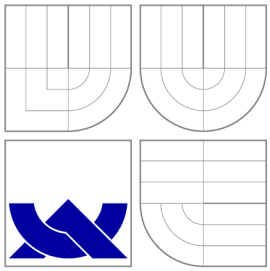
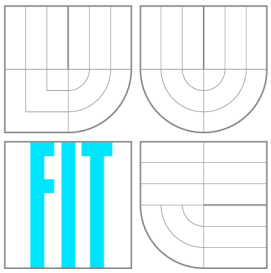


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENETICKÉ VYLEPŠENÍ SOFTWARE PRO KARTÉZSKÉ GENETICKÉ PROGRAMOVÁNÍ

GENETIC IMPROVEMENT OF CARTHESIAN GENETIC PROGRAMMING SOFTWARE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB HUSA

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. LUKÁŠ SEKANINA, Ph.D.

BRNO 2016

Abstrakt

Genetické programování je přírodou inspirovaná metoda programování umožňující automatizovaně vytvářet a adaptovat programy. Již téměř dvacet let je tato metoda schopna poskytovat výsledky porovnatelné s těmi vytvořenými člověkem, a to napříč mnoha obory. Tato práce čtenáře seznamuje s problematikou evolučních algoritmů, genetického programování a způsobů, jakými mohou být použity pro vylepšení stávajícího software. Dále je navržen program, který je touto metodou schopen vylepšit implementaci kartézského genetického programování (CGP). Program je poté otestován na implementaci CGP vytvořené pro potřeby tohoto projektu, a jeho funkčnost je dále ověřena i na převzatých již existujících implementacích CGP.

Abstract

Genetic programming is a nature-inspired method of programming that allows an automated creation and adaptation of programs. For nearly two decades, this method has been able to provide human-comparable results across many fields. This work gives an introduction to the problems of evolutionary algorithms, genetic programming and the way they can be used to improve already existing software. This work then proposes a program able to use these methods to improve an implementation of cartesian genetic programming (CGP). This program is then tested on a CGP implementation created specifically for this project, and its functionality is then verified on other already existing implementations of CGP.

Klíčová slova

evoluce, genetické programování, kartézské genetické programování, genetické vylepšování, symbolická regrese.

Keywords

evolution, genetic programming, cartesian genetic programming, genetic improvement, symbolic regression.

Citace

HUSA, Jakub. *Genetické vylepšení software pro kartézské genetické programování*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Sekanina Lukáš.

Genetické vylepšení software pro kartézské genetické programování

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.

.....
Jakub Husa
1. června 2016

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu práce panu prof. Ing. Lukáši Sekaninovi Ph.D. za jeho vedení, rady a velkorysou ochotu věnovat mi svůj čas při tvorbě mojí diplomové práce.

Obsah

Kapitola 1

Úvod

Programování je tradičně čistě lidskou disciplínou. Člověk-programátor se snaží porozumět problému, rozložit jej na jeho nejzákladnější části, a řešení těchto částí pak implementovat ve zvoleném programovacím jazyce. Cílem této snahy je pak nejčastěji automatizování nějaké dříve ručně prováděné činnosti. Tento velmi dobře ozkoušený způsob tvorby programů má však také řadu nevýhod. Programátor musí být vyškolen v technikách programování, dekompozice problémů, algoritmizace činností, a v používání nejméně jednoho programovacího jazyka. Než programátor může s tvorbou jakéhokoliv programu začít, musí být také seznámen s podrobnostmi problematiky, která má být programem řešena. Protože řešené problémy jsou zřídkakdy triviální, programátor musí na základě svých znalostí a schopností celý problém rozložit na řadu dílčích samostatně řešitelných problémů, opakovaně tak dlouho, dokud jednotlivé problémy nejsou dostatečně jednoduché na to, aby mohly být přímo implementovány ve zvoleném programovacím jazyce. Celý tento proces je velmi sofistikovaný a mentálně náročný.

Existují však i jiné způsoby, jakými lze programy tvořit. Jednou z možných alternativ, která se snaží překonat nedostatky tradičního přístupu, je genetické programování. Místo toho, aby se přístup pokoušel o dekompozici problému metodou shora-dolů jako člověk, genetické programování řeší problém opačným způsobem, tedy zdola-nahoru. Začíná s množinou příkazů nějakého jazyka a snaží se je uspořádat, nejčastěji do podoby syntaktického stromu, takovým způsobem, aby vykonání výsledného programu přeměnilo poskytnuté vstupy na požadované výstupy.

První výhodou této relativně nové metody programování je, že nevyžaduje detailní znalost řešeného problému. Druhou výhodou je, že program není tvořen po částech, ale celý současně, což umožňuje lepší integraci jednotlivých částí programu a tvorbu efektivnějšího kódu. Genetické programování má ale také řadu nevýhod. Pokud problém nemůže být ze své podstaty vyřešen relativně krátkým kódem, je počet možných způsobů, jakými lze příkazy programovacího jazyka uspořádat jednoduše příliš velký na to, aby mohl být se současnými technologiemi pomocí genetického programování v rozumném časovém horizontu prozkoumán. Možnosti praktického využití genetického programování jsou tedy omezeny na hledání řešení relativně jednoduchých úloh, které mají nejčastěji charakter symbolické regrese.

Nedostatky obou těchto přístupů však lze překonat použitím programovací techniky, která je propojuje, nazývané *genetické vylepšování*. Při tvorbě programů genetickým vylepšováním je problém nejprve vyřešen programátorem, který vytvoří program tradičním způsobem. Na tento program je pak aplikováno genetické programování, které se pokusí originální implementaci optimalizovat. Protože genetické programování je aplikováno na již funkční kód, lze ho použít i na velmi rozsáhlé programy, kde je klasické genetické pro-

gramování nepoužitelné. Tento postup je sice náročnější než přímé použití obou metod samostatně, ale umožňuje ve výsledku zachovat výhody obou metod.

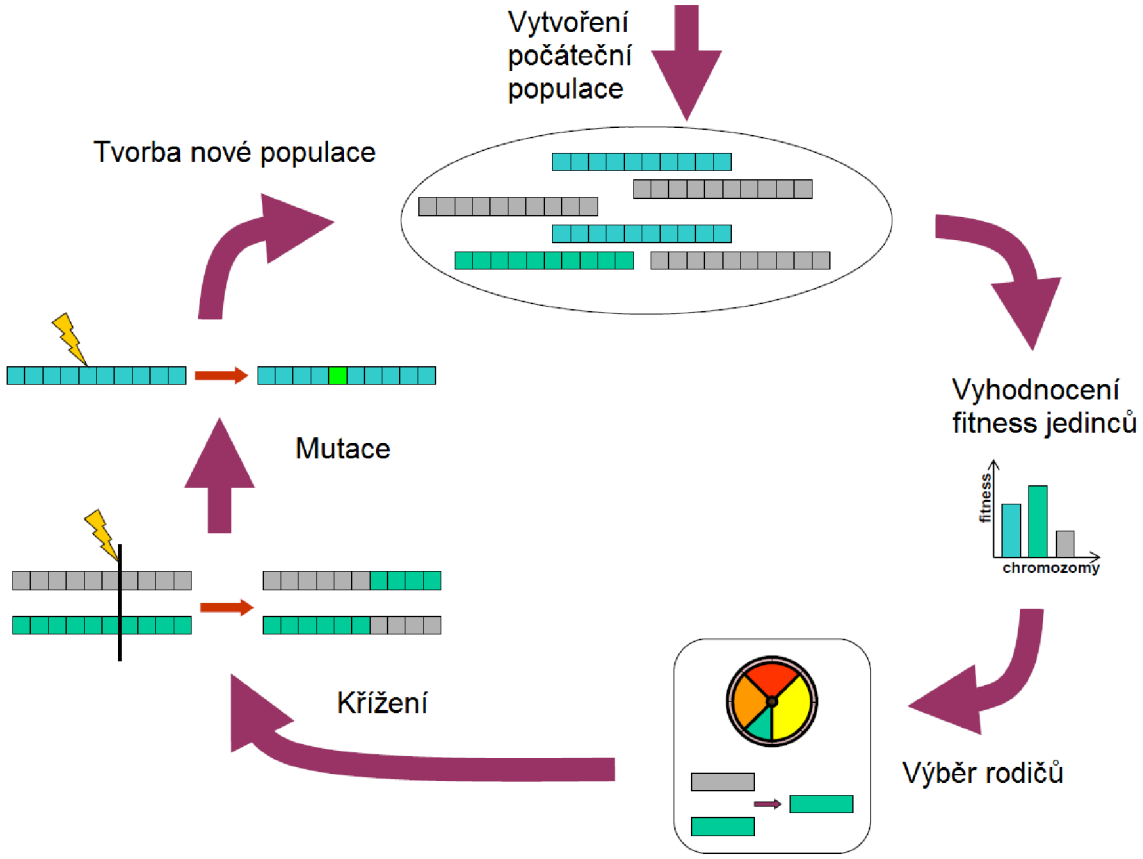
Prvním cílem této diplomové práce je čtenáře seznámit s problematikou evolučních algoritmů včetně jejich různých variant, jako je například kartézské genetické programování. Druhým cílem práce je navrhnout, implementovat a otestovat program, který metodou genetického vylepšování bude schopen vylepšit implementaci programu provádějícího kartézské genetické programování.

Tělo práce je členěno do kapitol. Úvod je následován kapitolou 2, která čtenáře seznámí se základy evolučních algoritmů a jejich využitím pro prohledávání stavového prostoru. Kapitola 3 se zaměřuje na genetické programování jako jedno ze specifických využití evolučních algoritmů. V kapitole 4 je podrobněji popsán specifický typ genetického programování, zvaný kartézské genetické programování. Kapitola 5 se věnuje genetickému vylepšování stávajícího software a uvádí konkrétní příklady jeho použití. Kapitola 6 popisuje návrh programu, který bude geneticky vylepšovat stávající implementaci kartézského genetického programování. Kapitola 7 popisuje způsob, jakým byla provedena implementace programu pro kartézské genetické programování a optimalizátoru, jehož účelem bylo tuto implementaci evolučně vylepšit. Testy obou programů jsou uvedeny v kapitole 8, ve které jsou dále popsány i konkrétní modifikace, které optimalizátor v originální implementaci provedl. V rámci této kapitoly je pak optimalizátor vyzkoušen i na jiných implementacích vstupního programu. V kapitole 9 se nachází závěr práce, který shrnuje všechny dosažené výsledky a zamýšlí se nad tím, jak by na práci šlo v budoucnu navázat. Na konci práce jsou pak uvedeny odkazy na použitou literaturu a přílohy.

Kapitola 2

Evoluční algoritmy

Evoluční algoritmy jsou heuristickou metodou prohledávání stavového prostoru. Od druhé poloviny padesátých let minulého století existovaly pokusy o využití poznatků přírodních věd a napodobení evolučního procesu pro potřeby programování [17]. Nezávisle na sobě potom vzniklo několik technik, jako jsou genetické algoritmy, evoluční strategie, evoluční programování a genetické programování. Tyto metody byly v devadesátých letech spojeny a zastřešeny pod společným názvem evoluční algoritmy [22].



Obrázek 2.1: Obecné schéma evolučního cyklu [22].

Celá myšlenka evolučních algoritmů vychází z Darwinovy evoluční teorie o vývoji druhů a různých teorií neodarwinismu. Organismy ve volné přírodě řeší řadu problémů. Jak sehnat potravu, jak se nebýt uloven predátorem, či jak najít partnera pro rozmnožování. Schopnost každého živočicha tyto a další problémy řešit rozhoduje nejen o přežití jeho samotného, ale i celého jeho druhu. Jedinci, kteří tyto problémy řeší hůře, jsou v nevýhodě. Ti lepší naopak dosáhnou zplození potomků a předání úspěšnějšího genové konfigurace další generaci. V důsledku přirozeného výběru tak dochází k vývoji celého živočišného druhu, který se stále lépe přizpůsobuje svému prostředí [23].

Tento princip je možné použít pro konstrukci optimalizačního algoritmu pracujícího na základě opakovaného vyhodnocování a křížení populace kandidátních řešení. Základní kroky tohoto algoritmu, který bude popsán ve zbytku této kapitoly, ukazuje obrázek 2.1.

2.1 Pojmy

V rámci evolučních algoritmů se vyskytuje řada pojmů, pocházejících původně z přírodních věd, které jsou však pro tyto účely používány v mírně odlišném kontextu [1].

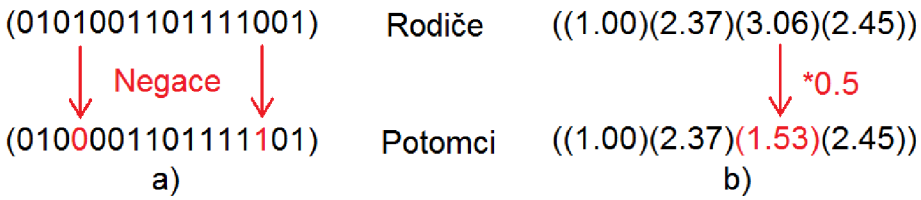
- **Jedinec** – Jedinec v evolučním algoritmu představuje jedno konkrétní kandidátní řešení zkoumaného problému.
- **Populace** – Populace je množinou jedinců obvykle o konstantní velikosti. Používání celé populace místo samotných jedinců, přináší do algoritmu paralelizaci řešení a možnost kombinování zkoumaných řešení, čímž se odlišuje od většiny ostatních algoritmů pro prohledávání stavového prostoru.
- **Chromozom** – Chromozom je zápis jednoho konkrétního kandidátního řešení zkoumaného problému ve formátu, se kterým pracuje evoluční algoritmus. V praxi může jít například o textový řetězec, stromovou strukturu, či binární nebo číselný vektor. V závislosti na řešeném problému může být délka chromozomu buď fixní, nebo proměnlivá.
- **Gen** – Geny jsou základními stavebními jednotkami chromozomů. Každý gen má definovanou množinu možných hodnot. Konkrétní hodnota genu se nazývá *alela*.
- **Genotyp** – Genotyp označuje množinu všech genů jedince.
- **Fenotyp** – Zatímco genotyp představuje zakódované řešení, fenotyp je celkové kandidátní řešení zkoumaného problému. Fenotyp tak není obecně závislý na způsobu reprezentace chromozomu. Chromozom může být výrazně větší, být uložen v jiném formátů, či obsahovat řadu informací, které pro tvorbu výsledného fenotypu vůbec nemusí být použity.

2.2 Genetické operátory

Genetické operátory slouží pro vytváření nových kandidátních řešení. Pracují na úrovni chromozomů. Od standardních matematických operátorů se odlišují tím, že kladou důraz na respektování logiky řešeného problému. V závislosti na počtu jedinců, se kterými operátor pracuje, lze rozlišit dvě základní kategorie [15].

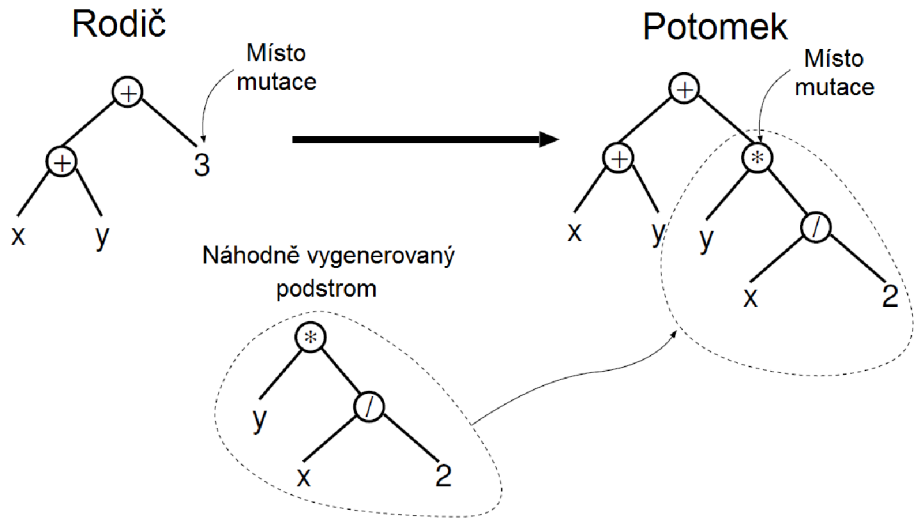
Operátory mutace

Operátory mutace pracují pouze s jedním jedincem ze kterého vytvářejí nového jedince provedením jedné nebo více náhodných změn. Pokud je chromozomem bitový vektor, je mutace prováděna negací k z jeho bitů. Pokud je chromozom číselný vektor, jsou některé z jeho hodnot nahrazeny nebo upraveny náhodně vygenerovanými čísly jak ukazuje obrázek 2.2.



Obrázek 2.2: Mutace chromozomů reprezentovaných a) binárním vektorem, b) vektorem reálných čísel.

U stromem reprezentovaných chromozomů mutace probíhá vybráním náhodného uzlu který se nahradí náhodně vygenerovaným podstromem jak ilustruje obrázek 2.3.

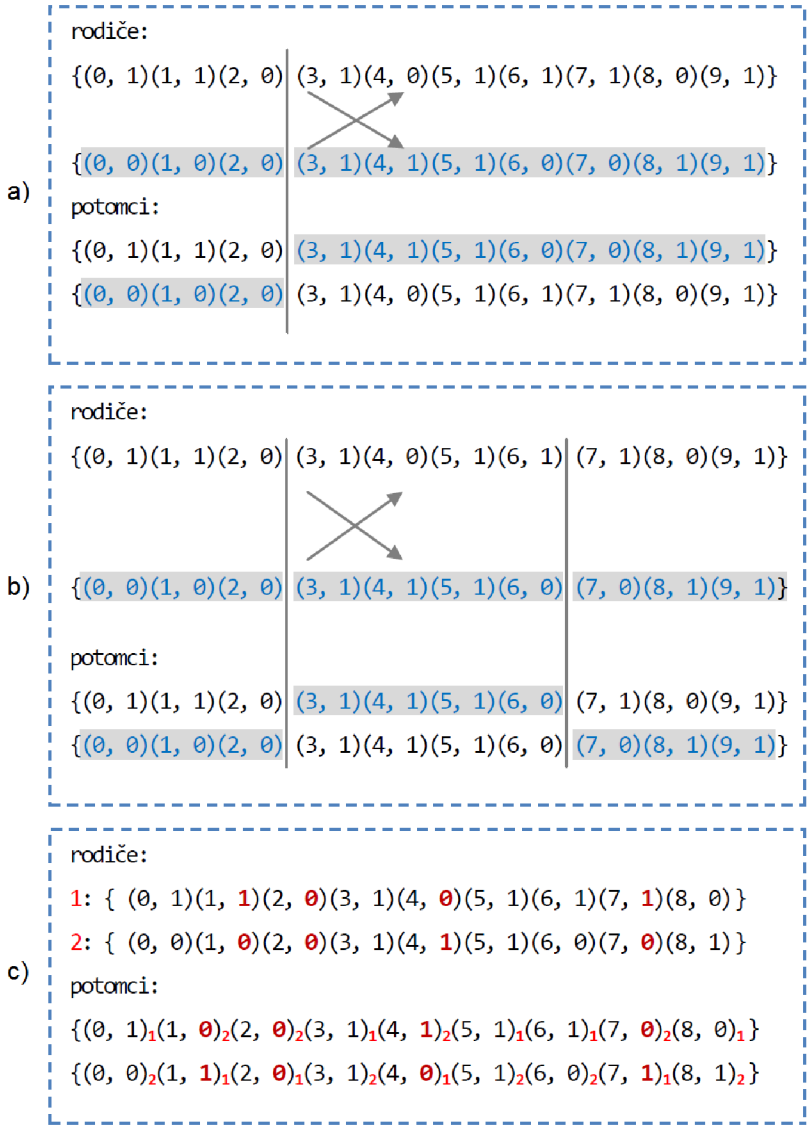


Obrázek 2.3: Mutace chromozomu reprezentovaného stromem [17].

Mutace přinášejí do chromozomů nové náhodné geny, zabraňují uváznutí populace v lokálním minimu prohledávaného prostoru a zaručují její diverzitu. Protože však jsou tyto změny náhodné a často velmi radikální, mohou také zničit funkční nalezené řešení. Aby nedocházelo k naprostému zničení genetické informace rodičů a dědičné rysy zůstaly v populaci zachovány, jsou mutace prováděny pouze s malou pravděpodobností. Určitá míra mutace však musí být součástí všech evolučních algoritmů.

Operátory křížení

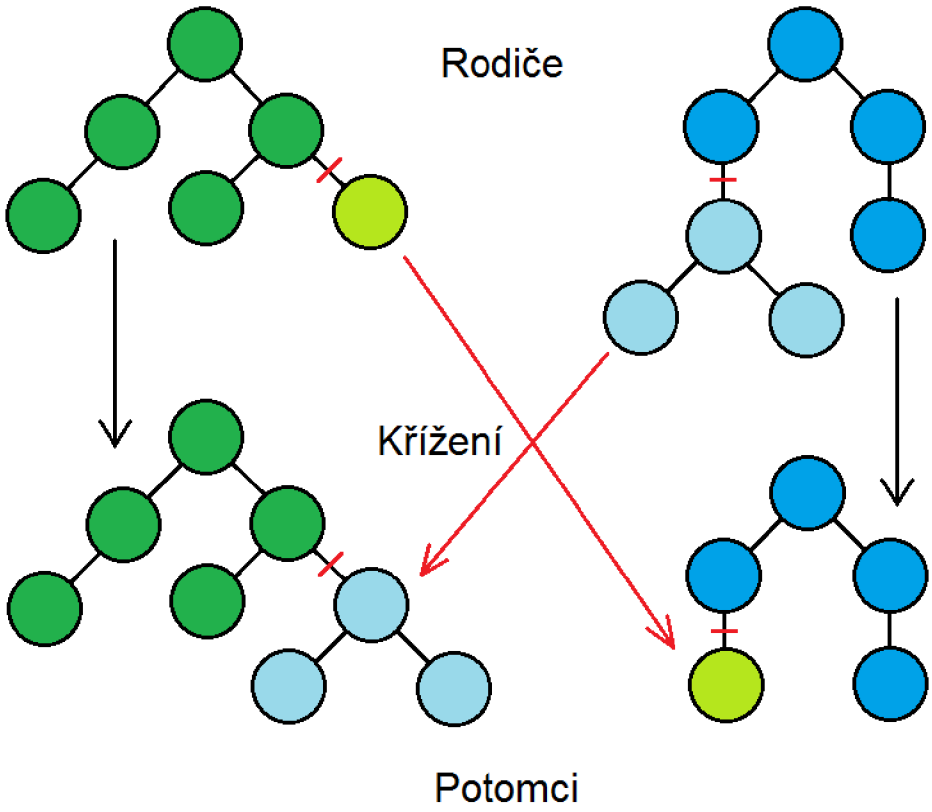
Operátory křížení tvoří nové potomky kombinováním typicky dvou již existujících jedinců. Způsob, jakým je křížení prováděno, závisí na způsobu reprezentace chromozomu a obvykle vytváří počet potomků shodný s počtem rodičů.



Obrázek 2.4: Tři základní přístupy ke křížení vektorů hodnot. a) Jednobodové křížení, b) vícebodové křížení, c) uniformní křížení [1].

Pokud je chromozom tvořen vektorem binárních hodnot, existují tři základní způsoby křížení: *jednobodové*, *vícebodové*, a *uniformní*. Jak ukazuje obrázek **??**, při jednobodovém křížení je nejprve v chromozomech rodičů určena pozice, kde má dojít ke křížení. Potomci poté převezmou část chromozomu před vybranou pozicí od jednoho rodiče, a zbytek chromozomu od rodiče druhého. Při vícebodovém křížení je takových pozic zvoleno několik, a potomci berou úseky střídavě od jednoho a druhého rodiče (vždy každý z jiného). Při uniformním křížení se v chromozomech rodičů nedochází k určování pozic a u každého genu je o použitém rodiči rozhodnuto samostatně.

Pokud je chromozom tvořen vektorem reálných čísel, používá se křížení buď *aritmetické*, nebo *heuristické*. Při aritmetickém křížení je vytvořen náhodný koeficient k, obvykle v rozsahu $[-0.25, 1.25]$, a geny potomka jsou vypočítány jako součet k násobku hodnoty genu jednoho rodiče a $1 - k$ násobku hodnoty genu rodiče druhého. Druhý potomek je spočten stejným způsobem, ale role obou rodičů jsou zaměněny. Důvodem, proč je rozsah koeficientu mimo očekávatelný rozsah $[0, 1]$ je, že čím více se hodnota parametru k blíží hodnotě 0.5, tím menší je mezi vytvořenými potomky rozdíl, a křížení se tak blíží průměrování, čímž dochází ke ztrátě diverzity populace. Při heuristickém křížení jsou oba potomci vytvořeni rozdílným způsobem. Jeden je vytvořen matematickou kombinací hodnot obou rodičů, například stejným způsobem jako u aritmetického křížení. Druhý je vytvořen pouhou kopií lepšího z rodičů. Který z rodičů je považován za lepšího je určeno na základě funkce fitness, která bude vysvětlena v následující podkapitole.



Obrázek 2.5: Ukázka křížení chromozomů reprezentovaných stromem.

Pokud je chromozom tvořen stromem, je u každého z rodičů vybrán jeden náhodný uzel, a potomci jsou vytvořeni výměnou těchto uzlů i se všemi jejich potomky, jak ukazuje obrázek 2.5. Pokud Strom reprezentuje například matematickou rovnici, může tak být jedno číslo nahrazeno nejen jinou hodnotu ale i celým dalším výpočtem.

Kromě již zmíněných způsobů křížení existují i další přístupy pro křížení chromozomů uložených v jiných formátech jako mohou být například textové řetězce, permutačních řetězce a další.

2.3 Evoluční cyklus

Evoluční algoritmy při prohledávání stavového prostoru pracují iterativně. Na jejich počátku je populace vytvořená náhodně, nebo na základě nějakého již existujícího řešení. Nad populací je poté opakován evoluční cyklus tak dlouho, dokud není nalezeno požadované řešení, nebylo provedeno požadované množství iterací, nebo dokud nevyprší čas, který byl pro běh evolučního algoritmu přidělen. Samotný evoluční cyklus pak sestává z pěti kroků, kterými jsou: vyhodnocení populace, výběr rodičů, křížení, mutace a tvorba nové populace.

Vyhodnocení populace

Aby mohl bát evoluční algoritmus použit pro hledání nejlepšího možného řešení, musí existovat způsob, jakým určit kvalitu řešení, které jedinci představují. K tomu je použita takzvaná funkce *fitness*, která každému jedinci přiřazuje jednoznačné číselné ohodnocení udávající jeho kvalitu. Jakým způsobem je fitness funkce konkrétně definována závisí na problému, který je evolučním algoritmem řešen. Musí však platit, že čím je řešení lepší, tím lepší hodnota fitness mu je fitness funkcí přidělena. Jestli je za lepší hodnotu fitness považována hodnota vyšší, nebo nižší, záleží na řešením problému.

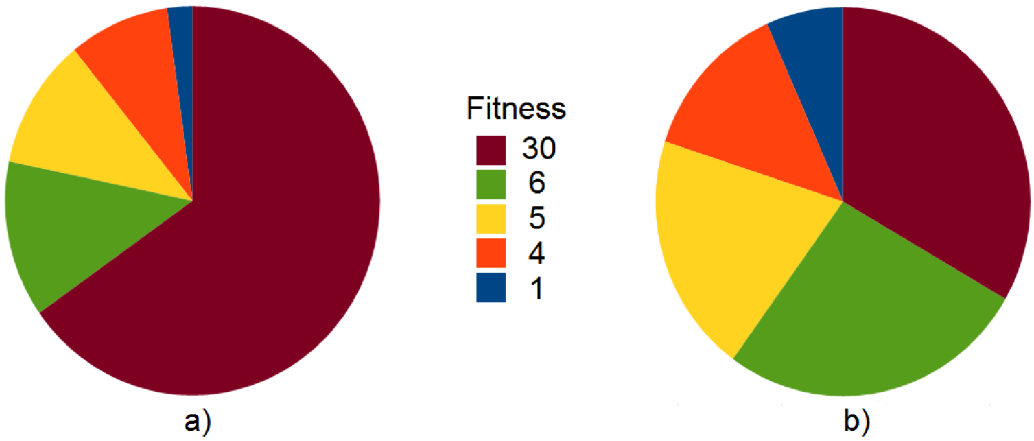
Pokud se evolučním algoritmem snažíme například navrhnout kombinační obvod, může být fitness funkce definována jako počet hodnot vstupů, na které daný jedinec poskytuje požadované výstupy. Kromě funkčnosti samotné je možné v ní zohlednit i řadu dalších parametrů. Například u již zmíněného kombinačního obvodu může být zohledněn počet hradel které daný obvod používá, jeho očekávaná energetická spotřeba, či zpoždění.

Ať už je fitness funkce definována jakkoliv, prvním krokem evolučního cyklu je určení hodnoty fitness všech jedinců populace. Tento krok bývá také označován termínem *evaluace*.

Výběr rodičů

Druhým krokem cyklu je výběr rodičů, někdy též zvaný *selekce*. Když je současná populace vyhodnocena, musí být proveden výběr jedinců, kteří se stanou rodiči generace následující. Při tomto výběru je zapotřebí najít rovnováhu mezi dvěma protichůdnými požadavky. Prvním je, aby byli preferováni jedinci s vyšší hodnotu fitness, protože představují lepší řešení zkoumaného problému a jejich geny jsou pro populaci přínosnější. Druhým požadavkem je, že výběr musí mít dostatečnou diverzitu. I jedinci s malou hodnotu fitness mohou ve svém chromozomu obsahovat velmi užitečné geny a jejich využití při křížení může vést ke vzniku jedinců s velmi vysokou fitness. Pokud by byli vybíráni pouze ti nejlepší z jedinců, budou si všichni jejich potomci vzájemně velmi podobní a algoritmus bude prohledávat pouze velmi malou část stavového prostoru možných řešení. Pokud by byl výběr prováděn zcela náhodně, algoritmus naopak nebude směřovat k lepším řešením.

Počet rodičů, kteří mají být vybráni je při jejich výběru obecně předem známý. Použitý algoritmus je opakován tak dlouho, dokud jich není vybrán dostatečný počet. Kolikrát může být jedinec z populace vybrán při tom není nijak omezeno, a každý jedinec může být rodičem celé řady potomků, se stejnými i různými partnery. Z množství algoritmů, které se snaží tyto dva požadavky uspokojit, jsou nejčastěji používány následující [15]:



Obrázek 2.6: Poměrná šance jedinců s rozdílnou fitness na to stát se rodičem při výběru a) ruletou, b) dle pořadí.

- **Ruleta** – Při výběru ruletou je fitness všech jedinců podělena sumou fitness celé populace. Výsledné hodnoty jsou seřazeny do rozsahu $[0, 1]$, ze kterého jsou pak rodiče "losováni" generováním náhodných čísel. Čím větší je fitness jedince, tím větší část intervalu mu připadne, a tím větší je šance, že bude ruletou vybrán. Jedinci s nižší fitness mají šanci na výběr nižší, ale stále mohou být vybráni. Nevýhodou tohoto přístupu je, že míra, s jakou jsou lepší jedinci preferováni před horšími, závisí na způsobu, jakým je fitness funkce definována. Pokud jsou rozdíly ve fitness jedinců příliš malé, budou všichni vybíráni s velmi podobnou pravděpodobností a algoritmus se bude k hledanému řešení blížit pouze velice pomalu. Pokud jsou rozdíly ve fitness naopak příliš velké, může být ve výsledku pro generování populace použita pouze velmi malá skupina rodičů a dojde ke ztrátě diverzity.

- **Výběr dle pořadí** – Tento algoritmus pracuje na velmi podobném principu jako výběr ruletou, ale zohledňuje jeho hlavní nedostatky. Rozsah $[0, 1]$ je rozdělen jiným způsobem. Místo toho aby byl rozdělen přímo podle hodnot fitness jedinců kteří populaci tvoří, je rozdělen na základě jejich počtu na několik stejně velkých dílů. Jedinci jsou podle svojí fitness seřazeni a tomu s nejnižší hodnotou fitness je přidělen jeden díl, druhému nejhoršímu jedinci jsou přiděleny díly dva, třetímu nejhoršímu tři, atd. Problém s rozdíly v hodnotách fitness je tímto smazán a lepší jedinci jsou vždy preferováni stejnou měrou. Porovnání metody výběru ruletou a výběru dle pořadí je ukázáno na obrázku **??**.

- **Turnaj** – Při výběru turnajem jsou z populace vybráni dva náhodní jedinci, jejich fitness je porovnána a ten lepší je použit jako rodič. Výhodou tohoto systému je jeho jednoduchost. Fitness jedinců není potřeba nijak přepočítávat ani řadit a stačí ji pouze umět porovnat. Přestože šance každého jedince na zúčastnění se turnaje jsou stejné, jedinci s lepší fitness mají větší šanci svého soupeře porazit a skutečně se stát rodičem.

- **Výběr elity** – při výběru elity je populace seřazena podle hodnot jejích fitness a jako rodiče příští generace jsou vybráni ti nejlepší jedinci. Protože tento způsob výběru

nesplňuje požadavky na diverzitu je používán pouze u těch evolučních algoritmů, které ji zaručují jiným způsobem. Příkladem takového algoritmu je například kartézské genetické programování, kterému je v této práci věnována kapitola 4.

Křížení

Když jsou rodiče vybráni, jsou z nich pomocí operátorů křížení vytvořeni potomci. Nejčastěji jsou spolu dvojice kříženy v tom pořadí, v jakém byly při výběru rodičů zvoleny. Pokud nejsou některé kombinace rodičů povoleny (například nechceme, aby se jedinec křížil sám se sebou), je způsob jejich výběru upraven tak, aby neplatným kombinacím rodičů předcházel, nebo jsou místo jednotlivců vybírány přímo platné dvojice.

Mutace

Jedinci vytvoření křížením jsou poté podrobeni mutaci. Množství prováděných mutací opět závisí na konkrétní aplikaci, ale v porovnání s křížením je obvykle velice malé. Výjimku v tomto případě tvoří kartézské genetické programování, u kterého je křížení zcela odstraněno a generování jedinců je prováděno pouze pomocí velkého množství mutací.

Tvorba nové populace

Po provedení mutací jsou noví jedinci připraveni stát se součástí nové generace. V závislosti na vzájemném počtu rodičů a potomků a řešeném problému existuje několik způsobů jak určit, kteří z rodičů a potomků se stanou členy nové generace.

- **Čisté nahrazení** – Pokud je rodičů i potomků stejný počet, bývají rodiče svými potomky zcela nahrazeni. Tento přístup také bývá označován jako *generační*.
- **Nahrazení elitou** – pokud je potomků méně než rodičů, je potomky nahrazena pouze část populace rodičů (ta s nejnižší fitness). Pokud například chceme, aby nejlepší dosud nalezený jedinec byl součástí každé nové populace, bude při křížení a mutaci vygenerováno o jednoho potomka méně, než je celková velikost populace, a nejlepší z rodičů v ní bude ponechán.
- **Rovnoměrné nahrazení** – Výběr mezi rodiči a jejich potomky může být proveden zcela náhodně, nebo může být vybrán náhodný počet jedinců z každé kategorie.
- **Nahrazení dle fitness** – Pokud je požadováno starou a novou populaci kombinovat, ale náhodný výběr není pro danou aplikaci vhodný, můžeme u potomků vyhodnotit jejich fitness a pro výběr jedinců, kteří vytvoří novou generaci, použít stejné metody jaké byly použity pro výběr rodičů. Množinou, ze které vybíráme, mohou být jak samotní potomci (pokud bylo vytvořeno více potomků, než je velikost populace), tak i smíšená skupina potomků a jejich rodičů.

Ukončení cyklu

Když je nová populace úspěšně vytvořena, je celý cyklus opakován znovu od začátku. Případně, pokud je splněna ukončovací podmínka, cyklus končí a výsledkem evolučního algoritmu se stává fenotyp jedince s nejlepší hodnotou fitness, která byla v rámci všech generací kdy nalezena. V závislosti na použitém evolučním algoritmu jím vždy může být jedinec s nejlepší fitness v poslední generaci, nebo libovolný jedinec který kdy byl evolucí vytvořen.

Kapitola 3

Genetické programování

Genetické programování vzniklo v 80. letech minulého století. Tato kapitola popisuje, jak se genetické programování liší od dříve uvedených evolučních algoritmů a jaká jsou specifika jeho využití.

Hlavním rozdílem mezi evolučními algoritmy a genetickým programováním je, že se genetické programování nezabývá optimalizací parametrů funkcí a programů, ale tvorbou programů samotných. Chromozomy jedinců tak představují spustitelné struktury, zapsané nejčastěji v podobě syntaktického stromu, textového řetězce nějakého programovacího jazyka, nebo matematického výrazu [4].

Genetické programování má řadu variant, jako jsou například: gramatická evoluce, lineární genetické programování a kartézské genetické programování. Pokud není uvedeno jinak, ve zbytku kapitoly předpokládáme nejobecnější variantu genetického programování, která reprezentuje kandidátní programy pomocí syntaktických stromů.

3.1 Přípravné kroky

Než může být genetické programování použito pro řešení nějakého problému, je třeba provést pět hlavních kroků, které spočívají v definici pěti komponent genetického programování [28][6].

Množina terminálů

V genetickém programování terminály představují listy syntaktického stromu. Může jít například o vstupy programu, konstanty nebo funkce bez dalších parametrů. Vstupy programu vždy musejí být známy a každému z nich je přiřazen jeden terminál. Množina konstant, které budou při tvorbě programu potřeba, naopak známa není. Tento problém se řeší vytvořením dostatečně velkého počtu konstant bez určeného sémantického významu, které pak program může využít. Množina funkcí bez dalších parametrů je zpravidla závislá na řešeném problému a nemáme možnost si ji vybrat. Příkladem takovéto funkce může být například přečtení uživatelem stisknuté klávesy.

Množina funkcí

Funkce představují vnitřní uzly syntaktického stromu a dají se rozdělit do tří kategorií. První jsou standardní konstrukce jazyka, ve kterém je program implementován, jako jsou cykly, skoky a větvení. Druhou kategorií jsou standardní aritmetické a logické operace tak, jak jsou běžně používány. Tyto funkce mimo jiné zajišťují, že program bude moci při svém běhu použít i jiné hodnoty než jsou uvedeny mezi konstantami množiny terminálů. Podobně jako u množiny terminálů jsou třetí skupinou funkce specifické pro řešený problém. Například načtení vstupů z i-tého senzoru robota.

Fitness funkce

Způsob, jakým je určena fitness, závisí na problému, který se snažíme vyřešit, a parametrech, na jejichž kvalitu u výsledného programu klademe důraz. Pro určení fitness jedince je jím představovaný program interpretován a vyzkoušen na určitých vstupních datech, pro která jsou známy správné výsledky. Dále se určí, pro kolik vstupů program správné výsledky poskytl, případně jak moc se ke správným výsledkům přiblížil. Dalšími faktory, které mohou být ve fitness funkci zohledněny, jsou doba vykonání programu a velikost chromozomu, ze kterého je odvozen.

Na rozdíl od chromozomů konstantní délky, obvykle používaných v ostatních evolučních algoritmech, velikost syntaktických stromů může s postupem času narůstat teoreticky až do nekonečna a způsobit problémy s nedostatkem paměti. Čím je syntaktický strom větší, tím déle také trvá jej vyhodnotit, což běh algoritmu ještě více zpomaluje. Tento problém nepřiměřeného nárůstu velikosti vytvářených stromů se nazývá *bloat*. Stromům, jejichž velikost překročí přijatelné meze, je proto žádoucí snížit fitness a omezit tím jejich propagaci do dalších generací. Další možností, jak redukovat velikost syntaktických stromů, je v syntaktických stromech průběžně vyhledávat a odstraňovat takzvané *introny*, zbytečné uzly, jejichž vyhodnocení nemá žádný vliv na výstupy programu [16].

Parametry běhu

Parametry běhu znamenají nastavení programu, který bude vývoj provádět. Nastavitelnými vlastnostmi může být například velikost vyvíjené populace, algoritmus výběru rodičů, množství prováděných mutací nebo počet generací. To, jaké nastavení parametrů je nejlepší, závisí na konkrétním řešeném problému.

Ukončující podmínka

Ukončující podmínkou pro genetické programování bývá nejčastěji vypršení přiděleného času, vyhodnocení určeného počtu generací nebo nalezení dostatečně dobrého řešení. Výstupem programu je pak fenotyp odvozený z chromozomu s nejvyšší fitness, jaká byla v rámci vývoje nalezena.

3.2 Cyklus genetického programování

Cyklus genetického programování je velmi podobný cyklu ostatních evolučních algoritmů. Počáteční populace je vytvořena náhodně. Jako kořen kandidátního stromu je zvolena nějaká funkce z množiny funkcí a za její parametry jsou náhodně zvoleny buď terminály nebo další funkce, kterým jsou pak obdobně přiděleny další parametry. V závislosti na požadavcích na počáteční populaci existují tři hlavní přístupy k jejímu vytvoření [4].

- **Full** – U metody *full* je určena požadovaná hloubka kandidátního stromu. Všechny uzly před touto hloubkou jsou náhodně vybrány z množiny funkcí a všechny uzly, které se v této hloubce nacházejí, jsou náhodně vybrány z množiny terminálů. Tato metoda vytváří stromy s pokročilou funkcionalitou, ale relativně malou variabilitou.
- **Grow** – U metody *Grow* je maximální hloubka, ve které jsou vždy použity terminály, definována také, ale ve všech předcházejících úrovních stromu jsou uzly vybírány z obou množin současně. Výhodou této metody je, že vytvořené stromy jsou si vzájemně velmi málo podobné. Nevýhodou je fakt, že mnoho takto vytvořených stromů bude příliš malých na to, aby měly zajímavou funkcionalitu.
- **Ramped Half-and-Half** – Při použití této metody je určena maximální hloubka a populace je rozdělena do několika stejně velkých skupin, kde je každé skupině určena její vlastní maximální hloubka v rozsahu od dvou, do maximální hloubky celé metody. Pro každou hloubku je pak polovina stromu generována metodou *Full* a druhá polovina metodou *Grow*. Tato metoda zaručuje počáteční populaci s dostatečnou diverzitou i rozmanitostí.

Vyhodnocení populace je prováděno dříve zvolenou funkcí fitness. Výběr rodičů je prováděn jedním ze způsobů zmíněných v kapitole o evolučních algoritmech. Křížení chromozomů je prováděno stejně jako u ostatních stromových struktur, v obou rodičích je zvolen jeden náhodný uzel, a tyto uzly spolu s jejich podstromy jsou mezi rodiči zaměněny, vytvářejíc dva nové stromy. Mutace stromu se typicky provádí zvolením jednoho uzlu a jeho nahrazením pomocí nového náhodně vygenerovaného podstromu. Výběr nové generace je proveden jedním ze způsobů popsaných u evolučních algoritmů.

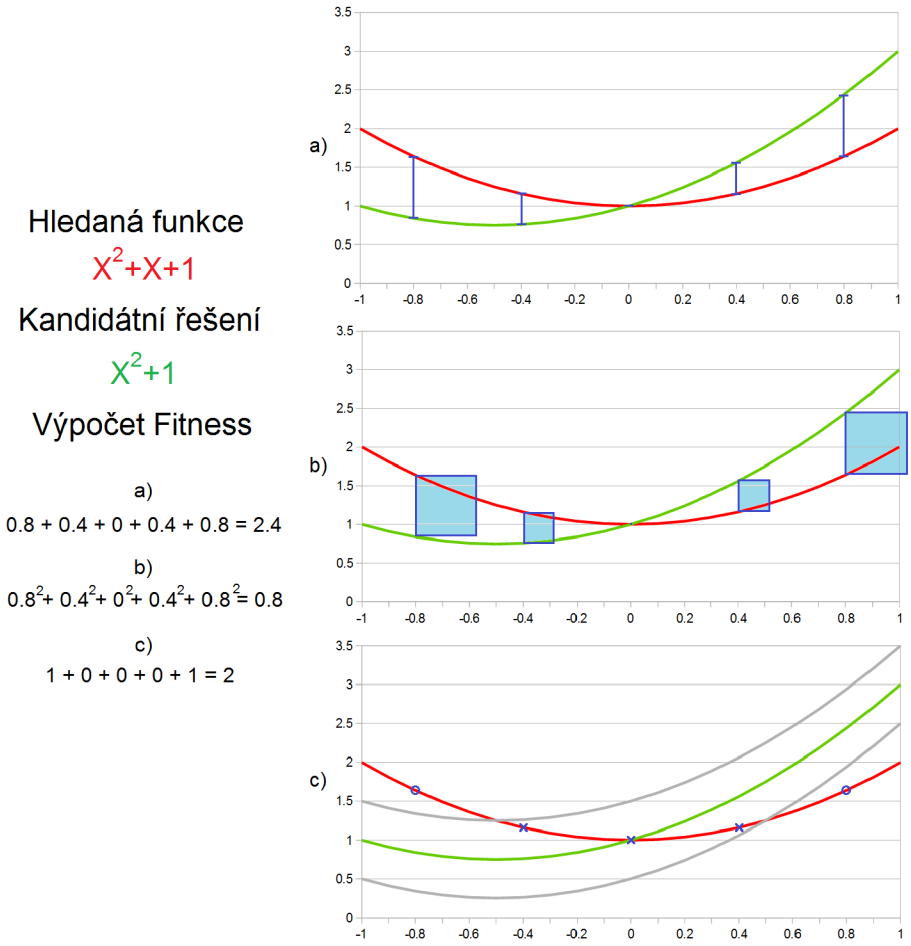
3.3 Možnosti využití

Možnosti využití genetického programování jsou široké. Kromě vylepšování software, kterému je věnována samostatná kapitola, je genetické programování možné využít také pro klasifikaci, vývoj herních strategií v ekonomii, zpracování obrazu v medicíně, návrh číslicových obvodů, a řady různých zařízení [23][3]. Za pomoci genetického programování byly vytvořeny desítky řešení schopných konkurovat řešením vyrobeným člověkem. Některé z nich byly dokonce i patentovány [5].

Symbolická regrese

Jednou z úloh, pro jejíž řešení je často využíváno genetického programování, je symbolická regrese. Jde o hledání výrazů, které by co nejlépe popisovaly vztah mezi hodnotami vzájemně závislých proměnných. Na rozdíl od standardních metod pro hledání výrazů které předpokládají určitý tvar řešení, nejsou v symbolické regresi žádná omezení tohoto typu kladena [19].

Vstupem symbolické regrese je posloupnost datových bodů a množina funkcí, které mají být pro sestavení hledaného výrazu použity. Výstupem je sestavená funkce, která se co nejméně odchyluje od hodnot určených hledanými body.



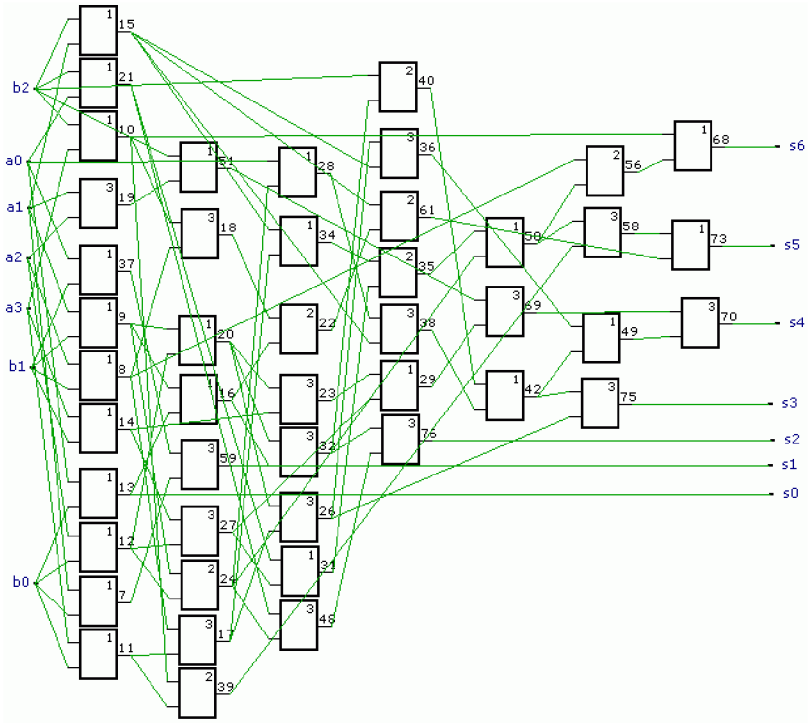
Obrázek 3.1: Tři způsoby výpočtu fitness u symbolické regrese pro referenční body *-0.8*, *-0.4*, *0*, *0.4* a *0.8* dle a) velikosti odchylek b) čtverců odchylek c) počtu dostatečných přiblížení.

Fitness funkci je možné u symbolické regrese určit několika způsoby podle toho, jaký reálný problém je symbolickou regresí řešen. Pokud aplikace vyžaduje zcela přesné nebo co nejpřesnější výsledky, bývá fitness definována jako suma všech odchylek, případně čtverců odchylek, požadovaných a vypočítaných hodnot pro všechny referenční body (v těchto případech pak platí, že čím je výsledná fitness bližší nule, tím je lepší), případně jako počet bodů, pro než se aproximovaná hodnota dostatečně přiblížila té požadované bez dalšího rozlišení o jak přesné přiblížení se jednalo [24]. Ilustrace všech tří způsobů je ukázána na obrázku 3.1.

Návrh kombinačních obvodů

Další úlohou, kterou je možno řešit pomocí genetického programování, je návrh kombinačních obvodů. Nejčastěji používanou metodou pro jejich návrh je využití takzvaného kartézského genetického programování, kterému je věnována kapitola 4. Nyní se zaměříme pouze na důvod proč bývá genetické programování použito.

Vytvoření kombinačního obvodu schopného provádět nějakou funkci je člověkem proveditelná úloha. Najít efektivní, respektive nejefektivnější možné zapojení je však velmi složité i pro malé obvody. Jak ilustruje obrázek 3.2, i návrh násobičky pro násobení tří a čtyř bitového čísla je velmi komplexní záležitostí. S každým přidaným vstupem obvodu se u kombinačních obvodů zdvojnásobuje množství možných vstupů, a v závislosti na funkci kterou má obvod vykonávat tak může dojít i k dvojnásobnému nárůstu počtu potřebných hradel.



Obrázek 3.2: Nejlepší známé řešení násobičky tří a čtyř bitového čísla [20].

Kromě sestavení obvodu je možné kombinační obvody pomocí genetického programování také vylepšovat. Například u již zmíněných násobiček se podařilo genetickým programováním snížit počet potřebných hradel v průměru o dvacet procent oproti nejlepšímu člověkem vytvořenému návrhu [20]. Přesto, že se současnými technologiemi lze efektivně navrhnout obvody s nejvýše 15 vstupními hodnotami, geneticky vylepšit lze i obvody které mají vstupů stovky [13].

Ne od všech kombinačních obvodů se vždy vyžadují zcela přesné výsledky. Pokud to zamýšlená aplikace umožňuje, je pak možné pomocí genetického programování hledat obvody, které poskytují dostatečně přesné výsledky za použití nižšího počtu hradel, případně obvody optimalizovat i na řadu dalších parametrů jako mohou být například zpoždění obvodu, nebo jeho elektrická spotřeba, která se projeví nejen na životnosti baterie zařízení ve kterém bude obvod umístěn, ale i na množství vyzářeného tepla [21].

Kapitola 4

Kartézské genetické programování (CGP)

Kartézské genetické programování (CGP) je jednou z podmnožin klasického genetického programování, od kterého se v mnoha ohledech odlišuje. Původně se jednalo o techniku pro automatický návrh číslicových obvodů, ke kterému se dodnes používá, ale později se vyvinulo v plnohodnotný způsob genetického programování [12]. V CGP jsou jedinci reprezentováni pomocí dvojrozměrné mřížky výpočetních uzlů. Na těchto mřížkách se podle chromozomů sestavují orientované acyklické grafy, které představují fenotypy jednotlivých možných řešení problému.

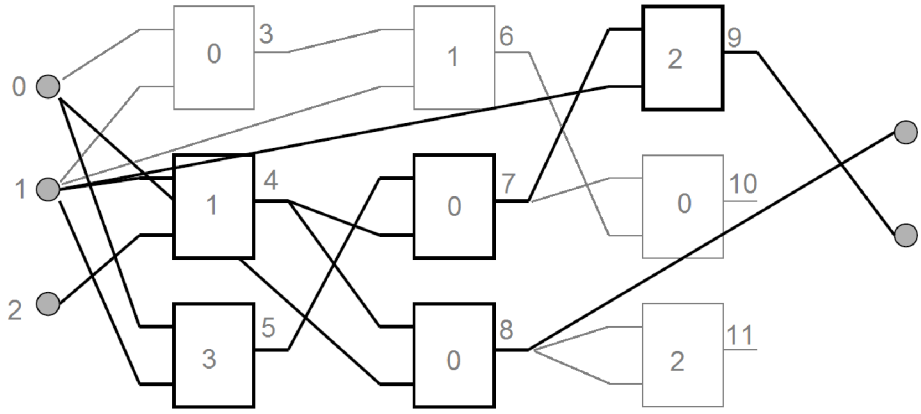
4.1 Reprezentace chromozomu problému

Místo stromů jsou v CGP chromozomy reprezentovány vektorem celých čísel konstantní délky. Jednotlivé geny kódují způsob propojení jednotlivých uzlů, funkce uzlů, a které z těchto uzlů jsou výstupem programu. Mřížka CGP přitom může představovat například pole logických hradel umístěných na fyzickém médiu. Velikost této mřížky, a tedy i počet spojení, je přitom po celou dobu výpočtu konstantní a chromozom si při výpočtu vždy zachovává svoji délku. Toto představuje výhodu oproti klasickému způsobu genetického programování, kde je potřeba pracovat s chromozomy proměnné a teoreticky i shora neomezené délky.

Množina funkcí, které mohou jednotlivé uzly vykonávat, je závislá na aplikační doméně a odpovídá množině funkcí tak, jak byla uvedena v kapitole o obecném genetickém programování. Protože tyto funkce musejí být všechny vykonatelné pomocí stejných výpočetních uzlů, mají všechny uzly vždy stejný počet vstupních parametrů, i když ne všechny parametry musejí být všemi možnými funkcemi využity. V chromozomu je každá funkce uzlu reprezentována unikátním číselným kódem. Když je při vyhodnocování jedince program spuštěn, jsou kódy převedeny na odpovídající funkce, podle kterých jsou pak vstupy uzlů zpracovány [14].

Počet uzlů, které se na mřížce nacházejí, odpovídá součinu její výšky (n_r) a šířky (n_c). Způsob, jakým jsou vzájemně propojeny, vychází z požadavku na acykličnost vytvořeného grafu. Uzly nacházející se v prvním sloupci tabulky mohou jako svoje vstupy použít pouze vstupy celého programu. Uzly nacházející se v dalších sloupcích mohou použít buď vstupy programu nebo výstupy uzlů z předcházejících sloupců. To, o kolik sloupců zpět se mohou uzly připojit, je určeno parametrem *levels-back*, který může nabývat hodnoty v rozsahu

$[1, n_c]$. Přičemž vstupy samotného programu jsou jednotlivým uzlům dostupné vždy bez ohledu na tento parametr. Toto omezení existuje proto, že pokud je výsledek programování určen pro hardwarovou implementaci, není mnohdy možné spojení mezi vzájemně vzdálenými sloupci fyzicky uskutečnit. Pokud je program určen pro softwarovou implementaci, bývá parametr levels-back nastaven na n_c, a parametr n_r nastaven na jedna. To v praxi znamená, že výpočty jednotlivých uzlů jsou prováděny jednotlivě a každý nový výpočet může využívat výsledků všech předcházejících výpočtů [13].



Obrázek 4.1: Ukázka mřížky CGP, součástí fenotypu jsou pouze černé uzly. Zapojení uvedené na obrázku by bylo zakódováno číselným vektorem [(0,1,0),(1,2,1),(0,1,3),(3,1,1),(5,4,0),(4,0,0),(7,1,2),(7,6,0),(8,8,2),8,9] [22].

Aby bylo možné propojení uzlů zakódovat do chromozomu, je všem vstupům programu a všem uzlům přiřazen unikátní index. Výsledný chromozom je složen ze dvou částí. V první jsou v podobě n-tic čísel zakódovány jednotlivé uzly, seřazené podle sloupců a řádku. Každá n-tice pak obsahuje indexy vstupů daného uzlu následovány indexem funkce, kterou má daný uzel vykonávat. V druhé části chromozomu jsou uvedeny indexy uzlů, které poskytují výstupy programu. Příklad chromozomu CGP je uveden na obrázku 4.1.

Důležitou vlastností CGP je, že i když délka chromozomu zůstává vždy stejná, fenotyp tvoří pouze ty uzly, které ovlivňují výstup vytvořeného programu. Velikost fenotypu je tedy proměnlivá a velikost chromozomu ji omezuje pouze shora. Ostatní uzly, které se v genotypu nacházejí, však také mají svoje využití. I když se nijak nepodílejí na celkovém výsledku, mohou stále nést užitečné geny a stát se součástí fenotypu ve chvíli, kdy se změní zapojení obvodu a jejich výstup je využit.

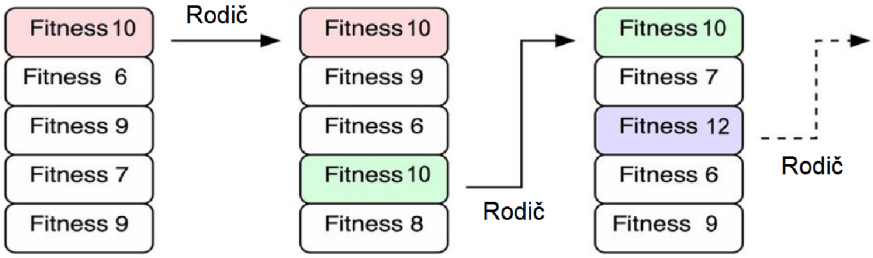
4.2 Genetické operátory

Jediný genetický operátor, který se u CGP obvykle používá, je mutace. Ta probíhá nahrazením jednoho nebo více genů náhodným jiným genem s platnou hodnotou. Může tedy změnit funkci, kterou uzel provádí, nebo může změnit, kam je připojen jeden z jeho vstupů. Všechny provedené změny však vždy musejí respektovat požadavek na acykličnost grafu a parametr levels-back.

Důvodem, proč se u CGP nepoužívá křížení, je, že pro většinu praktických aplikací dosud nebyl nalezen způsob, jak při křížení zachovat logiku obou rodičů. Pokud bychom totiž chtěli provést něco podobného jako při křížení genetických stromů, narážíme na problém, že jeden uzel může být u obou rodičů používán ke zcela jinému účelu. Provedení jakékoliv změny jeho funkce u případného potomka by mohlo funkci druhého rodiče zcela zničit. Křížení tedy nevede k zachování funkcionality a propagaci dobrých genů do dalších generací. A tak, i když možnosti použití křížení v některých aplikacích CGP existují, nejde o rozšířenou záležitost [2].

4.3 Evoluční cyklus CGP

CGP se kromě specifického způsobu reprezentace chromozomů a genetických operátorů vyznačuje také dalšími parametry evoluce. Velikost použité populace je většinou velmi malá, v rámci jednotek. Naopak počet generací je velmi vysoký. První generace je vytvořena zcela náhodně. V dalších generacích je pak populace tvořena vždy jen jediným rodičem a jeho potomky. Rodičem následující generace se vždy stává jedinec s největší fitness. Pokud je nejvyšší hodnota fitness sdílena více jedinci, je vždy vybrán ten, který nebyl rodičem současné generace, jak ukazuje obrázek 4.2.

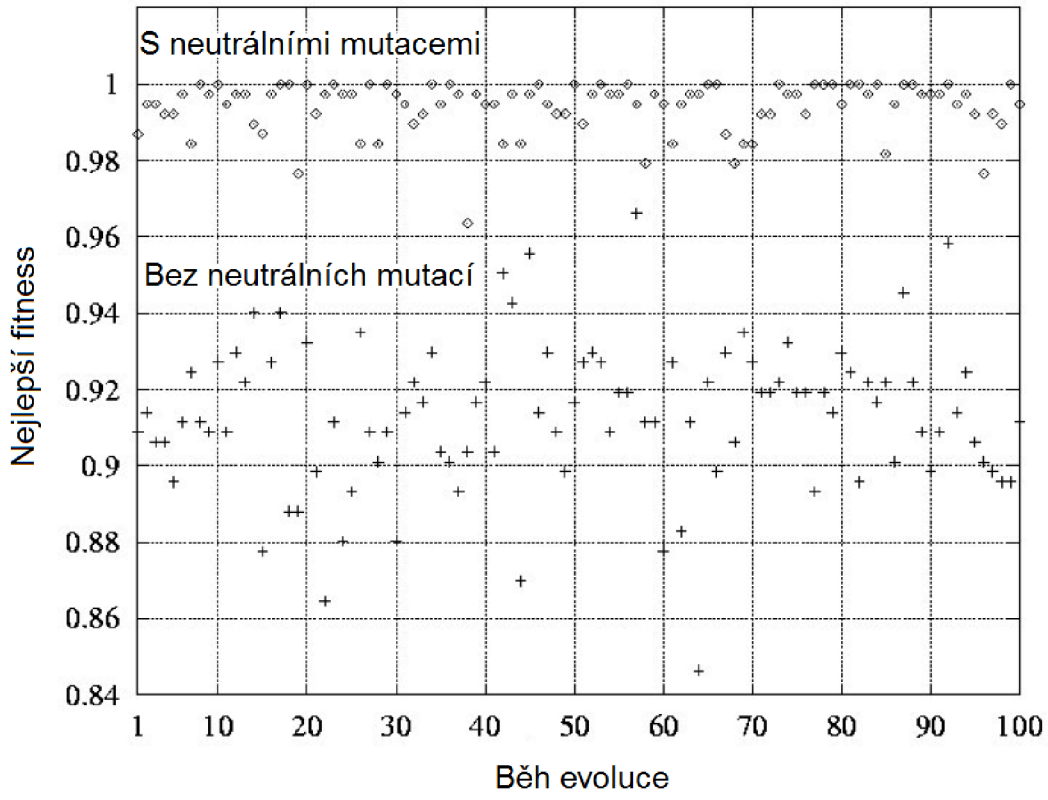


Obrázek 4.2: Výběr rodiče v CGP, při shodě fitness jsou preferováni potomci [13].

Jedinci, kteří ještě nebyli rodičem, jsou preferováni, proto, že i když fitness potomka je stejná, mohlo u něj dojít k vytvoření úspěšného genu v rámci takzvané *neutrální mutace*. Tedy mutace, která se uskutečnila mimo uzly, které jsou součástí fenotypu daného chromozomu. Kdyby byl vždy vybírán původní rodič, tak by se následující řešení opakovaně nacházela ve stejném podprostoru možných řešení [12]. Obrázek ukazuje statistiku vlivu náhodných mutací na kvalitu nalezeného řešení.

Z vybraného jedince je náhodnými mutacemi vytvořeno tolik potomků aby došlo k obnovení populace. Parametr počtu potomků při tom bývá označován symbolem λ. Vybraný rodič a jeho potomci pak vytvoří následující generaci pro kterou ni je opět vyhodnocena fitness a celý cyklus se opakuje. Ukončovací podmínkou může být buď nalezení dostatečně dobrého řešení, vyčerpání počtu generací nebo času který je pro běh evoluce přidělen.

Výstupem CGP je program sestavený na základě fenotypu jedince s nejlepší fitness v poslední generaci. Protože CGP jako rodiče dalších generací vždy vybírá jedince s nejlepší fitness, jde o výběr elity popsaný v 2.3 a je zaručeno, že v průběhu evoluce hodnota fitness nikdy nebude snížena, je zaručeno že nejlepší kdy objevený jedinec bude součástí koncové populace, a není tedy potřeba si průběžná nejlepší řešení ukládat.



Obrázek 4.3: Vliv neutrálních mutací na kvalitu řešení nalezeného pomocí CGP [13].

Kapitola 5

Genetické vylepšování software

Genetické vylepšování představuje využití genetického programování, popřípadě i jiných prohledávacích algoritmů pro vylepšení vlastností již existujících člověkem vytvořených programů.

Hlavním cílem práce programátora je obvykle vytvořit správně fungující kód, jehož optimalita (dle různých kritérií) hraje v mnoha aplikacích druhou roli. Pokud se poté ukáže, že na optimalitě daného kódu opravdu záleží, vyvstává problém. Optimalizace rozsáhlých programů je totiž velmi náročnou záležitostí. Samotná analýza i krátkého kódu může člověku zabrat dny pokud není dobře zdokumentován. A pokud byl program tvořen týmem programátorů, jak tomu u velkých projektů bývá, nemusí žádný z nich mít detailní přehled o funkčnosti všech součástí programu a způsobech, jakými spolu tyto součásti interagují, a perfektní optimalizace tedy může být nejen těžká, ale prakticky neproveditelná.

Dále uvažme situaci, kdy byl kód optimalizován běžně používanými technikami. Jedním ze způsobů, jak provést dodatečnou optimalizaci, je pro optimalizaci programu použití evoluční algoritmy. Vlastností, která je takto vylepšována, bývá nejčastěji doba běhu, ale mohou být vylepšeny i jiné vlastnosti, například paměťová náročnost, energetická spotřeba, důležitá především u mobilních zařízení, či bezpečnost. A to beze změny funkcionality programu ze kterého se vychází [8]. Parametry, které nemají přímý vliv na výstupy programu, jsou označovány pojmem *nefunkcionální*.

Podobně jako u klasického genetického programování je i u vylepšování naším cílem vytvořit program, který pro dané vstupy poskytuje co nejlepší výstupy. Díky tomu, že při genetickém vylepšování vycházíme z již existujícího programu, nepotřebujeme mít při vylepšování k dispozici formální definici požadovaných výstupů. Mnohdy stačí, že získané výstupy mohou být porovnány s výstupy originálního programu. Ten je tak použit jako takzvané *orákulum* rozhodující, jaké výstupy programu jsou považovány za správné. Pokud formální definici máme a originální program obsahuje nějaké chyby, případně pokud program poskytuje výsledky pouze s nějakou přesností, může být vylepšena i jeho funkcionalita.

5.1 Výhody vylepšování oproti programování

Protože genetické vylepšování hledá změny které se mají v originálním programu provést, a nesnaží se vytvořit zcela nový program, stačí u něj, aby v rámci evolučního cyklu pracovalo pouze se seznamy či stromy změn, místo s reprezentací celého programu. Tato vlastnost nejen šetří paměť, ale výrazně usnadňuje a urychluje také práci s chromozomy a celý ge-

netický cyklus, protože chromozomy neobsahují ty části programu, které dosud nebyly, případně ani nemohou být, změněny.

V závislosti na jazyce, v jakém je vylepšovaný program implementován, může být na rozdíl od klasického genetického programování prohledávání stavového prostoru omezeno pouze na určité úseky kódu. Díky tomuto zaměření se tak algoritmus může vyhnout zbytečným pokusům o přeprogramování těch částí programu, jejichž optimalizace je téměř bezvýznamná a zaměřit se pouze na důležité, vysoce vytížené části programu. Stejně tak je možné se vyhnut i marným pokusům o vylepšování dobře známých a již zcela optimálních algoritmů, jako mohou být například některé řídicí algoritmy.

Naopak, pokud máme správně fungující a již optimalizované funkce, či jiné části programu, může být genetické vylepšování použito i pro vložení těchto funkcí do jiných existujících programů za účelem jejich zefektivnění či zlepšení [27].

5.2 Příklady použití

Přestože genetické vylepšování je poměrně novou technikou, ukázalo se jako velmi zajímavá a perspektivní oblast. Mezi nejzajímavější projekty patří tyto:

- Použitím genetického vylepšení byly odstraněny chyby v desítce stávajících programů vytvořených v jazyce C. Zapotřebí při tom byl pouze jeden konkrétní vstup, který chybu programu demonstroval, a několik vstupů poskytujících správné výsledky, jejichž funkčnost zůstala nedotčena [26].

- Použitím genetického vylepšení na program *Bowtie2* používaného například pro zarovnávání sekvencí DNA, se podařilo tento 50000 řádkový program 70 krát zrychlit a zároveň mírně zpřesnit jeho výsledky [10].

- U programů pro zpracování obrazu, vědecké výpočty a finanční analýzu určených pro přenosná zařízení se v určitých případech podařilo snížit energetické nároky o téměř 20 %, pouze s minimálním vlivem na jejich spolehlivost [27].

- Použitím na program pro skládání RNA se podařilo dosáhnout paralelizace některých částí programu, a jeho běh tím v extrémních případech až deseti-tisícinásobně urychlit [18].

- Z binárních souborů již kompilovaných programů se genetickým vylepšování podařilo odstranit nepoužívané, nebo nechtěné funkce, bez omezení ostatní funkčnosti. Takto byla zmenšena velikost a zlepšena bezpečnost programu [7].

- Genetické vylepšení bylo použito pro automatický převod originální implementace unixové utility *Gzip* na nové moderní počítačové architektury. Podařilo se využít hardwarové podpory, která v době původní implementace ani neexistovala. To vše při zachování veškeré původní funkčnosti [9].

Kapitola 6

Návrh systému pro genetické vylepšování CGP

Tato kapitola se zabývá návrhem ideálního programu, který by byl použitelný pro vylepšení implementace CGP. Navržený ideální systém by však vyžadoval provádění automatizované syntaktické a sémantické analýzy zdrojového kódu na velmi vysoké úrovni přesahující rozsah diplomové práce. Proto je navržen i druhý jednoduší systém, který bude v rámci tohoto diplomového projektu skutečně implementován.

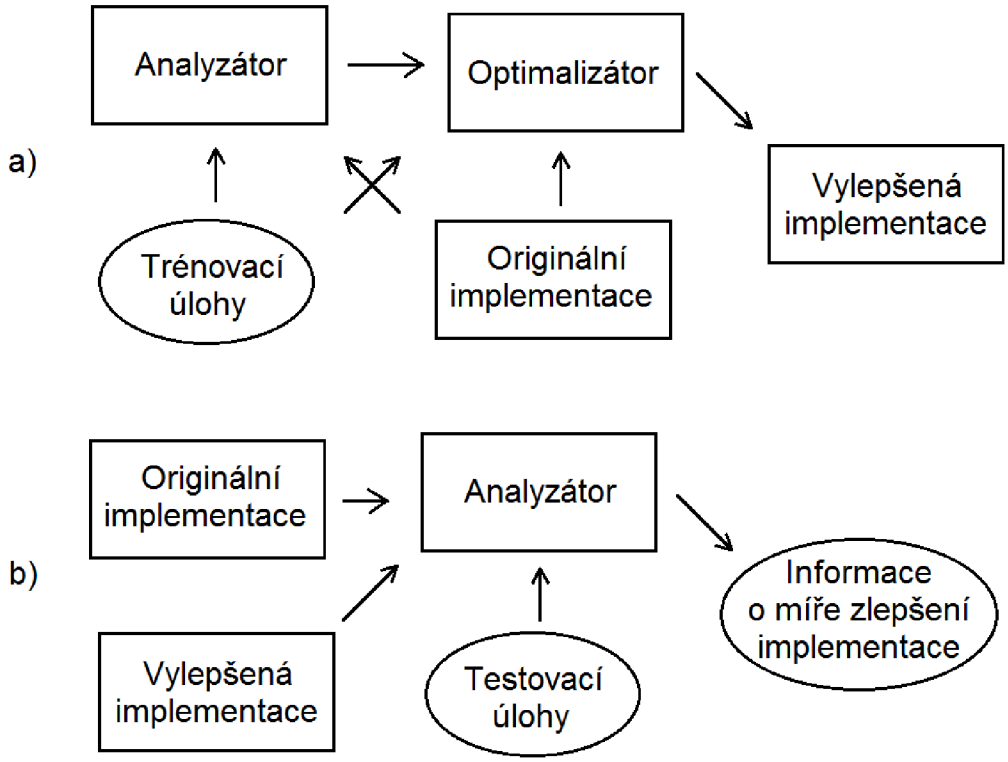
6.1 Vize genetického vylepšování implementace CGP

Ideální systém by sestával ze tří samostatných částí. První částí je originální implementace CGP, vytvořená člověkem, který se snažil implementovat CGP, jak nejlépe dovedl a o možnosti, že by jeho program mohl být dále nějak geneticky vylepšen, ani neuvažoval. Druhou částí je analyzátor kódu, který provede sofistikovanou analýzu zdrojového kódu a poskytne informace o tom, které jeho části skrývají největší potenciál pro zlepšení. Třetí částí je pak optimalizátor, který originální implementaci CGP na základě analyzátorem poskytnutých informací geneticky vylepší. Schéma funkce ideálního systému je ukázáno na obrázku 6.1.

6.1.1 Software pro kartézské genetické programování

První program bude představovat člověkem vytvořenou implementaci programu pro kartézské genetické programování. Řešeným problémem bude symbolická regrese. Zdrojový kód tohoto programu bude poté sloužit jako vstup pro další části projektu, které se tuto originální implementaci budou snažit genetickým programováním vylepšit.

Program bude implementován v jazyce C++ a pro jeho snadnější zpracování dalšími částmi projektu budou jeho řádky rozděleny do dvou skupin. První skupinou budou nedotknutelné řádky, které nebude evolučnímu algoritmu dovoleno nijak upravovat. Mezi tyto řádky budou patřit ty, které souvisejí se zpracováním vstupních parametrů, poskytováním textových výpisů a vyhodnocením úspěšnosti programu. Druhou skupinou budou řádky dotknutelné. Do této kategorie budou patřit ty řádky, které mají vliv na samotné kartézské genetické programování a způsob, jakým jej program provádí. Například konstanty určující velikost populace, počet iterací jednotlivých kroků, rozhodování, které funkce budou volány, a rozhodování v jakém pořadí budou jednotlivé kroky evoluce vykonávány.



Obrázek 6.1: Navrhované schéma systému pro genetické vylepšení CGP. a) Vytvoření vylepšené implementace. Analyzátor analyzuje funkci originální implementace při běhu na trénovacích úlohách. Optimalizátor využije výsledek analýzy a trénovací úlohy a vylepší implementaci CGP. Výstupem je nejlepší implementace, kterou optimalizátor v rámci evoluce vytvořil. b) Otestování vylepšené implementace. Analyzátor analyzuje funkci originální a vylepšené implementace CGP na testovacích úlohách a porovná jejich výsledky.

Protože evoluce je časově náročná a cílem celého projektu je vytvořit program pro evoluci programu, který sám o sobě provádí další evoluci, bude funkčnost programu testována na sadě relativně jednoduchých funkcí, u kterých bude aproximován pouze malý počet testovacích bodů.

Operátory, které budou kartézským genetickým programováním použity, budou: sčítání, odčítání, násobení, dělení, sinus a cosinus.

Fitness jedinců bude hodnocena podle počtu dostatečně přesně vypočtených hodnot. Na optimálnost implementace z pohledu ostatních parametrů, jako je rychlost či počet použitých uzlů, nebude brán zřetel. Ukončovací podmínkou algoritmu bude nalezení fenotypu který vypočte 100% hledaných bodů s povolenou chybou, případně uplynutí přiděleného času.

6.1.2 Analyzátor kódu

Druhým programem, který bude implementován, bude program pro analýzu textových výpisů prvního programu. Analyzátor bude první program opakovaně volat a zpracovávat údaje o jeho úspěšnosti a míře využití jednotlivých řádků kódu. Analyzátor poté provede statistické zpracování zjištěných dat a poskytne informace o průměrném vytížení jednotlivých částí prvního programu. Tyto informace zjišťujeme proto, že čím vícekrát je řádek použit, tím větší pozitivní vliv může jeho změna mít na rychlost běhu programu. Čím vícekrát byl řádek použit, tím větší mu bude při genetickém vylepšování věnována pozornost.

6.1.3 Optimalizátor

Třetím programem vytvořeným v rámci projektu bude optimalizátor založený na evolučních algoritmech. Jeho vstupem bude zdrojový kód prvního vytvořeného programu (implementace CGP). Cílem bude tento zdrojový kód optimalizovat. Vstup bude načten a za využití překladače *Clang* [11] převeden do podoby syntaktického stromu, který bude použit jako základ pro vytvoření počáteční populace optimalizátoru. Parametry běhu, udávající na které řádky se má optimalizace zaměřit, budou nastaveny na základě údajů zjištěných analyzátorem.

Způsobů, jakými se pokusíme program vylepšit, bude několik. Prvním bude vylepšení funkčnosti programu, druhým bude jeho zrychlení při zachování stejné funkčnosti a třetím bude zrychlení programu za cenu přijatelného zhoršení jeho funkčnosti.

Jednotlivé části evolučního algoritmu jsou navrženy následovně:

- **Reprezentace jedinců** – Místo toho, aby každý jedinec obsahoval celý upravený zdrojový kód originálního programu, budou chromozomy reprezentovány jako seznamy změn, které mají být v originálním kódu provedeny. Originální zdrojový kód tedy bude reprezentován prázdným seznamem. Toto umožní nejen snížit nároky na paměť a urychlit běh programu, ale také usnadní mutaci a křížení jedinců.

- **Populace** – Populace bude tvořena určitým počtem jedinců, který bude po celou dobu běhu programu konstantní. Počáteční populace bude tvořena jednotlivci vytvořenými z originálního kódu pomocí jedné náhodné mutace.

- **Množina terminálů** – Množina terminálů bude tvořena vybranými konstantami a proměnnými originálního programu a několika dodatečnými konstantami, které bude optimalizátor moci do programu přidat, a funkcemi originálního programu, které nevyžadují další argumenty. Konstanty a proměnné, které součástí množiny terminálů *nebudou*, budou ty, které program využívá pro hodnocení svojí vlastní funkčnosti. Nedotknutelnými funkcemi pak budu funkce použité pro řešení symbolické regrese.

- **Množina funkcí** – Množina funkcí bude tvořena funkcemi originálního programu, které vyžadují argumenty (například funkce fitness, funkce pro provedení mutace jedince), řídicími konstrukcemi jazyka C (*if*, *for*, atd.) a aritmetickými operacemi. Aby se program nemohl dostat do nekonečné smyčky, nebude do množiny zařazena konstrukce *while*, která bude nahrazena konstrukcí *for* nastavenou na velmi vysoký počet iterací, doplněnou o podmíněný *break*.

- **Fitness funkce** – Protože vyvíjený program sám o sobě provádí vývoj, který je do velké míry ovlivněn náhodou, není určení fitness jednoduchou záležitostí. Protože

úspěšnost vytvořeného programu může být při každém běhu odlišná, bude určena statisticky. Každý jedinec bude opakovaně spuštěn na každé z funkcí, o jejichž symbolickou regresi se bude pokoušet, a fitness bude určena jako počet běhů, které jsou schopny v určeném časovém limitu najít dostatečně kvalitní řešení. Tato fitness poté, v závislosti na cíli genetického vylepšování bude doplněna o bonus či penalizaci určenou na základě celkového času, který výpočet fitness zabral.

- **Výběr rodičů** – Pro výběr jedinců, ze kterých bude vytvořena následující generace, bude použita metoda výběru turnajem.

- **Mutace** – Mutace budou prováděny na základě syntaktického stromu originálního chromozomu, provedením jedné náhodné syntakticky platné změny. Výběr, kde se má mutace provést, bude proveden na základě údajů o využití jednotlivých částí kódu. Výběr, jaká mutace má být provedena, bude proveden náhodně z množiny všech syntakticky planých mutací. Protože syntakticky správné mutace mohou vést k sémanticky neplatným konstrukcím, bude po jejím provedení optimalizátor testovat, zda je možné nově vniklý zdrojový kód přeložit. Pokud při překladu dojde k chybě, bude mutace označena jako neúspěšná.

- **Křížení** – Protože jedinci jsou reprezentováni jako seznam změn, které mají být v originálním kódu provedeny, a tyto změny na sobě budou často zcela nezávislé, bude křížení prováděno konkatenací seznamů změn obou rodičů. Protože ale změny na sobě nebudou nezávislé vždy, bude se po provedení konkatenace provádět kontrola, zda je takto vytvořeného potomka možno přeložit. Pokud to možné nebude, bude křížení označeno za neúspěšné, protože pokud upravený zdrojový kód nelze přeložit, není možné ho ani otestovat a určit jeho fitness. Aby se předešlo problému s nekontrolovaným nárůstem velikosti jedince (zvanému *bloat*), bude maximální délka seznamu jedince omezena horním limitem. Pokud by zkřížením jedinců došlo k překročení tohoto limitu, bude křížení označeno jako neúspěšné.

- **Tvorba nové populace** – Nová populace bude tvořena následujícím způsobem. Nejprve budou vybrány páry jedinců ze současné populace, které spolu zkusíme vzájemně zkřížit. Pokud bude křížení jedinců neúspěšné, bude vybrána nová dvojce o jejíž zkřížení se budeme pokoušet. Toto se bude opakovat, dokud potomky nezaplníme dostatečnou část nové populace, nebo dokud nedosáhneme limitu na maximální počet pokusů o křížení. V extrémních případech se může stát, že žádná vybraná dvojice rodičů nebude schopna vytvořit přijatelného potomka. Po skončení křížení bude zbytek populace doplněn jedinci vzniklými mutací jednotlivců vybraných ze současné populace. Protože i mutace mohou být neúspěšné, bude počet pokusů o mutaci také omezen. Pokud se novou populaci nepodaří doplnit mutacemi současné populace, bude nová populace doplněna náhodně vygenerovanými jedinci, podobně jako tomu bylo při tvorbě úplně první generace.

Na konci evoluce bude výstupem programu chromozom vylepšující originální zdrojový kód programu pro symbolickou regresi pomocí kartézského genetického programování. Tento nově vzniklý optimalizovaný kód bude přeložen a spolu s původním neoptimalizovaným programem vyzkoušen na sadě původních i řadě nových úloh. Výsledky budou statisticky vyhodnoceny a porovnány z hlediska průměrné rychlosti a úspěšnosti, aby mohlo být prokázáno, že původní kód byl optimalizátorem opravdu vylepšen.

6.2 Zjednodušený návrh pro implementaci

Zjednodušený návrh projektu sestává ze dvou částí, originální implementace CGP a optimalizátoru pro její vylepšení. Analyzátor byl z návrhu odstraněn, protože aby fungoval požadovaným způsobem, musel by být schopen velmi pokročilé syntaktické a sémantické analýzy, která přesahuje rámec i zaměření tohoto diplomového projektu.

6.2.1 Software pro kartézské genetické programování

Zdrojový kód originální, člověkem vytvořené, implementace CGP bude sloužit jako základ, který bude poté vylepšen optimalizátorem. Implementace proběhne v jazyce C++ a od teoretického návrhu popsaného v předchozí sekci se tato implementace bude lišit způsobem, jakým budou označena místa vhodná pro aproximaci. Místo rozdělení řádků na dotknutelné a nedotknutelné tak, aby mohly dále být zpracovány analyzátorem, který byl z návrhu po implementaci odstraněn, budou ve zdrojovém kódu zaneseny značky určené přímo pro samotný optimalizátor.

Programátor tak vezme zdrojový kód implementace a na místa, která uváží za důležitá, nebo jinak zajímavá, zanese značky, jejichž obsah poté bude optimalizátorem měněn. Toto platí nejen pro implementaci CGP vytvořenou v rámci tohoto projektu, ale i pro případné další implementace CGP a případně i další obdobné optimalizační programy převzaté z jiných zdrojů a od jiných autorů.

Protože se snažíme o návrh implementace, která by mohla být snadno nahrazena i jinou, převzatou implementací, budeme fitness definovat jednodušeji než v teoretickém návrhu. Vlastnosti, jako počet použitých uzlů, mají význam pouze při řešení některých úloh jako je například návrh kombinačních obvodů, nikoliv však u symbolické regrese. Aby byl optimalizátor použitelný i pro další implementace CGP řešící jiné úlohy, než které budou v rámci projektu zkoušeny, budeme jako výstup měřit pouze dvě veličiny.

První veličinou je výpočetní čas, který jsme zvolili, protože jde o veličinu, kterou vykazuje každá implementace CGP (a obecně i jakýkoliv jiný program). Víme, že nižší výpočetní čas je z praktického hlediska vždy lepší, a to bez ohledu na to, jaký problém konkrétní implementace CGP řeší, proč jej řeší, nebo kdo ji vytvořil. Výpočetní čas je samozřejmě závislý na výkonosti stroje, na kterém je program spuštěn. Pro potřeby této práce ale můžeme předpokládat, že poměrné zrychlení zůstane stejné i napříč platformami.

Druhou veličinou, která bude měřena, je kvalita řešení. Ta je v CGP vždy určena hodnotu funkce fitness, případě nějakým jejím ekvivalentem. Konkrétní definice a způsob jejího výpočtu se však může lišit napříč implementacemi, a být závislý na řadě faktorů, jako je počet použitých výpočetních uzlů, očekávaný příkon nebo zpoždění navrhovaného zařízení, a samozřejmě i velikost a počet vstupů konkrétní řešené úlohy.

Kvalitu řešení proto definujeme následovně. Hodnota nula představuje nejlepší možné řešení, jenž nemusí být nutně dosažitelné. Dále, větší hodnota představuje horší řešení. Záporné hodnoty kvality řešení pak nejsou povoleny.

Tato obecná definice kvality řešení může být použita v mnoha různých implementacích. V implementaci vytvořené v rámci projektu bude kvalitu řešení definovat suma čtverců odchylek od požadovaných hodnot, kterých se implementace CGP dopustí při symbolické regresi. Stejně tak dobře by ji ale bylo možné definovat jako množství bodů symbolické regrese, jejichž aproximovaná hodnota se od té požadované liší o více jak deset procent.

Výpočetní čas i kvalita řešení mají společné, že jejich ideální hodnotu je nula, jejich horní hodnota není teoreticky omezena, a v obou případech se snažíme o jejich minimalizaci. Toto nám v rámci optimalizace umožní s oběma parametry pracovat podobným způsobem. Fitness geneticky vylepšené implementace CGP pak bude spočtena kombinací obou těchto parametrů.

6.2.2 Optimalizátor

Optimalizátor bude implementován v jazyce C++. Protože součástí návrhu pro implementaci není automatizovaný analyzátor, optimalizátor nebude mít k dispozici syntaktický strom zdrojového kódu. Bude provádět úlohu, ke které jej nepotřebuje. Místo úpravy syntaxe zdrojového kódu se zaměří se na upravování numerických hodnot implementace CGP. To, které hodnoty konkrétně budou měněny, bude záležet na umístění značek ve zdrojovém kódu vylepšované implementace CGP. Všechny označené hodnoty budou vylepšovány, všechny neoznačené hodnoty budou ponechány v původním stavu.

Důvodem, proč se optimalizátor nebude snažit o detekci a vylepšení všech numerických hodnot ve zdrojovém kódu implementace CGP je, že se v ní může vyskytovat řada hodnot, jejichž změna nedává sémanticky smysl, nebo má pouze minimální vliv na funkčnost. Příkladem takových to hodnot může být například počet očekávaných vstupních parametrů programu, nebo velikost jen jednou použitého bufferu pro načtení vstupní úlohy.

Jedinci populace modifikovaných verzí CGP budou místo seznamu změn, které mají být v zdrojovém kódu vylepšovaného programu provedeny, reprezentovány pomocí vektoru hodnot. Počáteční populace bude vytvořena extrakcí originálních označených hodnot z vylepšované implementace. V průběhu evoluce pak budou tyto vektory kříženy a mutovány jako normální numerické vektory, jak bylo popsáno v sekci 2.2. Vylepšené implementace CGP budou z jedinců vytvořeny nahrazením originálních označených hodnot hodnotami z evolučně vylepšených vektorů.

Hodnota fitness jednotlivých programů bude zjištěna následujícím způsobem. Každý vytvořený jedinec bude přeložen a spuštěn na zvolené trénovací úloze. Optimalizátor změří výpočetní odchylku a dobu výpočtu, a z těchto hodnot vypočte unifikovanou fitness, která určí celkovou kvalitu daného jedince.

Protože značky budou do zdrojového kódu umístěny člověkem a není měněna syntaxe zdrojového kódu, nepředpokládá se, že budou v rámci evoluce vytvořeni jedinci, které by nešlo přeložit (křížení numerických vektorů nemůže být neúspěšné). Tvorba nové populace tedy bude probíhat výrazně jednodušším způsobem než u teoretického návrhu. Nejprve budou pro každého nového jedince vybráni dva rodiče na základě jejich fitness (metodou turnaje). Jejich zkřížením pak vnikne potomek, který bude ještě zmutován a umístěn do nové populace, která bude celá tvořena takto vytvořenými novými jedinci.

Optimalizátor poté vytvoří kopie originálního kódu implementace CGP, ve kterých nahradí původní označené hodnoty hodnotami nových jedinců, zavolá standardní překladač zdrojového kódu odpovídajícího programovacího jazyka (G++) a přeložené programy spustí na vybraných testovacích úlohách.

Protože v rámci evoluce může celková fitness vytvořených jedinců kolísat, bude si optimalizátor pamatovat fitness a vektor hodnot nejlepšího jedince, který byl v rámci evoluce kdy vytvořen. Výstupem optimalizátoru pak bude vylepšená implementace CGP vytvořená na základě zapamatovaného vektoru hodnot tohoto jedince.

Kapitola 7

Implementace

Tato kapitola popisuje, jak byl návrh pro implementaci popsaný v předchozí kapitole skutečně implementován. Schéma celého projektu je znázorněno na obrázku 7.1.

Celkem byly implementovány dva programy v jazyce *C++* a to originální implementace CGP řešící problém symbolické regrese a optimalizátor, který tuto implementaci geneticky vylepší. Dále byly implementovány dva krátké skripty v jazyce *Python*, které se starají o extrakci a nahrazování označených hodnot ze zdrojového kódu vylepšované implementace CGP.

7.1 Originální implementace CGP

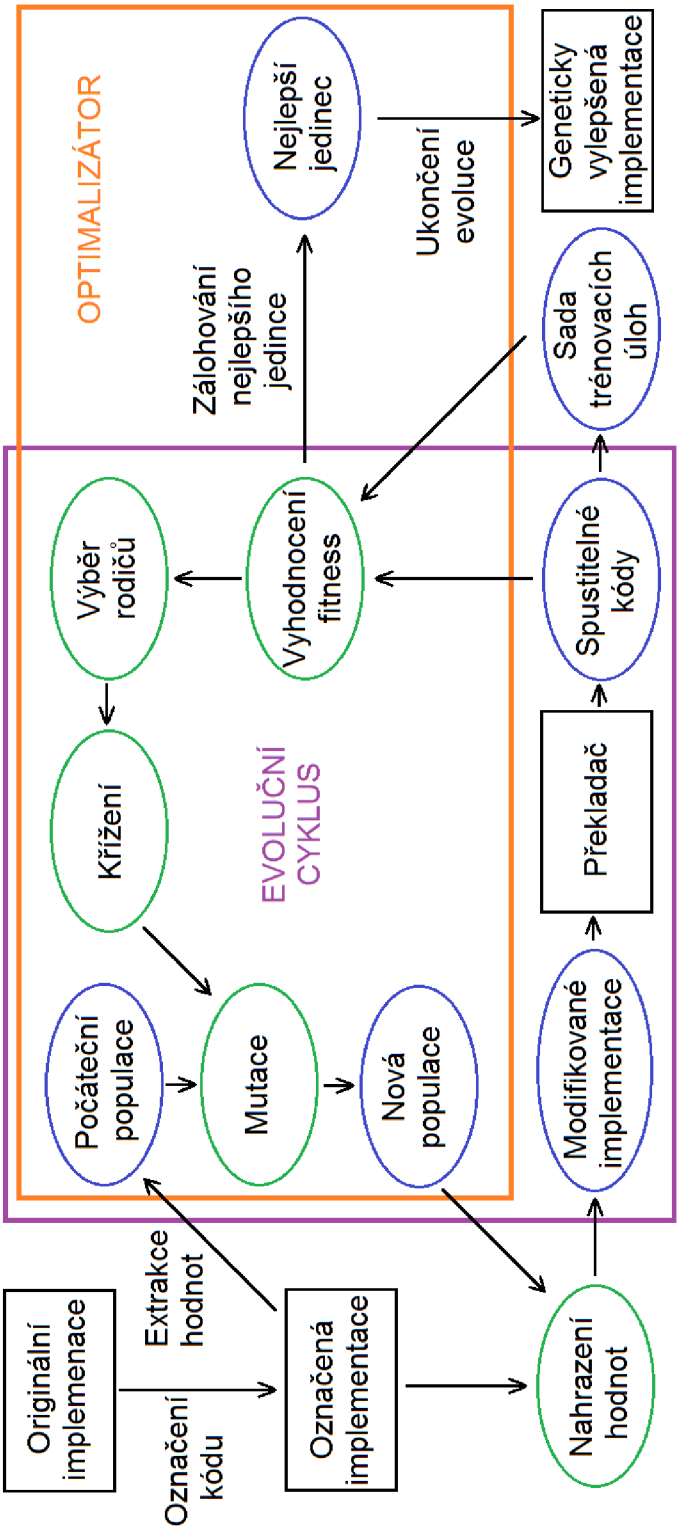
Originální implementace CGP řešící problém symbolické regrese je umístěna v souboru *original.cpp*. Program má pouze jeden vstupní parametr, kterým je jméno souboru se vstupní posloupností. Tato posloupnost musí být ve formátu `X,Y`, kde hodnota `X` představuje hodnotu závislé proměnné a hodnota `Y` představuje požadovanou hodnotu výstupu. Datovým typem obou hodnot je *double*. Každá dvojice se musí nacházet na vlastním řádku. Implementace tohoto programu vychází z již dříve vytvořeného projektu vytvořeném předmětu *Biologií inspirované počítače*, která byla inspirována implementací CGP pro návrh kombinačních obvodů, jejímž autorem byl Zdeněk Vašíček. Jedinou částí kódu, která od tohoto autora ve zdrojovém kódu zůstala, je funkce předpočítání platných mutací.

Implementace pracuje takovým způsobem, že nejdříve načte vstupní soubor a převede jej do interní reprezentace. Poté jsou v závislosti na parametrech CGP (umístěných začátku zdrojového kódu v podobě označených konstant) předpočítány platné hodnoty mutací pro jednotlivé pozice v chromozomu CGP. Poté je (při základním nastavení) provedeno patnáct nezávislých běhů CGP a to standardním způsobem, který byl popsán v teoretické části (kapitola 4) a v návrhu implementace (sekce 6.2).

V průběhu evoluce jsou ukládány výsledné fitness jednotlivých běhů, a jako výsledek je navrácen medián těchto hodnot. Samotná funkce fitness $F(x)$ je implementována jako suma čtverců odchylek vypočtených na základě funkce $P(i)$ evaluované CGP a očekávaných výstupních hodnot $Y(i)$.

$$F(x) = \sum_{i=1}^{n} (P(i) - Y(i))^2$$

Kde x představuje vstupní posloupnost a n počet jejích hodnot.



Obrázek 7.1: Schéma funkce vytvořeného systému pro vylepšení implementace CGP.Jednotlivé programy jsou vyznačeny černou barvou. Množiny dat jsou vyznačeny modře. Prováděné činnosti jsou vyznačeny zeleně. Vše nacházející se v oranžovém rámci je součástí činnosti optimalizátoru. Evoluční cyklus je vyznačen fialově.

Kromě měření fitness optimalizátor provádí i orientační měření výpočetního času, který každý evoluční běh i vykonání celého programu zabralo. Za zmínku stojí ještě konstanta `VYPISY`, která určuje množství výpisů, které bude implementace při svém běhu provádět. Při ručním použití tohoto programu se doporučuje nastavit tuto hodnotu na vysokou hodnotu, aby bylo možné sledovat jednotlivé běhy programu. Při automatizovaném použití optimalizátorem se doporučuje pro zachování přehlednosti nastavit tuto hodnotu na nulu a potlačit tím všechny textové výpisy této implementace.

Implementace CGP obsahuje pro potřeby symbolické regrese šest matematických funkcí, kterými jsou: sčítání, odčítání, násobení, dělení, sinus a cosinus, přičemž výsledek dělení nulou je považován za hodnotu nula. Dále pak obsahuje čtyři označené konstanty, které spolu s odpovídající hodnotou ze vstupní posloupnosti slouží jako primární vstupy výpočetní mřížky CGP. Počáteční nastavení těchto konstant je: 0.5, 1.0, 2.0 a 10.0, předpokládá se však, že v rámci následného genetického vylepšování budou jejich hodnoty změněny.

Překlad této implementace CGP v rámci tohoto projektu probíhá pomocí překladače *G++* s nastaveným parametrem *-std=c++11*. K překladu lze však použít i jiné překladače, případně dalších jiných doplňujících nastavení.

7.2 Optimalizátor

Program optimalizátor tvoří hlavní část tohoto projektu a jeho zdrojový kód se nachází v souboru *evolver.cpp*. Program provádí genetické vylepšování implementací CGP. Program očekává nejméně čtyři vstupní parametry. Prvním je jméno souboru, ve kterém se nachází zdrojový kód vylepšované implementace. Druhý a třetí parametr jsou použity při výpočtu hodnoty fitness a určují míru zaměření optimalizátoru na vylepšení parametru výpočetní odchylky a výpočetního času vylepšované implementace CGP (očekávány jsou hodnoty ve formátu *double*, a musejí být zadány v tomto pořadí). Jako parametr čtyři a více jsou očekávána jména trénovacích vstupních posloupností určených pro zpracování vylepšovanou implementací CGP. Minimální počet takovýchto vstupních souborů je jedna, maximální počet není omezen.

V horní části zdrojového kódu jsou umístěny konstanty pro nastavení parametrů evoluce programu. Konkrétně jde o velikost populace, počet generací, počet a velikost mutací. Kromě těchto konstant program obsahuje také také konstantu `VYPISY` pro nastavení množství výpisů.

Populace programu je implementována jako číselný vektor, jehož hodnoty mohou nabývat buď typů *integer* nebo *double*. Typ hodnoty je určen na základě přítomnosti desetinné tečky a zůstává neměnný po celou dobu evoluce. Důvodem pro rozlišení mezi těmito dvěma datovými typy je, že ve vylepšované implementaci mohou vylepšované parametry sloužit rozdílným účelům. Pokud jde například o konstantu CGP, je možné pro aproximaci desetinných hodnot využít čísel s plovoucí desetinnou čárkou. Pokud je vylepšovaná hodnota použita například pro nastavení velikosti mřížky CGP, hodnota s desetinnou čárkou by nedávala smysl. Datový typ tak omezuje množství hodnot, kterých může daná pozice vektoru hodnot nabývat.

Běh optimalizátoru probíhá způsobem, který je ukázán na obrázku 7.1. Nejprve předpokládáme, že do zdrojového kódu vylepšované implementace CGP byly člověkem, případně nějakým automatizovaným systémem, umístěny značky, jejichž význam byl popsán v předchozí kapitole (sekce 6.2). Optimalizátor nejprve zavolá skript pro extrakci označených hodnot z originální implementace CGP, tyto hodnoty standardním způsobem zmutuje a použije je jako počáteční populaci. Po vytvoření populace program pro každého člena

populace zavolá skript pro úpravu a překlad zdrojového kódu (implementace CGP), který se postará o vytvoření nových zdrojových souborů a jejich překlad. Optimalizátor poté přeložené modifikované implementace CGP pouští na úlohách, které měl zadány jako součást svých vstupních parametrů, měří výpočetní čas, který spuštěná implementace CGP k vyřešení všech úloh potřebuje, a jaké celkové odchylky výpočtu (suma návratových hodnot) se při tom dopustí. Na základě těchto hodnot poté spočte fitness jedinců populace.

Samotná funkce fitness je definována následujícím způsobem:

$$Fitness(i) = (P_k + P_i) * (T_k + T_i)$$

Hodny P_k a T_k přestavují konstantní hodnoty míry vlivu přesnosti výpočtu (respektive výpočetního času) na celkovou fitness jedince. Hodnoty P_i a T_i pak představují sumu odchylek (respektive výpočetních časů), které pro daného jedince i optimalizátor při jeho běhu na testovacích úlohách naměřil. Abychom rovnici vysvětlili, předpokládejme na okamžik, že přesnost výpočtu i výpočetní čas stejné implementace řešící stejnou úlohu na stejném stroji je vždy stejný. Pokud nastavíme hodnoty konstant na úroveň, která odpovídá skutečným výsledkům, které implementace na daných úlohách poskytuje (tedy nastavíme $P_k = P_i$ a $T_k = T_i$), pak snížení výpočetní odchylky nebo výpočetního času na polovinu sníží (tedy zlepší) celkovou fitness implementace o jednu čtvrtinu (původně $Fitness(i) = (1 + 1) * (1 + 1) = 4$, po úpravě $Fitness(i') = (1 + 0.5) * (1 + 1) = 3$). A nezáleží přitom, zda vylepšeným parametrem byla přesnost výpočtu nebo jeho výpočetní čas. Pokud však jednu z konstant, dejme tomu P_k, nastavíme na desetinásobek, dojde při snížení výpočetní odchylky ke snížení celkové fitness pouze o necelých pět procent (původně $Fitness(i) = (10 + 1) * (1 + 1) = 22$, po úpravě $Fitness(i') = (10 + 0.5) * (1 + 1) = 21$).

Ve výsledku to znamená, že nastavením obou konstant na jejich očekávané hodnoty (které zjistíme tak, že trénovací úlohy necháme nejprve vyřešit originální implementací CGP, kterou budeme poté vylepšovat), dosáhneme toho, že zlepšení přesnosti výpočtu i výpočetního času se ve výsledku projeví jako stejně důležité. Manipulací s těmito konstantami pak můžeme dosáhnout zvětšení nebo zmenšení jejich vlivu na celkovou fitness a tím nastavit, jak moc se optimalizátor má na tyto parametry v průběhu genetického vylepšování zaměřit.

Důvodem proč není použita jediná váha určující poměr obou parametrů je poskytnou uživateli více možností, jak optimalizátor nastavit. Pokud uživatel obě konstanty zvýší, bude optimalizátor preferovat vyvážené vylepšení obou parametrů, pokud obě hodnoty sníží, optimalizátor bude preferovat výrazné zlepšení jednoho z parametrů i na úkor toho druhého.

Problémem je, že předpoklad, že stejná implementace poskytne při řešení stejné úlohy vždy stejné výsledky, není pravdivý. Protože je ale výsledek CGP závislý na náhodě, a rychlost výpočtu je závislá na parametrech jako je momentální vytížení stroje na kterém je výpočet prováděn, není možné tento předpoklad nikdy splnit. Můžeme se mu však přiblížit prováděním opakovaných testů a použitím průměrných nebo jinak statisticky stabilnějších hodnot.

Po výpočtu fitness optimalizátor provede zálohu jedince s nejlepší dosud dosaženou hodnotou fitness, a pokud nedošlo k dosažené maximálního nastaveného počtu generací, přejde k výběru rodičů. Ten probíhá metodou turnaje. Z populace jsou vybráni dva jedinci (může jít i o toho samého) a ten s lepší hodnotu fitness se stává prvním rodičem prvního potomka. Poté jsou vybráni další dva jedinci současné populace, a ten s lepší fitness se stává druhým rodičem prvního potomka. Tento postup je opakován tolikrát, dokud nejsou vybráni rodiče pro každého potomka příští generace.

Po výběru rodičů následuje křížení. Pro každou z jeho hodnot je vygenerováno náhodné číslo X v rozsahu $[-0.25, 1.25]$. A hodnota potomka je spočtena jako X-násobek odpovídající hodnoty prvního rodiče, a $(1 - X)$-násobek hodnoty rodiče druhého. V případě že datovým typem křížené hodnoty je *integer*, je výsledek ještě zaokrouhlen na nejbližší platnou hodnotu. Implementace také obsahuje ochranu, která zaručuje, že budou vždy generovány pouze kladné hodnoty. Nulová, nebo záporná hodnota typu *integer*, která bývá používána pro nastavení velikosti mřížky nebo populace CGP, případně počítadla cyklů, by neměla smysl. U hodnot typu *double* je kladná hodnota vyžadována proto, že pokud by nějaká hodnota byla nastavena na nulu, nebylo by možné ji díky způsobu jakým je křížení (a později mutace) implementováno, dále změnit. V případě, že implementace vyžaduje záporná čísla, je možné jich dosáhnout umístěním značky do vylepšované implementace CGP mezi znaménko minus a hodnotu samotnou.

Po křížení jsou noví jedinci ještě zmutováni. U každého jedince dojde několikrát (v závislosti na nastavení parametrů optimalizátoru) k výběru jedné z hodnot vektoru a jeho náhodné změně. Tato změna je implementována tak, že se nejprve určí zda bude hodnota zvýšena nebo snížena (obojí se může stát se stejnou pravděpodobností), a poté bude vynásobena nebo vydělena hodnotu v rozsahu $[1, k]$. Parametr k určující míru randomizace, je jedním z parametrů evoluce optimalizátoru. Pokud je prováděno mutací několik, může být ta samá hodnota mutována i vícekrát. Podobně jako u výpočtu hodnot při křížení, i mutace obsahuje ochranu před nulou a zápornými čísly, a to ze stejných důvodů.

Po provedení mutace se evoluční cyklus následně opakuje tak dlouho, dokud není proveden požadovaný počet generací. Výstupem optimalizátoru jsou (kromě volitelných textových výpisů) tři soubory, *changes_best.txt* obsahující vektor hodnot nejlepší nalezené implementace CGP, *modified_best.cpp* obsahující upravený zdrojový kód a *modified_best.exe* představující odpovídající vylepšenou implementaci CGP přeloženou takovým způsobem, jaký byl použit při jejím testování.

7.3 Skript pro extrakci označených hodnot

Skript pro extrakci označených hodnot je umístěn v souboru *extractor.py*. Má dva vstupní parametry. Prvním je jméno originální implementace CGP, ze které mají být označené hodnoty extrahovány, druhým je jméno souboru, kam mají být extrahované hodnoty uloženy. Tento skript není pouštěn člověkem, optimalizátor jej dle potřeby (při vytvoření počáteční populace) volá automaticky.

Skript pracuje tak, že otevře potřebné soubory, a řádek po řádku pochází zdrojový kód originální implementace, ve které pomocí regulárních výrazů hledá umístěné značky. Označené hodnoty pak přepisuje do výstupního souboru. Hledané značky mají podobu víceřádkových komentářů ve formátu `/*>*/X/*<*/`, kde znak `X` značí hodnotu která bude tímto skriptem extrahována. Důvodem, proč byly pro značky zvoleny víceřádkové komentáře je, že z pohledu jazyka, ve kterém jsou použity jsou průhledné, a lze je tady umístit prakticky kamkoliv bez změny funkčnosti originálního kódu.

Důvodem, proč tato funkcionalita nebyla zahrnuta přímo do zdrojového kódu samotného optimalizátoru je, že úpravou tohoto by bylo možno extrahovat i jiné typy značek, než které byly v rámci tohoto projektu použity, a umožnit tím optimalizátoru geneticky vylepšovat i implementace CGP napsané v jiných jazycích něž C++.

7.4 Skript pro úpravu a překlad zdrojového kódu

Skript pro úpravy zdrojového kódu je umístěn v souboru *replacery.py*. Má tři vstupní parametry: jméno souboru obsahujícího originální implementaci CGP, jméno souboru s novými geneticky vylepšenými hodnotami, které do nevylepšené implementace umístí, a jméno výstupního souboru, do kterého bude zdrojový kód vylepšené implementace zapsán. Tento skript není pouštěn člověkem, optimalizátor jej dle potřeby volá automaticky.

Skript pracuje tak, že otevře potřebné soubory, a prochází zdrojový kód, ve kterém pomocí regulárních výrazů hledá značky, jejichž obsah nahrazuje hodnotami ze souboru změn, které jsou v něm uvedeny ve stejném počtu i pořadí jako byly v originální implementaci. Po přepsání celého zdrojového kódu implementace CGP je poté volán překladač *G++*. Použitými parametry jsou *-Wfatal-errors*, který byl přidán, protože při automatizovaném překladu nebude k dispozici programátor, který by případné nalezené chyby mohl analyzovat a opravit, a *-std=c++11*, který byl použit, protože originální implementace CGP využívala některých funkcí, které nejsou součástí dřívějších verzí jazyka C++.

Důvodem proč provádění úprav ve zdrojovém kódu nebylo přímo součástí implementace optimalizátoru je, že úpravou tohoto skriptu se dá změnit podoba hledaných značek, očekávaných přípon souborů, i nastavení překladače. A pouhou změnou tohoto skriptu je tedy možné optimalizátor použít pro genetické vylepšení implementací CGP provedených i v jiných programovacích jazycích než C++.

Kapitola 8

Experimentální ověření funkčnosti programu

Tato kapitola popisuje, jak byla implementace otestována, jakých výsledků dosáhla originální implementace a všechny její optimalizátorem vytvořené modifikace. Konkrétní hodnoty modifikovaných programů jsou umístěny v příloze. Testy byly prováděny na notebooku Acer Aspire E15, s čtyř-jádrovým procesorem AMD A10-7300, 1.9–3.2 GHz, 8 GB RAM, pod operačním systém Windows 10 Home (64-bit). Všechny testy sledovaly dva hlavní parametry, a to celkovou odchylku od požadovaného výsledku a výpočetní čas.

Protože výstup CGP je ovlivněn náhodou, je nemožné míru zlepšení určit zcela jednoznačně. Aby byl vliv náhody minimalizován, byla originální implementace i všechny její modifikace shodně nastaveny tak, aby vždy provedly 15 samostatných a zcela nezávislých běhů, ze kterých byl za konečný výsledek vybrán medián. Průměr nebyl použit, protože na rozdíl od mediánu může být snadno ovlivněna, odlehlými hodnotami.

Jako výsledek časové náročnosti programu byla použita suma dob výpočtu potřebných pro provedení všech 15 běhů. Předběžné testy ukázaly, že výpočetní čas jednotlivých běhů velmi výrazně kolísá a není nijak zřejmě závislý na kvalitě výsledků, které dané běhy poskytovaly. Suma dob výpočtu se pak ukázala jako výrazně stabilnější, a tedy spolehlivější ukazatel než výpočetní čas běhu, který poskytl "mediánový" výsledek i než medián vypočtený z dob výpočtu všech běhů.

8.1 Vstupní posloupnosti

Testování bylo prováděno na patnácti posloupnostech různé délky, vygenerovaných různě složitými matematickými vztahy. Všechny vztahy byly vytvořeny kombinací šesti matematických operací, kterých je implementace CGP schopna. Konstanty, které byly ve vztazích použity, se však neomezovaly na konstanty původní implementace. Rozlišovat mezi matematickými vztahy na základě operací, které využívají, je zcela běžné (polynomiální funkce, goniometrické funkce). Rozlišovat je ale podle hodnot, které se v nich mohou vyskytovat, nikoliv. Konstanty programu budou navíc jednou z vylepšovaných veličin, a testy by měly odhalit, jaké hodnoty konstant jsou nejužitečnější.

Celkem bylo pro testování použito patnáct vstupních posloupností. Prvních šest se od sebe výrazně odlišovalo a byly určeny pro testování schopnosti optimalizátoru vylepšit parametry originální implementace pro velmi rozdílné vstupy. Tyto posloupnosti se liší použitými funkcemi, svojí délkou a řády hodnot, které výstupní hodnoty nabývají. Další čtyři

posloupnosti byly navrženy tak, aby se lišily pouze svojí složitostí a shodovaly se svojí délkou i velikostí výstupů, a byly tedy snadno porovnatelné. Tyto posloupnosti jsou určeny pro optimalizaci programu na několika posloupnostech současně. Kdyby se posloupnosti výrazně lišily, mohla by se vylepšování programu zaměřit pouze na zlepšení výsledků té posloupnosti s řádově největšími výstupy nebo nejdelším potřebným časem, a ostatní méně významné posloupnosti zanedbat, což by znehodnotilo význam optimalizace na několika posloupnostech současně. Zbývajících pět posloupností je pak použito k závěrečnému porovnání originálního CGP s nejlepší modifikovanou implementací, kterou optimalizátor v rámci testů vytvořil.

Konkrétní matematické vztahy, které byly pro generování vstupních posloupností použity, spolu s jejich obory hodnot, jsou součástí přílohy A.1. Ve zbytku této kapitoly budou všechny posloupnosti označovány pouze jejich pořadovým kódovým číslem (1-15).

8.2 Hodnoty modifikované v originální implementaci

Při zanášení značek do originální implementace CGP jsme se rozhodli označit ty hodnoty, u niž víme, že jejich změna má z hlediska programu smysl. Nebudeme tak například upravovat očekávaný počet vstupních parametrů nebo velikost vstupního bufferu, který implementace CGP obsahuje. Ze všech upravitelných hodnot jsme pak vybrali ty, u kterých jsme očekávali, že budou mít největší vliv na výstupy a dobu běhu programu. Aby byly prováděné testy vzájemně porovnatelné, byly počáteční hodnoty originální implementace při všech bězích optimalizátoru stejné. Význam vybraných parametrů a jejich hodnoty jsou následující:

- **Velikost populace** – 5
- **Počet mutací** – 3
- **Počet sloupců** – 4
- **Počet řádků** – 4
- **Levels-back** – 4
- **Počet generací** – 10000
- **1. konstantní funkce CGP** – 0,5
- **2. konstantní funkce CGP** – 1,0
- **3. konstantní funkce CGP** – 2,0
- **4. konstantní funkce CGP** – 10,0

Neobvykle nízký počet generací byl u originální implementace zvolen z časových důvodů. Při vylepšování implementace bude běh programu mnohonásobně opakován a je tedy potřeba, aby jednotlivé běhy skončily v přijatelném čase.

8.3 Nastavení parametrů

Protože vylepšování originální implementace pracuje na principu křížení číselných vektorů a nikoliv na principu stavby syntaktických stromů, byla při nastavování parametrů optimalizátoru zvolena malá populace, pro niž jsme poté hledali dostatečný počet generací. Předběžné testy ukázaly, že pro populaci dané velikosti dochází k výraznému zlepšení výsledků již v několika prvních generacích. Protože nehledáme optimální řešení, ale snažíme se ověřit schopnost optimalizátoru vylepšit původní implementaci, ponechali jsme i počet generací optimalizátoru na nízké hodnotě. Parametry mutací, jejich počet a velikost, byly nastaveny tak, aby v parametrech optimalizovaného programu docházelo k velkému počtu malých změn. Aby byly všechny výsledky porovnatelné, byly při všech testech parametry optimalizátoru nastaveny shodně, následujícím způsobem:

- **Velikost populace** – 5
- **Počet generací** – 20
- **Počet mutací** – 3
- **Míra mutace** – 2.0

8.4 Testy originální implementace CGP

První provedenou sadou testů byly testy originální implementace CGP. Účelem těchto testů bylo získat vzorové hodnoty, se kterými budou výsledky všech pozdějších testů porovnávány a které poslouží pro nastavení vstupních parametrů optimalizátoru v ostatních testech. Nalezení vhodného kompromisu mezi přesností výstupu (chybou) a dobou výpočtu vyžaduje znalost jejich původních hodnot a jejich poměru. Právě pro jejich získání tato sada testů slouží.

Protože hodnoty odchylky výpočtu závisejí na konkrétních vstupních posloupnostech, očekává se, že se budou nacházet v řádech desítek až desítek tisíc, v závislosti na délce vstupní posloupnosti a velikosti požadovaných výstupů. Vzhledem k tomu, že počet generací i velikost výpočetní mřížky (počet výpočetních uzlů) CGP je pro všechny testy stejná, by čas potřebný pro získání výsledku měl záviset pouze na počtu vstupních hodnot jednotlivých posloupností.

Pro použitý testovací stroj a vstupní parametry by výsledky všech testů měly být získány v rámci jednotek sekund. Vliv na dobu výpočtu by mohlo mít i to, jaké funkce byly pro vygenerování vstupních posloupností použity (výpočet funkce sinus zabere výrazně více času než výpočet součtu). Výsledné řešení ale zpravidla využívá méně výpočetních uzlů než kolik výpočetní mřížka CGP obsahuje uzlů, a u každého jedince jsou počítány výstupy všech uzlů, tedy i těch, které na výstup nemají vliv. Vliv matematického vztahu, který byl na vytvoření vstupní posloupnosti použit, by tedy měl být zanedbatelný.

Testy originální implementace uvedené v tabulce 8.1 ukázaly, že předpoklady byly správné. Největší výpočetní odchylka byla nalezena u funkcí s velkým rozpětím výstupních hodnot. Malá výpočetní odchylka je naopak viditelná u jednoduchých posloupností, jejichž regrese se originální implementaci CGP úspěšně zdařila, a u posloupností s malým rozpětím výstupních hodnot. Pro připomenutí, výpočetní odchylkou rozumíme sumu čtverců odchylek hodnot vypočtených na základě funkce sestavené symbolickou regresí a hodnot předepsaných vstupní posloupností o jejíž symbolickou regresi se pokoušíme. Výpočetní

vstup	výpočetní odchylka	výpočetní čas [s]
1	10	6.889
2	22960	9.676
3	51	5.020
4	26060	8.086
5	648	4.794
6	2401	7.128
7	1985	5.459
8	2002	5.280
9	1433	5.399
10	1794	5.005
11	7497	7.916
12	7939	7.889
13	9375	8.167
14	6407	8.675
15	8016	8.976

Tabulka 8.1: Testy výpočetní odchylky a rychlosti výpočtu originální implementace CGP na různých vstupních posloupnostech. V tomto i dalších testech prováděných na této implementaci CGP, je výpočetní odchylka mediánem z patnácti samostatných běhů. Výpočetní čas je sumou pro těchto patnáct běhů.

časy potřebné pro dokončení regrese jsou v rámci sekund. V souladu s předpoklady se zdají být ovlivněny pouze počtem hodnot vstupní posloupnosti a náhodou.

8.5 Vylepšení implementace na základě jedné vstupní posloupnosti

Druhá sada testů byla zaměřena na schopnost optimalizátoru vylepšit hodnoty originální implementace tak, aby vyhovovaly potřebám regrese jednotlivých funkcí. Takto získané výsledky nejsou zobecnitelné na schopnost optimalizovaného programu provádět regresi jiných funkcí, ani to není jejich cílem. Každý z těchto testů bude proveden ve třech variantách. Při první variantě bude optimalizátor nastaven tak, aby vylepšení přesnosti symbolické regrese věnoval desetkrát větší váhu než zlepšení její rychlosti. Očekává se, že u funkcí dojde ke zlepšení nebo zachování funkčnosti a zachování či zhoršení výpočetního času. V druhé variantě bude zlepšování přesnosti i výpočetního času věnována stejná váha a očekávává se, že dojde ke zlepšení alespoň jednoho z těchto parametrů. Ve třetí variantě pak bude vylepšení výpočetního času považováno za desetkrát důležitější než vylepšení přesnosti výpočtu, a očekává se, že výpočetní čas bude zkrácen na úkor přesnosti.

Cílem této sady testů je zjistit maximální schopnosti optimalizátoru při použitém nastavení. Získané výsledky pak budou sloužit pro porovnání s pozdějšími testy optimalizátoru, provedenými na několika funkcích současně, a na jiných implementacích CGP. Bez ohledu na variantu testu se očekává, že jakékoliv případné zhoršení jednoho ze sledovaných parametrů bude více než vyváženo zlepšením toho druhého a vylepšená implementace CGP bude celkově lepší než originál. Testy budou provedeny na prvních šesti posloupnostech.

První varianta testů zaměřená na minimalizaci výpočetní odchylky uvedená v tabulce

	výpočetní odchylka		výpočetní čas [s]	
vstup	hodnota	změna	hodnota	změna
1	0	-100 %	3.202	-53.52 %
2	23	-99.9 %	37.739	+290.03 %
3	51	0 %	0.791	-84.24 %
4	9921	-61.93 %	4.178	-48.33 %
5	18	-97.22 %	6.926	+44.47 %
6	2310	-3.79 %	2.444	-65.71 %

Tabulka 8.2: Výsledky modifikovaných implementací CGP, po vylepšení se zaměřením na snížení odchylky výpočtu.

8.2 jasně ukázala schopnost optimalizátoru snížit výpočetní odchylku výsledků, které implementace CGP poskytuje. V polovině případů bylo zlepšení přesnosti více jak desetinásobné a v žádném z případů se nestalo, že by se přesnost výpočtu snížila. U výpočetního času byly změny menší. Ve většině případů došlo ke zhruba dvojnásobnému snížení výpočetního času, ve zbytku případů však výpočet vylepšené implementaci trval déle než implementaci originální. V těchto případech však byl nárůst výpočetního času více než vyvážen řádovým zlepšením přesnosti výpočtu. Je tedy patrné, že ve všech případech byla implementace celkově vylepšena.

Z vektorů hodnot geneticky vylepšených parametrů CGP uvedených v příloze B je zřejmé, že při zaměření programu na přesnost výsledku došlo ve většině případů ke snížení velikosti populace CGP, nárůstu velikosti mřížky a odpovídajícímu nárůstu parametru levels-back. Překvapením bylo snížení počtu generací a velmi výrazný nárůst počtu prováděných mutací.

	výpočetní odchylka		výpočetní čas [s]	
vstup	hodnota	změna	hodnota	změna
1	0	-100 %	1.855	-73.07 %
2	636	-97.23 %	25.937	+168.05 %
3	51	0 %	0.529	-89.46 %
4	26130	+0,27 %	3.227	-60.09 %
5	13	-97.99 %	12.743	+165.81 %
6	2599	+7.84 %	0.693	-90.27 %

Tabulka 8.3: Výsledky modifikovaných implementací CGP, po vylepšení se zaměřením na vyvážené snížení výpočetního času a odchylky výpočtu.

Druhá varianta testů jejíž výsledky jsou uvedeny v tabulce 8.3, kladla stejnou váhu na zlepšení výpočetní odchylky i výpočetního času, měla smíšené výsledky. V jedné polovině případů došlo k více jak desetinásobnému zlepšení přesnosti, většinou však na úkor znatelného zvýšení doby výpočtu. V druhé polovině případů došlo k výraznému zlepšení výpočetního času, většinou na úkor malého zhoršení přesnosti výpočtu. Všechna zlepšení však více jak desetinásobně převýšila případná zhoršení. I tato varianta testu tak prokázala celkové vylepšení funkčnosti implementace CGP.

Velikost populace i počet mutací se měnily oběma směry, což je významná odlišnost od modifikací, zaměřených na zvýšení přesnosti výpočtu. Velikost výpočetní mřížky a parametr levels-back většinou narostly, počet generací se naopak snížil a to výrazněji, než u optimalizace prováděné pouze na snížení výpočetní odchylky programu.

vstupní posloupnost	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
1	41	+310 %	1.476	-78.57 %
2	22960	0 %	1.749	-81.92 %
3	51	0 %	1.883	-62.49 %
4	26494	+1.67 %	0.934	-88.45 %
5	2329	+259.41 %	0.031	-99.35 %
6	2599	+8.24 %	0.187	-97.37 %

Tabulka 8.4: Výsledky modifikovaných implementací CGP, po vylepšení se zaměřením na snížení výpočetního času.

U třetí varianty testů jejíž výsledky jsou uvedené v tabulce 8.4 došlo ve všech případech k výraznému snížení výpočetního času. V žádném z testovaných případů se však nepodařilo snížit výpočetní odchylku. Jakékoliv zhoršení přesnosti výpočtu bylo více jak vyváženo získaným zrychlením a získané výsledky tak splnily očekávané předpoklady.

Ve všech případech došlo v parametrech ke snížení velikosti populace na hodnotu dva, což je minimální možná velikost populace, při které je CGP schopno pracovat. U počtu mutací došlo ke změnám oběma směry. Velikost výpočetní mřížky byla ve všech případech snížena, a ve většině případů se vyskytlo i značné snížení počtu generací.

Všechny tři varianty testů ukázaly schopnost optimalizátoru vylepšit vlastnosti implementace tak, aby došlo ke zlepšení parametrů, na které se měl zaměřit. Způsob jakým byly testy provedeny však umožňoval optimalizátoru vylepšit implementaci pro potřeby konkrétních vstupních posloupností a výsledky tak na základě pouze těchto testů nelze zobecnit.

8.6 Testy implementací vylepšených na základě jedné vstupní posloupnosti

Třetí sada testů vyzkouší vylepšení implementace CGP vytvořené v rámci předchozího testu na dalších posloupnostech. Cílem je zjistit, zda optimalizátorem vylepšené hodnoty vyhovují pouze potřebám regrese jedné konkrétní posloupnosti, nebo jestli se zlepšení projeví i při použití vylepšené implementace na posloupnostech ostatních. Očekávaným výsledkem je, že měřené zlepšení bude výrazně menší než v předchozích testech a v ojedinělých případech by mohlo dojít i ke zhoršení parametrů oproti původní implementaci. Celkově by však vylepšení parametrů mělo být stále znatelné.

Aby byly výsledky testu vzájemně porovnatelné nejen s výsledky původní implementace, ale i mezi sebou, nebudou testovány na posloupnostech použitých k jejich stvoření, ale na čtveřici dalších nezávislých posloupností. Protože nás zajímá především celkové zlepšení přesnosti výpočtu a výpočetního času, budou výsledky pro všechny čtyři vstupní posloupnosti agregovány. Konkrétní hodnoty pro jednotlivé vstupní posloupnosti jsou uvedeny v příloze C.

modifikace implementace	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
originál	7214	0 %	21.143	0 %
1	4884	-32.3 %	50.352	+138.15 %
2	2205	-69.43 %	72.654	+243.63 %
3	8251	+14.51 %	1.705	-91.94 %
4	4222	-41.47 %	6.743	-68.11 %
5	2727	-62.2 %	20.663	-2.27 %
6	7506	+4.05 %	4.996	-76.37 %

Tabulka 8.5: Výsledky implementací CGP vylepšených se zaměřením na velikost výpočetní odchylky, při testech na dalších vstupních posloupnostech.

Testy modifikovaných implementací CGP jejichž vylepšení se zaměřovalo na snížení výpočetní odchylky při použití na dalších funkcích uvedené v tabulce 8.5 ukázaly, že výhody získané modifikací na přesnost prováděné na jediné posloupnosti, nejsou stabilní. Přesto, že většina modifikovaných implementací vykazovala zlepšení alespoň jednoho z měřených parametrů, ve třetině případů došlo ke zvýšení výpočetní odchylky, na kterou se měla optimalizace zaměřit. Výsledky tak sice jsou slabě pozitivní, jejich celkový přínos je ale diskutabilní.

modifikace implementace	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
originál	7214	0 %	21.143	0 %
1	3742	-48.13 %	19.309	-8.67 %
2	3784	-47.55 %	51.262	+142.45 %
3	7868	+9.07 %	1.430	-93.24 %
4	7937	+10.02 %	5.826	-72.44 %
5	3498	-51.51 %	41.28	+95.24 %
6	9106	+26.23 %	2.104	-90.05 %

Tabulka 8.6: Výsledky implementací CGP vylepšených se zaměřením na vyvážené snížení výpočetní odchylky a výpočetního času, při testech na dalších vstupních posloupnostech.

Testy modifikací CGP snažících se o vyvážené snížení výpočetní odchylky a výpočetního času uvedené v tabulce 8.6 dopadly o něco lépe. Ve všech případech došlo ke zlepšení alespoň jednoho z parametrů. S výjimkou jediného případu se tak ale vždy stalo na úkor parametru druhého. Nebylo tedy dosaženo ani zdaleka tak dobrých výsledků, jako u odpovídající sady testů z předcházející podkapitoly.

Testy modifikací zaměřených na snížení výpočetního času uvedené v tabulce 8.7 poskytly velmi stabilní výsledky. Ve všech případech došlo k mírnému zhoršení přesnosti (zhruba o 15 %) a výraznému snížení výpočetního času (okolo 90 %). To je značný rozdíl v chování oproti modifikacím částečně nebo zcela zaměřeným na zvýšení přesnosti výpočtu, které po aplikaci na další posloupnosti poskytly velmi nestabilní a obecně horší výsledky.

Cekově z třetí sady testů usuzujeme, že při modifikaci implementace na základě pouze jediné vstupní posloupnosti lze dosáhnout obecného zlepšení výpočetního času, ale nikoliv přesnosti výpočtu. Toto zjištění vysvětlujeme tak, že posloupnosti generované různými ma-

modifikace implementace	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
originál	7214	0 %	21.143	0 %
1	8359	+15.87 %	2.385	-88.72 %
2	8427	+16.81 %	2.248	-89.37 %
3	8338	+15.58 %	5.575	-73.63 %
4	7597	+5.31 %	2.091	-90.11 %
5	8966	+24.29 %	0.822	-96.11 %
6	9092	+26.03 %	0.39	-98.16 %

Tabulka 8.7: Výsledky implementací CGP vylepšených se zaměřením na snížení výpočetního času, při testech na dalších vstupních posloupnostech.

tematickými vztahy vyžadují pro svoje efektivní řešení různé nastavení parametrů. Zatímco posloupnosti generované složitějšími matematickými vztahy vyžadují pro správné vyřešení především větší výpočetní mřížku, posloupnosti generované jednoduššími funkcemi, pro než má výpočetní mřížka dostatečně velkou rezervu, vyžadují především nárůst počtu generací.

Pokud je ale cílem optimalizace snížení výpočetního času, bude menší výpočetní mřížka a nižší počet generací vždy vyhodnocen rychleji, a to bez ohledu na to, jaká funkce byla pro vytvoření vstupní posloupnosti použita. Dalším faktorem, který je možné vidět z tabulky vektorů hodnot jednotlivých modifikací uvedené v příloze je také fakt, že všechny modifikace zaměřené na snížení výpočetního času, shodně snížily velikost populace CGP na pouhé dva členy. Tím se výrazně odlišují od modifikací zaměřených na přesnost výpočtu nebo vyvážené zlepšení obou parametrů, které byly vzájemně více rozdílné. Je logické, že implementace s podobnými parametry budou na stejných vstupních posloupnostech dosahovat podobných výsledků. Původní očekávání této sady testů tak byla splněna pouze částečně, ale všechny odchylky jsou logicky zdůvodnitelné.

8.7 Vylepšení implementace na základě několika vstupních posloupností současně

Čtvrtá sada testů je určena pro ověření schopnosti optimalizátoru upravit originální implementaci tak, aby poskytovala lepší výsledky pro více posloupností současně a nebyla tak vylepšována pouze na řešení jednoho konkrétního problému. Což, jak ukázaly testy provedené v předchozí sekci, vedlo k výsledkům, které se, s výjimkou modifikací zaměřených na snížení výpočetního času, nepodařilo stabilně reprodukovat při použití na dalších vstupních posloupnostech.

Jako vstup testů v této sekci byly použity čtyři vstupní posloupnosti se srovnatelnou časovou náročností, na kterých originální implementace dosahovala zhruba stejné výpočetní odchylky. Optimalizace bude opět provedena ve třech variantách. První bude přikládat desetinásobnou váhu přesnosti výpočtu, druhá bude snížení výpočetní odchylky a času přikládat stejnou váhu a třetí bude rychlost výpočtu považovat za desetkrát důležitější než jeho přesnost. Jako referenční hodnota použitá pro nastavení optimalizátoru byla použita suma výsledných výpočetních odchylek a časů, kterých na daných vstupních posloupnostech dosáhla originální implementace CGP.

Očekává se, že zlepšení přesnosti výpočtu i výpočetního času nebude ani zdaleka tak velké, jako když byla implementace optimalizována pouze na jedinou vstupní posloupnost. I tak by však změna měla být znatelná a především reprodukovatelná i po použití vylepšené implementace CGP na jiných vstupních posloupnostech v rámci následných testů. Od vektorů hodnot se očekává, že v nich dojde k podobným změnám jako u předchozích modifikací vylepšených zaměřením optimalizátoru na stejné parametry.

Jak bylo popsáno v implementaci projektu (kapitola 7), jsou modifikované implementace testovány na vstupních posloupnostech v průběhu evoluce střídavě. Neagregované výsledky vylepšených implementací CGP na jednotlivých posloupnostech tak nejsou známy a na rozdíl od testů z předchozí podkapitoly nejsou součástí přílohy.

zaměření vylepšení	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
originální implementace	7214	0 %	21.143	0 %
přesnost výpočtu	3067	-57.49 %	32.448	+53.47 %
přesnost i rychlost	2970	-58.83 %	8.777	-58.49 %
rychlost výpočtu	8629	+19.61 %	1.147	-94.57 %

Tabulka 8.8: Výsledky implementací CGP optimalizovaných na několika vstupních posloupnostech současně v porovnání s výsledky originální implementace CGP.

Výsledky testu uvedené v tabulce 8.8 ukázaly, že implementace CGP při jejímž vylepšování se optimalizátor zaměřil na snížení výpočetní odchylky opravdu dosáhla jejího snížení. Dle tabulky hodnot vektorů uvedené v příloze B.1 toho bylo dosaženo kombinací zvětšení výpočetní mřížky a počtu generací evoluce. Neúměrnému zvýšení výpočetního času pak bylo zabráněno snížením velikosti populace. Stejně jako u všech ostatních modifikací zaměřených na zlepšení přesnosti výpočtu pak došlo také k výraznému nárůstu počtu mutací, který v tomto případě zhruba odpovídá zvětšení velikosti výpočetní mřížky.

Modifikace zaměřená na vyvážené zlepšení obou parametrů poskytla velmi dobré výsledky. Přesnost i rychlost výpočtu této implementace CGP byly více jak zdvojnásobeny. Implementace byla dokonce ještě přesnější, než implementace vytvořená optimalizací zaměřenou především na přesnost výpočtu. Od té se vektor hodnot liší především menší populací a výrazně nižším počtem generací. Parametrem, který je naopak neúměrně navýšen, je počet prováděných mutací, který se tak znovu ukazuje jako zásadní pro přesnost a rychlost výpočtu symbolické regrese.

Modifikovaná implementace CGP vytvořená při zaměření optimalizátoru na snížení výpočetního času měla velmi podobné výsledky jako ostatní implementace CGP které optimalizátor vytvořil při stejném zaměření v předchozích testech. Vektor hodnot ukazuje, že i u této modifikace bylo zvýšení rychlosti dosaženo zmenšením výpočetní mřížky, snížením počtu generací a zmenšením velikosti populace.

8.8 Testy implementací vylepšených na základě několika vstupních posloupností

Pátá sada testů bude zkoušet modifikované implementace CGP z předchozí sekce na dalších úlohách, které při jejich tvorbě nebyly použity. Cílem je zjistit, zda zlepšení výstupních parametrů zůstane zachováno a zda modifikované implementace CGP jsou skutečně poskytují přesnější výsledky v nižším výpočetním čase, i na dalších vstupních posloupnostech. Podobně, jako u testů prováděných na modifikovaných implementacích vytvořených na základě jedné vstupní posloupnosti se očekává, že výsledky budou o něco horší, než při použití na úlohách na kterých byly vytvořeny. Míra zhoršení by však měla být výrazně menší a výsledky dosažené na jednotlivých úlohách by se měly vzájemně více podobat.

K otestování geneticky vylepšených implementací CGP budou použity posloupnosti, které byly dříve použity pro tvorbu vylepšených implementací CGP z druhé sady testů (sekce 8.5). Toto nám umožní modifikace porovnat nejen oproti originální implementaci CGP, ale i oproti ostatním dříve vytvořeným vylepšeným verzím.

číslo vstupní	výpočetní odchylka		výpočetní čas [s]	
posloupnosti	hodnota	změna	hodnota	změna
1	0	-100 %	36.239	+426.04 %
2	4287	-81.33 %	32.839	+239.39 %
3	48	-5.88 %	20.722	+312.79 %
4	3607	-86.16 %	19.426	+140.24 %
5	36	-94.44 %	14.878	+210.35 %
6	2216	-7.71 %	21.466	+201.15 %

Tabulka 8.9: Výsledky implementace CGP vylepšené se zaměřením na velikost výpočetní odchylky, při testech na dalších vstupních posloupnostech.

Testy vylepšené implementace CGP vytvořené při zaměření na snížení výpočetní odchylky uvedené v tabulce 8.9 ukázaly, že si vytvořená implementace CGP svoje vlastnosti zachovává i při použití na jiných vstupních posloupnostech. Ve většině případů došlo k výraznému snížení výpočetní odchylky a v žádném z případů nebyla výpočetní odchylka zvýšena. Problémem se však ukázal být výpočetní čas. Ten narostl výrazně výše, než se čekalo. Ve všech případech došlo k jeho více jak dvojnásobnému nárůstu, což je výrazně, více než v testu z předchozí sekce. Modifikovaná implementace byla navíc výrazně pomalejší bez ohledu na to, zda se jí podařilo výrazně zlepšit přesnost výsledku, či nikoliv. Přestože modifikace vykazuje stabilní výsledky, pracuje v průměru ještě hůře, než modifikované implementace CGP, které které byly vytvořeny na základě jediné vstupní posloupnosti.

Výsledky testu implementace CGP vytvořené zaměřením na vyvážené zlepšení přesnosti výpočtu i výpočetního času uvedené v tabulce 8.10 ukázaly velmi dobré výsledky. Ve většině případů došlo ke zlepšení přesnosti výpočtu a ve všech případech došlo k výraznému snížení výpočetního času. Přestože zlepšení přesnosti výpočtu není tak výrazné, tato implementace pracovala ve všech případech více jak pětkrát rychleji než modifikace, která se zaměřovala pouze na zvýšení přesnosti výpočtu.

Tato vylepšená implementace CGP se tedy výrazně liší od implementací vytvořených na základě pouze jediné vstupní posloupnosti při stejném zaměření optimalizátoru, které ve většině případů vylepšily jeden z parametrů na úkor druhého. Schopnost této vylepšené implementace CGP spolehlivě zlepšit oba parametry ukazuje, že její výsledky jsou zobecnitelné a došlo k vytvoření stabilně lepší implementace.

číslo vstupní posloupnosti	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
1	11	+10 %	4.821	-30.02 %
2	15726	-31.51 %	5.653	-41.57 %
3	44	-13.73 %	3.233	-35.60 %
4	3782	-85.49 %	2.19	-72.92 %
5	199	-69.29 %	2.692	-43.85 %
6	2293	-4.50 %	2.692	-62.23 %

Tabulka 8.10: Výsledky implementace CGP vylepšené se zaměřením na vyvážené snížení výpočetní odchylky a výpočetního času, při testech na dalších vstupních posloupnostech.

číslo vstupní posloupnosti	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
1	369	+3590 %	0.037	-99.46 %
2	63663	+177.28 %	0.548	-94.34 %
3	61	+19.61 %	0.277	-94.48 %
4	5096	-80.45 %	0.265	-96.72 %
5	2312	+256.79 %	0.22	-95.41 %
6	2599	+8.25 %	0.329	-95.38 %

Tabulka 8.11: Výsledky implementace CGP vylepšené se zaměřením na snížení výpočetního času, při testech na dalších vstupních posloupnostech.

Tabulka 8.11 ukazuje výsledky testů implementace CGP kterou optimalizátor vytvořil při zaměření se na snížení výpočetního času, byla ve všech případech více jak desetkrát rychlejší než originální implementace, což je srovnatelné zrychlení jako u posloupností, na kterých byla optimalizována. Zrychlení je ještě větší než u implementací vylepšených na základě jediné vstupní posloupnosti, což ukazuje že zlepšení tohoto parametru je stabilnější.

Přesnost poskytnutého řešení však silně utrpěla. V polovině případů došlo k více jak dvojnásobnému zhoršení přesnosti a přestože přesnost byla v jednom případě oproti originální implementaci vylepšena, změna je průměrně horší než u implementací vylepšených na základě jediné posloupnosti a výrazně méně stabilní. Zhoršení jsou navíc natolik výrazná, že implementace zřejmě zcela ztratila svoji schopnost úspěšně aproximovat řešení některých problémů.

Celkově tato série testů ukázala, že změny rychlosti výpočtu jsou výrazně stabilnější, než změny v jeho přesnosti, kde rozsah změn velmi závisí na konkrétní vstupní posloupnosti. Ze tří testovaných vylepšených implementací se jako jednoznačně nejlepší ukázala implementace vytvořená při zaměření optimalizátoru na vyrovnané zlepšení obou parametrů.

8.9 Testy nejlepší vylepšené implementace na dalších úlohách

Šestá sada testů ze zaměřuje na porovnání originální implementace CGP s nejlepší dosud vytvořenou vylepšenou implementací CGP. Cílem je detailně ověřit, že implementace vytvořená optimalizátorem poskytuje přesnější výsledky v kratším čase než originální im-

plementace. Očekává se, že zlepšení přesnosti výpočtu i výpočetního času budou rámcově stejnéí jako u testů z předchozí sady. Pro zvýšení vypovídací hodnoty bude každý z testů proveden několikrát. Výsledky jednotlivých běhů jsou součástí přílohy C.

vstupní posloupnost	výpočetní odchylka			výpočetní čas [s]		
	originál	modifikace	změna	originál	modifikace	změna
11	7505.8	2848.2	-62.05 %	8.6524	2.5613	-70.4 %
12	7841.8	4521.7	-42.34 %	8.8444	2.5876	-70.74 %
13	10145.1	5205.5	-81.65 %	8.0853	2.5244	-69.17 %
14	5976.9	1096.8	-81.07 %	9.2178	2.8423	-66.12 %
15	6743.3	1276.4	-48.69 %	9.5179	3.2243	-68.78 %

Tabulka 8.12: Výsledky porovnání originální a nejlepší modifikované implementace CGP, která byla v rámci testů vytvořena, na dalších úlohách.

Výsledky provedeného testu jsou uvedeny v tabulce 8.12. Ukázalo se, že velikost odchylky byla u optimalizované implementace v průměru nižší o 63.16 % a výpočetní čas se v průměru snížil o 69.04 %. Předpoklad, že modifikovaná implementace bude poskytovat stabilně lepší výsledky, se tedy potvrdil. Optimalizátoru se podařil vylepšit implementaci CGP takovým způsobem, aby poskytovala lepší výsledky než originální implementace CGP v obou měřených parametrech. A tyto výsledky se podařilo ověřit i na jiných vstupních posloupnostech, než které byly pro vytvoření vylepšené implementace použity.

Za zmínku stojí také zjištění, že oběma implementacím zabraly nejdelší dobu výpočtu ty posloupnosti, které využívají goniometrické funkce, jak je vidět z tabulky vstupních posloupností uvedené v příloze A. Při opakovaných testech se tak jednoznačně projevil i vliv rozdílné doby výpočtu jednotlivých elementárních operací CGP.

8.10 Test optimalizátoru na jiné implementaci CGP

Sedmá sada testů ověří funkčnost optimalizátoru na převzaté implementaci CGP. Implementací, která bude při těchto testech použita, je neakcelerovaná implementace CGP pro symbolickou regresi, jejímž autorem je Zdeněk Vašíček [25]. V této implementaci byly před testováním provedeny následující změny:

- Do implementace bylo dodáno sedm značek. Čtyři byly umístěny okolo parametrů CGP, kterými jsou počet generací, počet sloupců výpočetní mřížky, počet mutací a velikost populace. Tři byly umístěny kolem konstantních funkcí CGP.
- Návratová hodnota úspěšného běhu programu byla změněna z nuly na výslednou hodnotu fitness.
- Počet generací CGP byl z časových důvodů snížen z původní hodnoty 500 000, na 50 000.

V první fázi testu byla implementace spuštěna na čtyřech vstupních posloupnostech, které budou následně použity jako vstup pro její vylepšení. Cílem tohoto testu je určit výpočetní odchylku a výpočetní čas této převzaté implementace CGP. Tyto hodnoty pak budou použity pro nastavení optimalizátoru, aby vylepšení obou výstupních parametrů mohlo být provedeno vyváženým způsobem.

Očekávaným výsledkem je, že převzatá implementace bude poskytovat přesnější výsledky, a to v kratším čase, než implementace vytvořené pro potřeby toho projektu. Důvodem tohoto předpokladu je, že převzatá implementace vrací jako výsledek nejlepší nalezený výsledek, nikoliv medián patnácti samostatných běhů. Všechny testy provedené na převzaté implementaci CGP tedy budou provedeny opakovaně.

vstupní posloupnost	výpočetní odchylka	průměrný čas výpočtu [s]
7	21.2	1.7318
8	0	1.3867
9	4.4	1.9066
10	63.2	1.8686
celkem	88.8	6.8937

Tabulka 8.13: Průměrné výsledky originální převzaté implementace CGP na vstupních posloupnostech 07-10. Kompletní sada výsledků je umístěna v příloze D.1.

Z výsledků uvedených v tabulce 8.13 vyplývá, že naše očekávání bylo správné a převzatá implementace poskytuje velmi přesné výsledky a pracuje rychleji než dříve zkoušená implementace. Tuto implementaci nyní zkusíme vylepšit pomocí optimalizátoru. Obdobně jako u stejného testu provedeného na vlastní implementaci CGP (viz sekce 8.7), bude optimalizace provedena ve třech verzích. Verze F, která klade snížení výpočetní odchylky desetkrát větší váhu než snížení výpočetního času. Verze C, která klade oběma vylepšovaným parametrům váhu stejnou. A verze T, která bude považovat snížení výpočetní odchylky za desetkrát méně důležité, než snížení výpočetního času. Vektory hodnot původní převzaté implementace CGP i všech jejích vylepšení jsou uvedeny v příloze D.2.

	výpočetní odchylka		výpočetní čas [s]	
implementace	hodnota	změna	hodnota	změna
originální	88.8	0 %	6.8937	0 %
verze F	37	-58.33 %	4.355	-36.83 %
verze C	87	-2.03 %	2.563	-62.82 %
verze T	1690	+1803.15 %	0.003	-99.96 %

Tabulka 8.14: Výsledky vylepšených implementací CGP na několika vstupních posloupnostech současně v porovnání s výsledky neupravené převzaté implementace CGP.

Při vylepšení zaměřeném na přesnost výsledku došlo ke snížení odchylky výsledku na polovinu, a současně došlo i k snížení výpočetního času, zhruba o jednu třetinu. Tabulka upravených hodnot ukazuje, že v implementaci došlo ke snížení velikosti populace, ale došlo k nárůstu velikosti výpočetní mřížky. K výrazným změnám došlo i u hodnot konstantních funkcí CGP. Jedna z funkcí výrazně narostla, čímž umožnila CGP aproximovat vysoké hodnoty s pomocí menšího počtu uzlů, a u ostatních konstant došlo k zvětšení jejich relativního rozestupu, tak aby lépe pokryly množinu čísel. Konstanty byly navíc upraveny tak, že podělením dvou z nich (konstanty A a B uvedené v příloze D.2) vznikne aproximace hodnoty π, kterou by z originálních hodnot bylo výrazně těžší aproximovat.

U vylepšení zaměřeného na vyvážené zlepšení přesnosti výpočtu a výpočetního času, došlo k zachování kvality výsledku, při výrazném snížení výpočetního času. Vzhledem k tomu, že přesnost výpočtu byla u originální implementace velmi vysoká, bylo toto chování očekávané. V chromozomu implementace došlo ke snížení velikosti výpočetní mřížky, a snížení počtu generací, a mírným změnám ostatních parametrů. K velkým změnám však došlo u konstantních funkcí CGP, které podobně jako u předchozího zaměření narostly, a vykompenzovaly tím snížení velikosti výpočetní mřížky při potřebě velkých konstant. Další funkce se přiblížila hodnotě 2π, která je velmi užitečná při výpočtu goniometrických funkcí, a nebyla přítomna v původní implementaci.

U vylepšení zaměřeného na rychlost výpočtu optimalizátor zcela obětoval schopnost implementace aproximovat výsledky snížením velikosti populace na jedna, čímž zabránil provádění jakýchkoliv mutací. Zaměřil se zcela na minimalizaci doby výpočtu, kterou snížil více jak tisícinásobně. Toto chování ukazuje, že přílišná snaha o snížení výpočetního času může zcela znehodnotit jeho funkceschopnost. Tomuto chování by šlo předejít změnou parametrů optimalizátoru, a snížením poměru důležitosti, který přikládá kvalitě a rychlosti získání výsledku.

Všechny tři vylepšené implementace nyní budou vyzkoušeny na řadě dalších vstupních posloupností a jejich výsledky budou porovnány s originální implementací. Očekávaným výsledkem testu je, že stejně jako v předchozích případech vylepšené implementace poskytnou menší zlepšení přesnosti výpočtu než u posloupností, na jejichž základě byly vylepšovány, zatímco zvýšení rychlosti výpočtu zůstane zachováno. Celkově by se měla potvrdit funkčnost optimalizátoru vytvořit implementaci, která celkově vykáže lepší vlastnosti než originální implementace. Každý z testů bude proveden desetkrát, konkrétní výsledky jsou součástí přílohy D.

vstupní	výpočetní odchylka			výpočetní čas [s]		
posloupnost	originál	modifikace	změna	originál	modifikace	změna
11	0	1.5	—	2.5385	1.4809	-41.66 %
12	0	0	—	2.5484	1.5373	-39.68 %
13	15.9	8.1	-49.06 %	3.259	1.5201	-53.36 %
14	18.5	14.6	-21.08 %	3.8041	1.8899	-50.32 %
15	3.2	3.4	+6.25 %	2.9401	1.7841	-39.32 %

Tabulka 8.15: Výsledky vylepšené převzaté implementace CGP v porovnání s originálem, při vylepšení zaměřeném na snížení odchylky výsledku.

Tabulka 8.15 ukazuje, že implementace vylepšená za účelem snížení odchylky od správného řešení byla v průměru schopna zachovat či mírně zlepšit kvalitu výsledku, a zachovala si svoji zvýšenou rychlost výpočtu.

Výsledky testu implementace, která byla vylepšována vyváženým způsobem v tabulce 8.16 ukazují, že u implementace došlo ke zhoršení odchylky výpočtu. Vzhledem k tomu, že odchylka výpočtu originální implementace je velmi malá, je procentuální zhoršení výsledku značné, určitá míra zhoršení přesnosti však byla očekávána. Zrychlení výpočtu zůstalo zachováno. Pokud odchylku porovnáme s zcela neaproximovanými výsledky z tabulky 8.17, vytvořenými modifikací zaměřenou pouze na rychlost výpočtu. Chyba kombinovaně vylepšované implementace je stále relativně malá.

vstupní posloupnost	výpočetní odchylka			výpočetní čas [s]		
	originál	modifikace	změna	originál	modifikace	změna
11	0	0.6	—	2.5385	0.7195	-71.66 %
12	0	0.9	—	2.5484	0.7814	-69.34 %
13	15.9	56.5	+255.35 %	3.259	0.8267	-74.63 %
14	18.5	27.4	+48.11 %	3.8041	1.1424	-69.97 %
15	3.2	3.4	+6.25 %	2.9401	0.7204	-75.5 %

Tabulka 8.16: Výsledky vylepšené převzaté implementace CGP v porovnání s originálem, při vylepšení zaměřeném na kombinované snížení odchylky výsledku a výpočetního času.

vstupní posloupnost	výpočetní odchylka			výpočetní čas [s]		
	originál	modifikace	změna	originál	modifikace	změna
11	0	2400	—	2.5385	0.0011	-99.96 %
12	0	1281.2	—	2.5484	0.0014	-99.95 %
13	15.9	758	+4667.3 %	3.259	0.0012	-99.96 %
14	18.5	527	+2748.65 %	3.8041	0.0013	-99.97 %
15	3.2	655.4	+20381.25 %	2.9401	0.0017	-99.94 %

Tabulka 8.17: Výsledky vylepšené převzaté implementace CGP v porovnání s originálem, při vylepšení zaměřeném na snížení výpočetního času.

Celkově tato série testů předvedla schopnost vytvořeného optimalizátoru vylepšit i převzatou implementaci CGP, obohacenou pouze o značky a formátovaný výstup. Dále se ukázalo, že pokud implementace již v základní podobě poskytuje velmi přesné výsledky, je při vylepšování implementace na ně třeba klást důraz, aby zůstaly zachovány. Jinak může dojít i k celkové ztrátě funkčnosti programu. Nejlepší výsledky pak vykázala implementace, která kladla vylepšení přesnosti výpočtu desetinásobný důraz než na výpočetní čas. Úpravou konstantních funkcí a změnou parametrů CGP v průměru dosáhla snížení odchylky výpočtu o 26.6 % a zrychlení výpočtu o 45.59 %.

8.11 Test náročnosti aproximace konstant

Testy v předchozí sekci ukázaly zajímavé změny v konstantních funkcích vylepšené implementace. V osmé sadě testů proto vyzkoušíme schopnost originální a vylepšené převzaté implementace CGP aproximovat další konstanty. Cílem tohoto testu je ukázat, že vylepšená implementace je schopna aproximovat různá zajímavá čísla přesněji a s využitím menšího počtu výpočetních bloků, při regresi stejných funkcí si vystačí s menší výpočetní mřížkou a bude tedy schopna poskytnout výsledky po uplynutí kratšího výpočetního času.

Testy budou provedeny na posloupnostech se vstupními hodnotami od jedné do sta, a konstantním výstupem. V obou implementacích bude upraven parametr počtu sloupců. Každý test bude proveden pětkrát. Protože hledáme limity jednotlivých implementací, bude za výsledek brán výsledek nejlepšího z běhů, nikoliv jejich průměr. Vylepšenou implementací, kterou v tomto testu použijeme, bude implementace zaměřená na zpřesnění výsledků, která v předchozím testu jako jediná dokázala v průměru zlepšit oba parametry, a ze všech vylepšených implementací CGP se tedy ukázala jako nejlepší. Výsledky jednotlivých běhů jsou součástí přílohy E.

verze implementace	počet sloupců	odchylka od hledané konstanty				
		0.001	$\sqrt{2}$	e	π	1000
originální	2	0.1	41.421	71.828	114.159	79800
	3	0.1	8.579	28.172	14.159	42304
	4	0.002	0.193	3.354	14.159	41420
	5	0.001	0.193	0.961	0.047	41132
	6	0.001	0.193	0.23	0.047	40650
vylepšená	2	0.1	23.822	70.757	28.426	41796.547
	3	0.1	3.124	9.365	0.621	34741.323
	4	0.011	0.193	1.595	0.621	2111.984
	5	0.01	0.193	0.538	0	442.594
	6	0.004	0.193	0.001	0	408.932

Tabulka 8.18: Výsledky testů odchylky vylepšené převzaté implementace CGP při aproximaci vybraných konstant s omezeným počtem sloupců výpočetní mřížky.

Výsledky testu uvedené v tabulce 8.18 ukázaly, že vylepšená implementace je díky vylepšeným konstantním funkcím v naprosté většině případů schopna (se stejným počtem výpočetních bloků) lepší nebo alespoň stejně dobré aproximace konstant než originální implementace. To snižuje počet výpočtů potřebný k získání dostatečně dobrého výsledku, vede k lepšímu využití výpočetních bloků CGP, a pokud je potřeba, umožňuje redukci jeho velikosti a snížení výpočetního času.

8.12 Test optimalizátoru na implementaci CGP pro návrh kombinačních obvodů

Devátá sada testů ověří schopnost optimalizátoru vylepšit implementaci CGP, určenou pro řešení jiného typu úlohy něž symbolická regrese, kterou prováděly obě předchozí vylepšované implementace. Implementace vylepšovaná v této sérii testů bude řešit návrh kombinačních obvodů. Autorem implementace bude stejně jako u předchozí převzaté implementace Zdeněk Vašíček [25] v jehož implementaci CGP byly pro potřeby testů provedeny následující změny:

- Do implementace bylo dodáno pět značek, umístěných kolem parametrů CGP. Konkrétně u počtu generací, velikosti populace, počtu mutací, počtu sloupců, a parametru levels-back. Funkce této implementace nebyly označeny, protože na rozdíl od implementace pro symbolickou regresi implementace pro návrh obvodů neobsahovala žádné konstantní funkce.
- Návratová hodnota úspěšného běhu programu byla změněna z nuly na rozdíl mezi nejlepší možnou a dosaženou hodnotu fitness. Protože při návrhu kombinačních obvodů je maximální možná hodnota fitness obecně známa, originální implementace CGP byla navržena tak, aby hodnotu fitness zvyšovala, a bylo tedy třeba ji takto obrátit.
- Počet generací CGP byl z časových důvodů snížen z původní hodnoty 5 000 000 na 100 000.

Protože bylo do implementace umístěno méně značek něž v předchozích případech, byl také snížen počet mutací, které optimalizátor provádí ze tří na dvě. Ostatní parametry optimalizátoru byly ponechány dle předchozího nastavení.

V první fázi testů byla originální implementace CGP otestována na třech úlohách. Stejně jako v dřívějších testech je cílem tohoto testu získat hodnoty pro pozdější nastavení optimalizátoru, a hodnoty pro srovnání s jím vytvořenými vylepšenými implementacemi. Očekávaným výsledkem testu je, že originální implementace nebude schopna všechny úlohy zcela správně vyřešit, a bude proto možné optimalizátorem vylepšit nejen její výpočetní čas, ale i přesnost výsledku. Jako vstupní úlohy budou použity vzorové příklady, které byly převzaty spolu s implementací. Pro zvýšení vypovídací hodnoty testů bude každá úloha řešena desetkrát. Výsledky jednotlivých běhů jsou součástí přílohy F.

úloha	výpočetní odchylka	doba běhu [s]
adder3_3	3.2	2.8216
adder4_4	42	11.1542
adder5_5	489.2	36.2178
parity9	0	16.098
multiplier4x4	214.6	8.8563
multiplier5x5	2082.6	36.572

Tabulka 8.19: Výsledky testů originální převzaté implementace CGP pro návrh kombinačních obvodů na vzorových úlohách.

Výsledky testu jsou uvedeny v tabulce 8.19. Ukázalo se, že jednotlivé vstupní úlohy jsou různě časově náročné a originální implementace je schopna je řešit s různou přesností. V druhé fázi testu nyní tyto hodnoty použijeme pro nastavení optimalizátoru. Jako v dřívějších testech se pokusíme implementaci vylepšit na tři způsoby. S desetinásobným zaměřením na přesnost výpočtu, s vyváženým zaměřením na přesnost výpočtu a výpočetní čas, a s desetinásobným zaměřením na výpočetní čas. Vstupními úlohami použitými při vylepšování budou první tři úlohy. Další tři úlohy budou ponechány jako testovací pro pozdější ověření vlastností vylepšených implementací CGP. Kvůli způsobu, jakým evoluce probíhá, nejsou neagregované výsledky známy a nejsou tedy součástí přílohy.

zaměření vylepšení	výpočetní odchylka		výpočetní čas [s]	
	hodnota	změna	hodnota	změna
originální implementace	534	0 %	61.526	0 %
přesnost výpočtu	27	-94.94 %	132.753	+115.77 %
přesnost i rychlost	175	-67.23 %	20.687	-66.38 %
rychlost výpočtu	1155	+116.29 %	1.075	-98.25 %

Tabulka 8.20: Výsledky vylepšených implementací CGP pro návrh kombinačních obvodů v porovnání s implementací originální.

Výsledky vylepšení implementace CGP uvedené v tabulce 8.20 ukázaly, že při zaměření na přesnost výpočtu, či výpočetní čas je optimalizátor schopen tento parametr výrazně (více jak desetinásobně) vylepšit za cenu dvojnásobného navýšení druhého z parametrů. V případě zaměření optimalizátoru na vyvážené zlepšení přesnosti výpočtu a výpočetního času došlo k více jak dvojnásobnému zlepšení obou měřených parametrů. Optimalizátor se tak chová velmi podobně jako v dřívějších testech na jiných implementacích CGP a svoji funkci plní dle očekávání.

Náhled do tabulky vektorů hodnot originální a vylepšených implementací, která je uvedena v příloze F.2, ukazuje, že u verze vylepšené se zaměřením na přesnost výpočtu nedošlo k očekávanému zvýšení počtu generací, ale především k nárůstu velikosti populace a velikosti výpočetní mřížky. U verze vylepšené s vyváženým zaměřením přesnost výpočtu i výpočetní čas pak došlo ke snížení velikosti výpočetní mřížky a populace, a naopak k nárůstu počtu generací a velmi výraznému nárůstu počtu mutací.

Oproti vylepšení zaměřenému na přesnost výpočtu je tedy prostor možných řešení touto implementací prohledáván chaotičtěji. Z tohoto odvozujeme, že implementace má vetší šanci objevit nějaké lepší řešení, ale bude mít obtíže již nalezené dobré řešení dále vylepšit, což odpovídá požadavkům na současné vylepšení přesnosti výpočtu i výpočetního času, se kterými byla tato implementace stvořena. U verze vylepšené se zaměřením na zkrácení výpočetního času došlo k obdobným změnám jako v ostatních případech zaměřených stejným způsobem, tedy snížení počtu generací, velikosti populace i velikosti výpočetní mřížky.

Třetí fáze testů nyní ověří schopnost vylepšených implementací CGP navrhovat i další kombinační obvody. Cílem testu je ověřit schopnost vylepšených implementací zachovat si svoje vylepšené parametry i na dalších úlohách. Očekávaným výsledkem je, stejně jako když vylepšení probíhalo na jiných implementacích CGP, že zlepšení přesnosti výpočtu nebude tak výrazné, jako u úloh použitých při vytvoření vylepšených implementací, ale změna v délce výpočetního času zůstane zachována s minimálními změnami.

Použitými úlohami budou zbývající tři vzorové obvody, které při vytvoření vylepšených implementací nebyly použity. Pro zvýšení statistické významnosti testů bude každý z nich proveden desetkrát. Konkrétní výsledky jednotlivých běhů jsou součástí přílohy F.

zaměření vylepšení	vstupní posloupnost	výpočetní odchylka		výpočetní čas [s]	
		hodnota	změna	hodnota	změna
výpočetní odchylka	parity9	0	—	37.588	+133.49 %
	multiplier4x4	205.3	-4.33 %	18.3109	+106.76 %
	multiplier5x5	1871.9	-10.12 %	69.8523	+91 %
výpočetní odchylka i čas	parity9	0	—	8.5011	-47.19 %
	multiplier4x4	271.2	+26.37 %	4.7549	-46.31 %
	multiplier5x5	2342.2	+12.47 %	19.2577	-47.34 %
výpočetní čas	parity9	153.6	—	0.2913	-98.19 %
	multiplier4x4	452.8	+111 %	0.181	-97.96 %
	multiplier5x5	2775.6	+33.28 %	0.6	-98.36 %

Tabulka 8.21: Výsledky vylepšených implementací CGP pro návrh kombinačních obvodů při testech na dalších úlohách v porovnání s výsledky originální implementace.

Výsledky testu jsou uvedeny v tabulce 8.21. Implementace vylepšená se zaměřením na přesnost výpočtu ztratila při použití na dalších vstupních úlohách svoji schopnost výrazně zlepšit přesnost výpočtu. Dvojnásobné navýšení výpočetního času naopak zůstalo zachováno. Implementace, která se zaměřovala na vyvážené vylepšení přesnosti výpočtu i výpočetního času, dosáhla mírně horších výsledků, než originální implementace, ale zachovala si znatelnou míru zrychlení. Implementace, která byla vylepšena se zaměřením na výpočetní čas, si v testech zachovala jak výrazné zhoršení přesnosti výpočtu, tak i jeho zrychlení. Celkově tedy optimalizátor při vylepšení implementace CGP pro návrh kombinačních obvodů, dosáhl horších výsledků, než když jím byly vylepšovány implementace CGP pro symbolickou regresi.

Důvodů může být několik. Prvním je, že trénovací úlohy se hrubě lišily, co se týče vzájemné časové náročnosti i velikosti průměrné výpočetní odchylky, které se u nich originální implementace dopouští. Například úloha *adder3_3* představující kombinační obvod tříbitové sčítačky měla stokrát nižší výpočetní odchylku a desetkrát nižší výpočetní čas než úloha *adder5_5* představující sčítačku pětibitovou. Při optimalizaci tedy nejspíše došlo k přílišnému zaměření na vylepšení implementace CGP pro potřeby jedné konkrétní úlohy (*adder5_5*) což, jak ukázaly dřívější testy (sekce 8.6), vede k optimalizaci, která není schopna si vylepšení přesnosti výpočtu uchovat při použití na dalších úlohách. Druhým možným důvodem je, že implementace CGP pro návrh kombinačních obvodů neobsahovala žádné zajímavé konstanty, jejichž hodnoty by šlo upravit, a zvýšit tak efektivitu výpočtu, jako toho bylo dosaženo u implementace CGP provádějící symbolickou regresi (sekce 8.11).

Přestože vylepšené implementace CGP nedosáhly tak dobrých výsledků při návrhu kombinačních obvodů jako v předchozích případech, testy ukázaly, že optimalizátor je schopen vylepšit implementaci CGP provádějící i jiné úlohy než symbolickou regresi, a podařilo se mu výrazně zlepšit jeden z parametrů na úkor menšího zhoršení parametru druhého. Konkrétně implementace vytvořená při zaměření na vyvážené snížení výpočetní odchylky i času, dokázala snížit čas výpočtu v průměru o 47.15 % za cenu průměrného zhoršení přesnosti výpočtu o 29.77 %. Tyto hodnoty by pak mělo být možné dále zlepšit použitím větší a lépe sestavené množiny trénovacích úloh.

Kapitola 9

Závěr

Cílem této práce bylo seznámit se s problematikou evolučního vylepšování a kartézského genetického programování (CGP) a navrhnout a implementovat metodu umožňující genetické vylepšování software, který základní variantu CGP implementuje.

První část této práce čtenáře seznámila s evolučními algoritmy, genetickým programováním, jeho kartézskou variantou, genetickým vylepšováním software a symbolickou regresí jako jedním z možných problémů, který lze těmito metodami řešit. V druhé části byla na základě těchto poznatků navržena metoda genetického vylepšení implementace CGP skrze úpravu vybraných numerických konstant nacházejících se ve zdrojovém kódu vylepšované implementace.

Za použití jazyka C++ byla vytvořena originální implementace CGP řešící problém symbolické regrese. Dále byly vytvořeny dva skripty v jazyce Python. Jeden pro extrakci hodnot, které mají být vylepšeny, ze zdrojového kódu implementace. Druhý pro modifikaci zdrojového kódu a jeho automatizovaný překlad překladačem G++. Hlavní část projektu pak tvoří program implementovaný v jazyce C++, který za využití zmíněných skriptů implementaci CGP geneticky vylepší z hlediska zvýšení přesnosti poskytovaných výsledků a snížení potřebného výpočetního času.

Funkčnost programu byla otestována na dvou implementacích CGP (jedné vlastní a druhé převzaté) řešících symbolickou regresi patnácti různých vstupních posloupností. V rámci testů byla zjištěna přesnost výsledků a výpočetní čas, který nevylepšené implementace potřebují na jejich získání. Obě implementace poté byly vylepšeny se zaměřením na zlepšení jednoho, druhého nebo obou těchto parametrů.

Vylepšovanými hodnotami byly evoluční parametry CGP a hodnoty konstantních vstupů (u vlastní implementace CGP), respektive konstantních funkcí (u převzaté implementace CGP). V obou případech došlo v závislosti na způsobu, kterým se vylepšení implementace mělo zaměřit, ke zlepšení jednoho nebo obou sledovaných parametrů.

U vlastní implementace CGP, která poskytovala méně přesné výsledky, došlo k přenastavení parametrů takovým způsobem, aby byla umožněna přesnější aproximace složitějších vstupních posloupností při zachování přesnosti na jednodušších vstupních posloupnostech a výrazném snížení potřebného výpočetního času.

U převzaté implementace CGP, která již v originální podobě byla schopna velmi rychle poskytovat velmi přesné výsledky, byla pozorovaná úroveň zlepšení menší. Kromě upravení evolučních parametrů jí bylo dosaženo především změnou hodnot konstantních funkcí.

V originální implementaci měly konstantní funkce hodnoty, které bylo možné získat pomocí malého počtu základních matematických operací s primárním vstupem nebo ostatními konstantními funkcemi. Geneticky vylepšená implementace naopak obsahovala funkce, jejichž hodnoty měly výrazně větší vzájemný rozestup, čímž umožnily lépe pokrýt větší spektrum aproximovaných hodnot za použití nižšího počtu výpočetních bloků mřížky CGP.

To mimo jiné umožnilo pomocí menšího počtu operací provádět lepší aproximaci řady konstant, čímž došlo ke zvýšení efektivity symbolické regrese. Například pro aproximaci hodnoty π s přesností na dva desetinné řády stačilo vylepšené implementaci použít tři výpočetní bloky. Nevylepšená implementaci pro provedení aproximace o podobné přesnosti vyžadovala výpočetních bloků nejméně pět.

Funkčnost vytvořeného programu poté byla otestována na třetí (převzaté) implementaci CGP řešící problém návrhu kombinačních obvodů. Míra zlepšení nebyla tak velká jako u implementací řešících problém symbolické regrese, testy ale ukázaly, že vytvořený program je schopen implementaci celkově vylepšit, případně ji upravit tak, aby došlo k výraznému zvýšení přesnosti výpočtu nebo snížení výpočetního času v závislosti na požadavcích uživatele.

Kromě již zmíněných výsledků by vytvořený program pro genetické vylepšování bylo možné využít pro vylepšení i dalších implementací CGP řešících různé úlohy a obecně i dalších programů provádějících aproximační výpočty. Jednoduchou úpravou skriptů pro automatizovanou extrakci hodnot, modifikaci a překlad zdrojového kódu by šlo vylepšovat i programy psané v dalších jazycích, než které byly v rámci této práce použity.

Literatura

[1] Bortel, M.: *Evoluční algoritmy*. Diplomová práce, Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2012.

[2] Clegg, J.; Walker, J. A.; Miller, J. F.: A new crossover technique for cartesian genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ACM, 2007, s. 1580–1587.

[3] Duffy, J.; Engle-Warnick, J.: Using symbolic regression to infer strategies from experimental data. In *Evolutionary computation in Economics and Finance*, Springer, 2002, s. 61–82.

[4] Koza, J. R.: *Genetic programming: on the programming of computers by means of natural selection*, ročník 1. MIT press, 1992.

[5] Koza, J. R.: 36 Human-Competitive Results Produced by Genetic Programming. 2003, cit. 2003-12-30.
URL http://www.genetic-programming.com/humancompetitive.html

[6] Koza, J. R.: Introduction to Genetic Programming Tutorial GECCO-2004. 2004.

[7] Landsborough, J.; Harding, S.; Fugate, S.: Removing the kitchen sink from software. In *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, ACM, 2015, s. 833–838.

[8] Langdon, W. B.: Genetic improvement of software for multiple objectives. In *Search-Based Software Engineering*, Springer, 2015, s. 12–28.

[9] Langdon, W. B.; Harman, M.: Evolving a CUDA kernel from an nVidia template. In *2010 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2010, s. 1–8.

[10] 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.

[11] Lattner, C.: The LLVM Compiler Infrastructure Project. 2016, cit. 2016-01-01.
URL http://llvm.org/

[12] Miller, J.: Cartesian Genetic Programming. In *Cartesian Genetic Programming*, editace J. F. Miller, Natural Computing Series, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17309-7, s. 17–34.

[13] Miller, J. F.: PPSN 2014 Tutorial: Cartesian Genetic Programming. 2014, cit. 2015-12-01.
URL http://ppsn2014.ijs.si/files/slides/ppsn2014-tutorial3-miller.pdf

[14] Miller, J. F.; Thomson, P.: Cartesian genetic programming. In *Genetic Programming*, Springer, 2000, s. 121–132.

[15] Munakata, T.: Genetic Algorithms and Evolutionary Computing. In *Fundamentals of the New Artificial Intelligence*, editace T. Munakata, Texts in Computer Science, Springer London, 2008, ISBN 978-1-84628-838-8, s. 85–120.

[16] Poli, R.; Langdon, W. B.; McPhee, N. F.; aj.: Genetic programming: An introductory tutorial and a survey of techniques and applications. *University of Essex, UK, Tech. Rep. CES-475*, 2007.

[17] Poli, R.; Langdon, W. B.; McPhee, N. F.; aj.: *A field guide to genetic programming*. Lulu. com, 2008.

[18] Reeder, J.; Steffen, P.; Giegerich, R.: pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. *Nucleic acids research*, ročník 35, č. suppl 2, 2007: s. W320–W324.

[19] Schmidt, M.; Lipson, H.: Symbolic regression of implicit equations. In *Genetic Programming Theory and Practice VII*, Springer, 2010, s. 73–85.

[20] Sekanina, L.: Kartézské genetické programování. 2015, cit. 2016-03-03.
URL https://www.fit.vutbr.cz/study/courses/BIN/private/prednasky/bin2015_p04.pdf

[21] Sekanina, L.; Vasicek, Z.: Evolutionary computing in approximate circuit design and optimization. In *1st Workshop on Approximate Computing (WAPCO 2015)*, 2015, s. 1–6.

[22] Sekanina, L.; Vašíček, Z.; Růžička, R.; aj.: *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Edice Gerstner, Academia, 2009, ISBN 978-80-200-1729-1, 328 s., cit. 2015-12-01.
URL
http://www.fit.vutbr.cz/research/view_pub.php.cz.iso-8859-2?id=9123

[23] Szöllösi, T.: *Evoluční algoritmy*. Diplomová práce, Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2012.

[24] Tomaštík, M.: *Genetické programování-Java implementace*. Diplomová práce, Vysoké učení technické v Brně. Fakulta strojního inženýrství, 2013.

[25] Vašíček, Z.: Efficient Phenotype Evaluation in Cartesian Genetic Programming. 2016, cit. 2016-05-15.
URL http://www.fit.vutbr.cz/~vasicek/cgp/

[26] Weimer, W.; Forrest, S.; Le Goues, C.; aj.: Automatic program repair with evolutionary computation. *Communications of the ACM*, ročník 53, č. 5, 2010: s. 109–116.

[27] White, D. R.; Singer, J.: Rethinking Genetic Improvement Programming. In *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, ACM, 2015, s. 845–846.

[28] Zhang, C.: Genetic Programming for Symbolic Regression. Cit. 2015-12-01. URL http://web.eecs.utk.edu/~czhang24/projects/cs528_Project2_Zhang.pdf

Přílohy

Seznam příloh

Příloha A

Vstupní posloupnosti

kód	generující vztah	hodnoty	nejmenší	inkrement	největší
1	$y = 2x + 1$	41	-10	0,5	10
2	$y = x^2 - 4x$	41	-10	0,5	10
3	$y = 3x + 1/x$	21	0,1	0,1	2,1
4	$y = 0,4x^2 - 7,5x + 48$	31	0	1	30
5	$y = 17\sin(x)$	17	0	$\pi/8$	2π
6	$y = 9\sin(1 + (x/3)) + 11\cos(1 - 3x)$	25	0	$\pi/4$	6π
7	$y = -9x + 50$	13	9	-0,5	3
8	$y = 4x^2 + 2x - 1$	13	-3	0,5	3
9	$y = 2x\cos(x)$	13	0	$\pi/3$	4π
10	$y = 20\sin(2\pi\cos(x))$	13	0	$\pi/12$	π
11	$y = 9x^2$	25	-2	0.2	2.8
12	$y = x^3 + x^2 + x + 1$	25	-2.7	0.3	4.5
13	$y = x^2 + (1 - x)^2$	25	-5.5	0.5	6.5
14	$y = 15cos(x)/x$	25	$\pi/50$	$\pi/50$	$\pi/2$
15	$y = 40sin(sin(sin(x)))$	25	0	$\pi/16$	$2\pi/3$

Tabulka A.1: Tabulka vztahů a parametrů pro generování posloupností použitých při testech programu.

Příloha B

Vektory hodnot vylepšených implementací CGP

Implementace jsou pojmenovány dle cíle, na který se se vylepšování zaměřovalo (F – maximální snížení odchylky řešení, C – kombinované snížení odchylky řešení a výpočetního času, T – maximální snížení výpočetního času), a kódu posloupnosti, na jejímž základě vylepšování probíhalo.

	parametry					
implementace	populace	mutace	sloupce	řádky	l-back	generace
originální	5	3	4	4	4	10000
F 1	4	21	3	14	12	13447
F 2	4	37	6	9	13	13545
F 3	2	8	2	4	3	6718
F 4	7	21	9	2	5	1732
F 5	3	15	10	4	6	7718
F 6	2	10	4	5	11	7731
F 7-10	3	13	8	6	8	12478
C 1	6	22	8	3	18	4459
C 2	6	11	7	4	9	9894
C 3	2	2	4	3	30	3566
C 4	3	2	5	4	15	4500
C 5	4	7	8	4	4	12035
C 6	2	31	3	1	55	7450
C 7-10	3	29	7	3	5	4299
T 1	2	1	2	6	60	6124
T 2	2	9	1	8	2	7214
T 3	2	1	4	2	16	19303
T 4	2	22	3	4	2	3420
T 5	2	14	1	3	20	4186
T 6	2	2	2	1	1	3572
T 7-10	3	3	3	1	5	3225

Tabulka B.1: Hodnoty parametrů geneticky vylepšených implementací CGP.

implementace	konstanty			
	A	B	C	D
originální	0,5	1,0	2,0	10,0
F 01	1.303638	3.731822	3.343251	14.980290
F 02	8.186467	2.258251	1.990698	35.043422
F 03	0.502393	1.8611255	9.780161	55.704215
F 04	0.638181	3.347916	5.791022	44.763158
F 05	0.853417	1.090245	1.025118	18.945430
F 06	0.928847	2.830417	2.141751	736.579658
F 07-10	0.924111	2.849794	10.652345	13.532885
C 01	0.936367	1.194073	2.022027	28.778635
C 02	0.820896	1.395391	2.836831	17.197725
C 03	5.283309	2.040713	24.0713	56.544489
C 04	0.180643	2.468707	2.046367	6.314087
C 05	0.762276	3.961695	11.333998	16.843901
C 06	1.041295	26.783812	5.030041	14.078898
C 07-10	4.781122	4.373142	2.849575	9.161243
T 01	0.660231	0.588148	9.575197	14.637154
T 02	0.891875	1.539409	0.746699	38.877048
T 03	4.036630	7.543655	1.460297	31.724781
T 04	1.544967	3.370857	3.642608	10.419943
T 05	0.558796	1.286883	31.625280	41.823251
T 06	1.766207	2.423939	2.058407	20.128517
T 07-10	0.547035	3.226117	2.478399	14.355686

Tabulka B.2: Hodnoty konstantních vstupů geneticky vylepšených implementací CGP.

Příloha C

Výsledky testů vylepšených implementací CGP

Implementace jsou pojmenovány dle cíle, na který se se vylepšování zaměřovalo (F – maximální snížení odchylky řešení, C – kombinované snížení odchylky řešení a výpočetního času, T – maximální snížení výpočetního času), a kódu posloupnosti, na jejímž základě vylepšování probíhalo.

	číslo posloupnosti			
implementace	07	08	09	10
originální	1985	2002	1433	1794
F 01	1529	195	1432	1728
F 02	243	59	390	1513
F 03	2739	2104	1525	1883
F 04	1426	448	627	1721
F 05	521	182	413	1611
F 06	2266	2015	1433	1792
C 01	903	141	967	1731
C 02	780	817	654	1533
C 03	2465	2061	1509	1833
C 04	2653	2024	1464	1796
C 05	599	182	1040	1677
C 06	3274	2301	1589	1942
T 01	2739	2165	1541	1914
T 02	2779	2165	1569	1914
T 03	2792	2091	1545	1910
T 04	2176	2043	1506	1872
T 05	3275	2197	1569	1925
T 06	3260	2301	1589	1942

Tabulka C.1: Velikost odchylky jednotlivých implementací CGP na dalších vstupních posloupnostech.

implementace	číslo posloupnosti			
	07	08	09	10
originální	5.459	5.280	5.399	5.005
F 1	12.368	12.162	13.678	12.144
F 2	22.195	14.905	17.810	17.744
F 3	0.409	0.433	0.453	0.410
F 4	1.650	1.625	1.781	1.687
F 5	5.140	4.999	5.294	5.230
F 6	1.249	1.187	1.312	1.248
C 1	4.640	4.624	5.323	4.722
C 2	11.859	12.046	13.720	13.637
C 3	0.312	0.328	0.391	0.399
C 4	1.450	1.484	1.517	1.375
C 5	9.654	9.625	11.115	10.886
C 6	0.578	0.516	0.495	0.515
T 1	0.593	0.574	0.625	0.593
T 2	0.526	0.571	0.573	0.578
T 3	1.359	1.343	1.545	1.328
T 4	0.437	0.546	0.624	0.484
T 5	0.197	0.203	0.203	0.219
T 6	0.094	0.094	0.109	0.093

Tabulka C.2: Doba výpočtu jednotlivých implementací CGP na dalších vstupních posloupnostech.

číslo běhu	přesnost		rychlost	
	originál	modifikace	originál	modifikace
1	3933	2248	8.858	2.554
2	7559	6063	8.622	2.705
3	8331	7175	8.673	2.742
4	7751	2167	8.622	2.442
5	8184	1433	8.565	2.478
6	7751	2239	8.743	2.445
7	7749	4343	8.774	2.464
8	7195	190	8.497	2.438
9	8391	1216	8.423	2.478
10	8184	1408	8.747	2.867
průměr	7502.8	2848.2	8.6524	2.5613

Tabulka C.3: Výsledky testů originální a nejlepší modifikované implementace CGP na posloupnosti 11.

číslo běhu	přesnost		rychlost	
	originál	modifikace	originál	modifikace
1	6107	5904	8.629	2.436
2	9182	3270	8.878	2.474
3	7939	5831	8.618	2.536
4	6476	2708	8.887	2.55
5	9108	3503	8.931	2.48
6	7939	7686	8.861	2.53
7	6450	6801	9.021	2.637
8	9055	3014	8.906	2.608
9	7824	3224	8.975	2.772
10	8338	3276	8.738	2.853
průměr	7841.8	4521.7	8.8444	2.5876

Tabulka C.4: Výsledky testů originální a nejlepší modifikované implementace CGP na posloupnosti 12.

číslo běhu	přesnost		rychlost	
	originál	modifikace	originál	modifikace
1	13264	2459	7.911	2.484
2	13376	8363	7.742	2.453
3	5216	3645	7.985	2.499
4	13317	2525	8.001	2.593
5	5240	12955	8.186	2.515
6	8335	1317	8.353	2.484
7	8093	2834	7.828	2.507
8	8093	8093	8.68	2.474
9	13203	4538	7.752	2.501
10	13314	5326	8.415	2.734
průměr	10145.1	5205.5	8.0853	2.5244

Tabulka C.5: Výsledky testů originální a nejlepší modifikované implementace CGP na posloupnosti 13.

číslo běhu	přesnost		rychlost	
	originál	modifikace	originál	modifikace
1	5660	2653	8.891	2.658
2	3501	879	9.412	2.84
3	7889	921	9.295	2.8
4	3565	913	9.044	2.828
5	12865	881	8.947	2.855
6	8555	980	9.589	2.836
7	3565	1007	9.443	2.879
8	3565	1046	9.212	2.95
9	3565	811	9.449	2.939
10	7039	877	8.896	2.838
průměr	5976.9	1096.8	9.2178	2.8423

Tabulka C.6: Výsledky testů originální a nejlepší modifikované implementace CGP na posloupnosti 14.

číslo běhu	přesnost		rychlost	
	originál	modifikace	originál	modifikace
1	8738	2047	8.896	2.973
2	5053	1300	9.42	3.179
3	5054	608	10.063	3.31
4	6088	872	10.02	3.129
5	5552	327	9.784	3.136
6	8738	1165	9.356	3.125
7	4729	996	9.718	3.207
8	8252	4448	9.597	3.651
9	7431	674	9.333	3.294
10	7798	327	8.992	3.239
průměr	6743.3	1276.4	9.5179	3.2243

Tabulka C.7: Výsledky testů originální a nejlepší modifikované implementace CGP na posloupnosti 15.

Příloha D

Výsledky testů převzaté implementace CGP

V této příloze jsou vylepšení implementace CGP pojmenovány dle cíle, na který se se vylepšování zaměřovalo. F – maximální snížení odchylky řešení, C – kombinované snížení odchylky řešení a výpočetního času, T – maximální snížení výpočetního času.

číslo	posloupnost 07		posloupnost 08		posloupnost 09		posloupnost 10	
běhu	odchylka	čas	odchylka	čas	odchylka	čas	odchylka	čas
1	8	1.568	0	1.410	1	1.519	59	1.464
2	13	1.364	0	1.449	0	2.188	87	1.520
3	63	1.834	0	1.367	1	2.045	80	1.662
4	16	1.952	0	1.275	7	1.849	42	2.236
5	15	1.375	0	1.367	7	1.755	72	1.937
6	20	2.359	0	1.394	0	1.711	46	2.059
7	18	1.649	0	1.364	0	1.885	39	1.801
8	23	1.720	0	1.400	3	1.952	58	2.172
9	16	1.897	0	1.401	25	2.335	77	2.003
10	20	1.600	0	1.440	0	1.827	72	1.832
průměr	21.2	1.7318	0	1.3867	4.4	1.9066	63.2	1.8686

Tabulka D.1: Výsledky testů převzaté implementace CGP na posloupnostech 7-10.

implementace	generace	populace	mutace	sloupce	konst. A	konst. B	konst. C
originální	50000	5	5	15	0.25	0.5	1.0
verze F	51379	2	12	33	1.8509	0.588	12.7729
verze C	25943	6	6	7	6.3716	0.4809	13.2883
verte T	3595	1	3	17	111.3667	4.8031	0.0104

Tabulka D.2: Tabulka vektorů hodnot převzaté implementace CGP a jejích vylepšení.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	0	0	0	2400
2	0	0	0	2400
3	0	6	0	2400
4	0	0	0	2400
5	0	0	0	2400
6	0	0	0	2400
7	0	1	3	2400
8	0	6	3	2400
9	0	1	0	2400
10	0	1	0	2400
průměr	0	1.5	0.6	2400

Tabulka D.3: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Odchylky od přesného výsledku posloupnosti 11.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	2.457	1.446	0.639	0.001
2	2.194	1.498	0.621	0.001
3	2.371	1.576	0.721	0.001
4	2.467	1.51	0.727	0.001
5	2.682	1.479	0.745	0.001
6	2.712	1.447	0.73	0.002
7	2.711	1.514	0.742	0.002
8	3.277	1.456	0.681	0.001
9	2.051	1.414	0.735	0.001
10	2.463	1.469	0.854	0
průměr	2.5385	1.4809	0.7195	0.0011

Tabulka D.4: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Výpočetní čas na posloupnosti 11.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	0	0	0	2387
2	0	0	0	2387
3	0	0	0	2387
4	0	0	1	2387
5	0	0	0	544
6	0	0	0	544
7	0	0	0	544
8	0	0	0	544
9	0	0	0	544
10	0	0	8	544
průměr	0	0	0.9	1281.2

Tabulka D.5: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Odchylky od přesného výsledku posloupnosti 12.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	2.348	1.436	0.782	0.001
2	2.6	1.499	0.818	0.002
3	3.129	1.515	0.78	0.002
4	2.519	1.932	0.873	0.001
5	2.317	1.473	0.713	0.001
6	2.389	1.501	0.741	0.001
7	2.497	1.557	0.778	0.001
8	2.571	1.529	0.702	0.001
9	2.506	1.462	0.710	0.002
10	2.608	1.469	0.917	0.002
průměr	2.5484	1.5373	0.7814	0.0014

Tabulka D.6: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Výpočetní čas na posloupnosti 12.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	16	7	73	758
2	8	15	13	758
3	2	12	74	758
4	72	12	63	758
5	14	11	51	758
6	15	3	51	758
7	1	6	72	758
8	6	4	58	758
9	22	2	74	758
10	3	9	36	758
průměr	15.9	8.1	56.5	758

Tabulka D.7: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Odchylky od přesného výsledku posloupnosti 13.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	2.701	1.46	0.78	0.001
2	2.795	1.466	0.734	0.001
3	2.967	1.503	0.977	0
4	3.855	1.626	0.933	0.002
5	3.945	1.536	0.816	0.002
6	2.64	1.589	0.914	0.002
7	2.826	1.458	0.962	0.001
8	3.598	1.484	0.769	0
9	3.456	1.536	0.64	0.002
10	3.807	1.543	0.742	0.001
průměr	3.259	1.5201	0.8267	0.0012

Tabulka D.8: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Výpočetní čas na posloupnosti 13.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	58	1	37	527
2	37	10	16	527
3	7	7	44	527
4	4	34	37	527
5	25	17	22	527
6	12	37	37	527
7	37	36	5	527
8	2	1	37	527
9	0	3	37	527
10	3	0	2	527
průměr	18.5	14.6	27.4	527

Tabulka D.9: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Odchylky od přesného výsledku posloupnosti 14.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	3.304	1.814	1.005	0.001
2	3.956	1.8	1.259	0.001
3	3.997	1.896	0.929	0.001
4	4.289	1.716	1.253	0.002
5	3.683	1.821	1.212	0.002
6	4.349	1.753	0.94	0.001
7	3.019	1.899	1.532	0.001
8	3.695	2.085	0.975	0.002
9	4.211	1.967	0.936	0.002
10	3.538	2.148	1.383	0
průměr	3.8041	1.8899	1.1424	0.0013

Tabulka D.10: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Výpočetní čas na posloupnosti 14.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	3	5	3	653
2	3	3	3	653
3	1	2	4	653
4	3	3	4	653
5	3	3	3	653
6	3	3	3	653
7	7	3	4	653
8	3	5	3	653
9	3	3	3	665
10	3	4	4	665
průměr	3.2	3.4	3.4	655.4

Tabulka D.11: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Odchylky od přesného výsledku posloupnosti 15.

číslo běhu	originální implementace	zaměření vylepšení		
		verze F	verze C	verze T
1	3.035	1.993	0.664	0.001
2	2.892	1.671	0.698	0.001
3	2.805	1.693	0.733	0.002
4	2.95	1.853	0.714	0.002
5	2.967	1.652	0.784	0.001
6	2.829	1.801	0.717	0.001
7	3.224	1.94	0.743	0.001
8	2.872	1.591	0.723	0.003
9	2.898	1.728	0.741	0.002
10	2.929	1.919	0.687	0.003
průměr	2.9401	1.7841	0.7204	0.0017

Tabulka D.12: Výsledky testů převzaté implementace a jejích vytvořených modifikací. Výpočetní čas na posloupnosti 15.

Příloha E

Výsledky testů aproximace vybraných konstant

počet sloupců	číslo běhu	odchylka od hledané konstanty				
		0.001	$\sqrt{2}$	e	π	1000
2	1	0.1	41.421	71.828	114.159	79800
	2	0.1	41.421	71.828	114.159	79800
	3	0.1	41.421	71.828	114.159	79800
	4	0.1	41.421	71.828	114.159	79800
	5	0.1	41.421	71.828	114.159	79800
3	1	0.1	8.579	28.172	14.159	42304
	2	0.1	8.579	28.172	14.159	42304
	3	0.1	8.579	28.172	14.159	42304
	4	0.1	8.579	28.172	14.159	42304
	5	0.1	8.579	28.172	14.159	42304
4	1	0.002	0.193	3.354	14.159	42208
	2	0.002	0.193	3.354	14.159	41420
	3	0.002	0.193	3.354	14.159	41420
	4	0.002	6.521	3.354	14.159	41420
	5	0.002	0.193	3.354	14.159	41420
5	1	0.001	0.193	0.961	0.047	41132
	2	0.001	0.193	0.961	1.285	41132
	3	0.001	0.349	1.021	1.694	41132
	4	0.001	0.193	0.961	1.24	41132
	5	0.002	0.492	1.021	0.047	41132
6	1	0.001	0.193	0.961	1.24	41132
	2	0.001	0.193	0.961	0.74	40650
	3	0.095	0.193	0.23	1.392	40964
	4	0.08	0.193	0.961	0.047	40650
	5	0.001	0.193	0.961	0.047	40985

Tabulka E.1: Výsledky testů převzaté implementace CGP při aproximaci vybraných konstant s omezeným počtem sloupců výpočetní mřížky.

počet sloupců	číslo běhu	odchylka od hledané konstanty				
		0.001	$\sqrt{2}$	e	π	1000
2	1	0.1	23.822	70.757	28.426	41796.547
	2	0.1	23.822	70.757	28.426	41796.547
	3	0.1	23.822	70.757	28.426	41796.547
	4	0.1	23.822	70.757	28.426	41796.547
	5	0.1	23.822	70.757	28.426	41796.547
3	1	0.1	8.579	9.365	0.621	34741.323
	2	0.1	3.124	9.365	0.621	34741.323
	3	0.1	3.124	9.365	0.621	34741.323
	4	0.1	15.131	9.365	0.621	34741.323
	5	0.1	15.131	9.365	0.621	34741.323
4	1	0.026	1.986	1.647	0.621	29691.323
	2	0.011	0.221	1.647	0.621	2111.984
	3	0.1	0.221	1.595	0.621	20437.018
	4	0.1	0.193	1.76	0.621	29691.323
	5	0.1	0.193	1.76	0.621	2111.984
5	1	0.025	0.221	1.647	0.621	2111.984
	2	0.097	0.221	1.595	0	442.594
	3	0.097	0.221	0.538	0.552	442.594
	4	0.01	0.193	0.538	0.621	2111.983
	5	0.097	0.221	1.647	0.621	10984.334
6	1	0.004	0.193	0.001	0.621	442.594
	2	0.09	0.193	0.54	0.621	442.594
	3	0.009	0.221	1.647	0	408.932
	4	0.004	0.221	1.647	0.02	447.102
	5	0.079	0.221	0.945	0.621	442.594

Tabulka E.2: Výsledky testů vylepšené převzaté implementace CGP při aproximaci vybraných konstant s omezeným počtem sloupců výpočetní mřížky.

Příloha F

Výsledky testů CGP pro návrh kombinačních obvodů

číslo běhu	adder3_3		adder4_4		adder5_5	
	fitness	čas	fitness	čas	fitness	čas
1	0	3.13	16	11.162	808	49.331
2	8	2.665	44	11.165	536	38.468
3	4	2.621	40	10.739	364	34.343
4	7	2.634	56	9.897	296	34.625
5	0	3.559	88	13.444	760	34.109
6	2	3.172	4	10.973	410	34.297
7	4	2.654	56	10.502	440	34.342
8	1	2.527	56	10.516	160	34.201
9	6	2.566	24	10.515	560	33.98
10	0	2.688	36	12.629	560	34.482
průměr	3.2	2.8216	42	11.1542	489.2	36.2178

Tabulka F.1: Výsledky testů neupravené implementace CGP pro návrh kombinačních obvodů na vzorových posloupnostech

implementace	generace	populace	mutace	sloupce	levels-back
originální	100000	5	1	80	40
verze F	82595	8	4	117	91
verze C	141853	3	7	66	27
verze T	10743	3	5	33	51

Tabulka F.2: Tabulka vektorů hodnot převzaté implementace CGP pro návrh kombinačních obvodů a jejích vylepšení. Verze F byla vylepšena se zaměřením na přesnost výpočtu, verze C byla vylepšena s vyváženým zaměřením na přesnost a rychlost výpočtu, verze T byla vylepšena se zaměřením na rychlost výpočtu.

číslo běhu	parity9		multiplier4x4		multiplier5x5	
	fitness	čas	fitness	čas	fitness	čas
1	0	15.091	175	8.845	2033	41.294
2	0	13.848	238	8.976	2125	37.103
3	0	14.216	262	8.704	2189	36.384
4	0	15.526	278	8.949	2171	36.326
5	0	16.913	174	8.843	2089	36.614
6	0	17.078	234	8.95	1863	35.552
7	0	17.031	222	8.85	1997	35.529
8	0	17.167	204	8.875	1661	34.984
9	0	17.136	182	8.739	2429	36.106
10	0	16.974	177	8.832	2269	35.828
průměr	0	16.098	214.6	8.8563	2082.6	36.572

Tabulka F.3: Výsledky testů neupravené implementace CGP pro návrh kombinačních obvodů na testovacích posloupnostech.

číslo běhu	parity9		multiplier4x4		multiplier5x5	
	fitness	čas	fitness	čas	fitness	čas
1	0	29.031	201	21.707	1946	76.791
2	0	30.371	244	19.636	1911	69.147
3	0	35.423	182	18.32	1915	72.962
4	0	37.268	189	17.023	1771	68.452
5	0	38.976	198	18.633	1859	67.094
6	0	39.421	194	17.35	1901	67.658
7	0	41.446	143	16.524	2031	67.531
8	0	41.402	252	17.212	1933	68.06
9	0	40.194	190	17.39	1763	67.844
10	0	42.348	260	19.314	1689	72.984
průměr	0	37.588	205.3	18.3109	1871.9	69.8523

Tabulka F.4: Výsledky testů implementace CGP pro návrh kombinačních obvodů po vylepšení se zaměřením na zlepšení fitness.

číslo běhu	parity9		multiplier4x4		multiplier5x5	
	fitness	čas	fitness	čas	fitness	čas
1	0	8.271	286	4.894	2171	18.41
2	0	7.974	220	4.706	2455	18.621
3	0	7.808	316	4.815	2025	18.496
4	0	7.655	254	4.709	2389	18.241
5	0	8.065	216	4.657	3131	19.122
6	0	8.637	290	4.816	2155	18.492
7	0	9.402	344	4.814	2139	18.846
8	0	9.16	248	4.849	2211	20.741
9	0	8.958	324	4.69	2291	21.346
10	0	9.081	214	4.599	2455	20.262
průměr	0	8.5011	271.2	4.7549	2342.2	19.2577

Tabulka F.5: Výsledky testů implementace CGP pro návrh kombinačních obvodů po vylepšení se zaměřením na vyvážené zlepšení fitness a výpočetního času

číslo běhu	parity9		multiplier4x4		multiplier5x5	
	fitness	čas	fitness	čas	fitness	čas
1	256	0.265	428	0.218	2841	0.61
2	256	0.304	428	0.187	2841	0.594
3	256	0.331	428	0.187	2919	0.625
4	0	0.294	428	0.203	2693	0.593
5	0	0.269	490	0.188	2693	0.625
6	0	0.265	490	0.187	2693	0.593
7	0	0.265	490	0.203	2777	0.578
8	256	0.296	490	0.141	2777	0.581
9	256	0.296	428	0.140	2761	0.598
10	256	0.328	428	0.156	2761	0.603
průměr	153.6	0.2913	452.8	0.181	2775.6	0.6

Tabulka F.6: Výsledky testů implementace CGP pro návrh kombinačních obvodů po vylepšení se zaměřením na zlepšení výpočetního času.

Příloha G

Obsah CD

Přiložené CD obsahuje veškeré zdrojové kódy které byly v rámci tohoto projektu vytvořeny nebo převzaty. Dále obsahuje vstupní posloupnosti které byly použity v rámci provedených experimentů a návod jak projekt přeložit, spustit a zopakovat provedené experimenty. Umístění a obsah jednotlivých a souborů a složek je následující:

- **original.cpp** – Zdrojový kód originální implementace CGP.
- **evolver.cpp** – Zdrojový kód optimalizátoru.
- **extractor.py** – Skript pro extrakci označených hodnot.
- **replacer.py** – Skript pro úpravu a překlad zdrojového kódu.
- **cgp-SR.cpp** – Zdrojový kód převzaté implementace CGP pro symbolickou regresi.
- **cgp-KO.cpp** – Zdrojový kód převzaté implementace CGP pro návrh kombinačních obvodů.
- **data original** – Složka se vstupními posloupnostmi ve formátu v jakém je očekává originální implementace CGP.
- **data cgp-SR** – Složka se vstupními posloupnostmi ve formátu v jakém je očekává převzatá implementace CGP pro symbolickou regresi.
- **data cgp-KO** – Složka se vstupními úlohami ve formátu v jakém je očekává převzatá implementace CGP pro návrh kombinačních obvodů.
- **README.txt** – Návod k přeložení a použití projektu.