wikipedia.org/wiki/Genetic_programming#/media/File:Genetic_Program_Tree.png

[19] Češka, M.: *Teoretická informatika - Jazyky a jejich reprezentace, algebra formálních jazyků*. FIT VUT v Brně, 2014.

Příloha A

Obsah přiloženého CD

- **report.pdf** – písemná zpráva
- **tex/** – zdrojový tvar písemné zprávy
- **grammar.txt** – gramatika použitá při implementaci aplikace
- **testResults.ods** – sešit aplikace Libre Office Calc s daty naměřenými během experimentů
- **usedData/**
  - 200surfsum.ts – testovací sada použitá při testování funkce *rectangleSurfaceSum*
  - 200maxfromthree.ts – testovací sada použitá při testování funkce *maxFromThree*
  - 200forMul.ts – testovací sada použitá při testování funkce *forMul*
  - test.c – zdrojový kód s implementací všech testovaných funkcí
- **application/**
  - main.py
  - gramEvol.py
  - astAnalyser.py
  - astGenerator.py
  - chromozomeGenerator.py
  - chromozomeHandler.py
  - testSetGen.py
  - compilerTester.py

Příloha B

Manuál

Aplikace je implementovaná v pythonu. Testování aplikace probíhalo na verzi 2.7.6. Další prerekvizitou je instalace pycparseru. Pokyny k instalaci jsou popsané v https://github.com/eliben/pycparser#installing.

V případě problémů s importem modulů je možné implementaci pycparseru stáhnout z https://github.com/eliben/pycparser a adresář se implementací aplikace *application* umístit do adresáře *pycparser-master*.

Nyní je možné aplikaci spustit pomocí:
python main.py

Pro spuštění opravy aplikace vyžaduje tři parametry.

1. **sourceFile.c** – jméno souboru s opravovaným zdrojovým kódem
2. **funcionName** – jméno opravované funkce
3. **testSet.ts** – jméno souboru s testovací sadou
4. **__repairDataSuffix** – volitelný parametr s koncovkou souboru *__repairData*

Pozor, aplikace vyžaduje zadání argumentů v popsaném pořadí.

Příklad spuštění automatické opravy:
python main.py ./usedData/test.c maxFromThree ./usedData/ 200maxfromthree.ts

Pokud uživatel potřebuje modifikovat konfiguraci opravy, je možné toho dosáhnout ručním nastavením nových hodnot v souboru *main.py*. Tento zdrojový kód obsahuje implementaci modulu *AutomaticRepair*, v jehož metodě *Start* je možné požadované modifikace realizovat. Popis jednotlivých parametrů se nachází v 5.13.

Aplikace při běhu gramatické evoluce stručně informuje uživatele o dosavadním průběhu opravy. Po ukončení výpočtu je vypsán přehled nejlepších nalezených řešení. Zdrojové kódy vygenerovaných jedinců jsou uloženy v rámci adresářové struktury *__repairData/*.