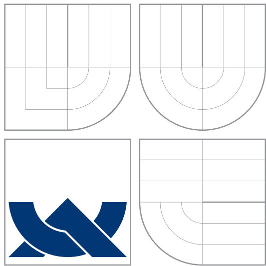
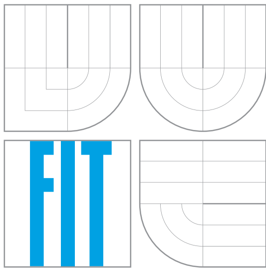


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

SOUBĚŽNÉ UČENÍ V KARTÉZSKÉM GENETICKÉM PROGRAMOVÁNÍ

CO-LEARNING IN CARTESIAN GENETIC PROGRAMMING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAKUB KORGO
AUTHOR

VEDOUCÍ PRÁCE — Ing. MICHAL WIGLASZ
SUPERVISOR

BRNO 2016

Zadání bakalářské práce/18118/2015/xkorgo01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Korgo Jakub**

Obor: Informační technologie

Téma: **Souběžné učení v kartézském genetickém programování**
Co-Learning in Cartesian Genetic Programming

Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s principy evolučních algoritmů, kartézského genetického programování a souběžného učení v evolučních algoritmech.
2. Navrhněte program umožňující řešit vybrané úlohy (např. symbolická regrese) pomocí kartézského genetického programování, které využívá souběžné učení.
3. Program z bodu 2 implementujte.
4. Ověřte funkčnost programu na zadaných úlohách.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Wiglasz Michal, Ing.**, UPSY FIT VUT

Datum zadání: 1. listopadu 2015

Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Tato práce se zabývá integrací souběžného učení do kartézského genetického programování. Úlohu symbolické regrese se již povedlo vyřešit kartézským genetickým programováním, ovšem tato metoda není dokonalá. Je totiž relativně pomalá a při některých úlohách má tendenci nenalézat požadované řešení. Ale se souběžným učením lze vylepšit některé z těchto vlastností. V této práci je představena plasticita genotypu, která je založena na Baldwinově efektu. Tento přístup umožňuje jedinci změnit jeho fenotyp během generace. Souběžné učení bylo testováno na pěti rozdílných úlohách pro symbolickou regresi. V experimentech se ukázalo, že pomocí souběžného učení lze dosáhnout až 15násobného urychlení evoluce oproti standardnímu kartézskému genetickému programování bez učení.

Abstract

This thesis deals with the integration of co-learning into cartesian genetic programming. The task of symbolic regression was already solved by cartesian genetic programming, but this method is not perfect yet. It is relatively slow and for certain tasks it tends not to find the desired result. However with co-learning we can enhance some of these attributes. In this project we introduce a genotype plasticity, which is based on Baldwins effect. This approach allows us to change the phenotype of an individual while generation is running. Co-learning algorithms were tested on five different symbolic regression tasks. The best enhancement delivered in experiments by co-learning was that the speed of finding a result was 15 times faster compared to the algorithm without co-learning.

Klíčová slova

Souběžné učení, kartézské genetické programování, evoluční algoritmus, Baldwinův efekt, plasticita genotypu, symbolická regrese.

Keywords

Co-learning, cartesian genetic programming, evolutionary algorithm, Baldwin effect, symbolic regression

Citace

KORGO, Jakub. *Souběžné učení v kartézském genetickém programování*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Michal Wiglasz.

Souběžné učení v kartézském genetickém programování

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Michala Wiglasze.

....
Jakub Korgo
17. května 2016

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Michalu Wiglaszovi za jeho rady a návrhy vedoucí k vyšší kvalitě této práce. Dále bych chtěl poděkovat mému nejbližšímu okolí, bez jejichž podpory by se tato práce nikdy neuskutečnila.

© Jakub Korgo, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **2**

2 Evoluční algoritmy **3**
- 2.1 Algoritmy vycházející z evolučních algoritmů . . . 5
- 2.2 Genetické programování . . . 6
- 2.3 Kartézské genetické programování . . . 9
- 2.4 Kartézské genetické programování se souběžným učením . . . 10
- 2.5 Symbolická regrese . . . 11

3 Návrh **12**
- 3.1 Chromozom . . . 12
- 3.2 Výpočet fitness hodnoty . . . 12
- 3.3 Získání aktivních uzlů . . . 13
- 3.4 CGP se souběžným učením . . . 13

4 Implementace **16**
- 4.1 Parametrizace spouštěných programů . . . 16
- 4.2 Načítání souboru s trénovacími vektory . . . 17
- 4.3 Generování náhodných čísel . . . 17
- 4.4 Sběr dat a jejich záznam . . . 18
- 4.5 Systém zálohování . . . 18
- 4.6 Kompilace a spuštění . . . 18

5 Experimentální vyhodnocení **20**
- 5.1 Úlohy použité pro experimenty . . . 20
- 5.2 Vlastnosti algoritmů se souběžným učením . . . 20
- 5.3 Srovnání jednotlivých algoritmů . . . 22
- 5.4 Shrnutí výsledků . . . 26

6 Závěr **27**

Literatura **28**

Kapitola 1

Úvod

Snaha vylepšovat algoritmy je neustálá. Takovéto vylepšení může nést plody v podobě podstatného zkrácení doby jejich vykonávání, případně vylepšení jiné vlastnosti. Při zaměření se na prohledávání prostoru možných řešení, mezi první vytvořené algoritmy zde patří metody používající náhodné prohledávání, které si pamatují nejlepší nalezené řešení. Vylepšení tohoto algoritmu pro některé případy byl horolezecký algoritmus, který si vybírá nejlepší řešení ve svém okolí a v tomto bodě tento proces opakuje. Mezi úspěšné algoritmy pro prohledávání prostoru řešení spadají i evoluční algoritmy.

Inspirací pro evoluční algoritmy byla Darwinova teorie. Tyto algoritmy byly z počátku populární pro problém hledání optimálních parametrů. Jedno z mnoha vylepšení bylo genetické programování. Zde byl algoritmus schopnen generovat spustitelné struktury pro řešení problému. Avšak i tento algoritmus měl své chyby, přičemž některé z nich byly vyřešeny kartézským genetickým programováním, na což navazuje tato práce, která se snaží zjistit jestli lze kartézské genetické programování vylepšit souběžným učením.

Cílem této práce je seznámení se s evolučními algoritmy, kartézským genetickým programováním, Baldwinovim efektem a symbolickou regresí. Po vysvětlení těchto oblastí je popsáno, jak byly navrženy a implementovány algoritmy pro souběžné učení k řešení symbolické regrese. Nakonec takto implementované algoritmy byly porovnány na předem vybraných úlohách.

V této práci se objevují následující kapitoly: Kapitola 2 se zabývá úvodem do evolučních algoritmů, následně jsou v této kapitole v části 2.1 rozebrány nejznámější algoritmy vycházející z evolučních algoritmu. Část 2.2 popisuje genetické programování, ze kterého vychází kartézské genetické programování 2.3. V části 2.4 je popsán Baldwinův efekt a plasticita v souvislosti se souběžným učením. V poslední části 2.5 jsou k vidění obecné informace ohledně typu testované úlohy – symbolické regrese. V kapitole 3 je dále navrženo, jak je možné řešit symbolickou regresi za pomoci kartézského genetického programování se souběžným učením. Způsob, jak následně byly jednotlivé části implementovány, je popsán v kapitole 4. Kapitola 5 obsahuje popis experimentů, parametrů pro experimenty a výsledky experimentů na testovaných úlohách.

Kapitola 2

Evoluční algoritmy

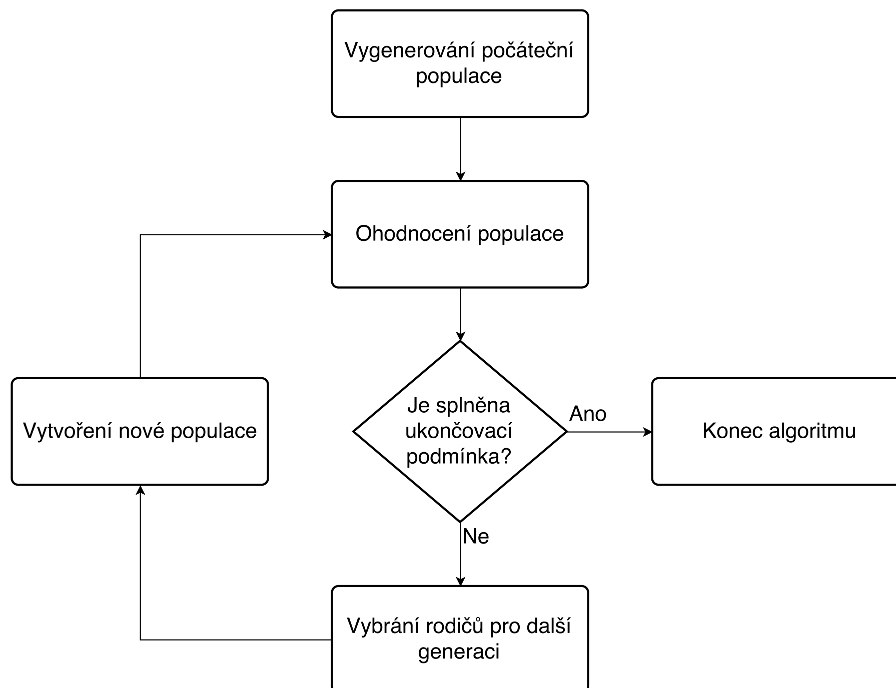
Darwinova evoluční teorie byla inspirací pro evoluční algoritmy. Vychází se zde z případů existujících v přírodě a podpořených touto teorií, jako jsou například populace měnící se v průběhu času nebo inspirace pravidlem „přežije silnější“. Tyto algoritmy jsou často používány u úloh, kde je prohledávaný prostor řešení obzvláště velký a průchod systematickým algoritmem by trval dlouhou dobu. Aby bylo možné přesně si přesně vysvětlit fungování evolučních algoritmů, musíme nejprve definovat klíčové pojmy, které jsou definovány shodně s knihou L. Sekaniny [8].

Pojmy

- **Gen**: Nejmenší jednotka genotypu, která může nabývat hodnot z předem dané abecedy.
- **Genotyp/chromozom**: Zakódovaná reprezentace jedince složená z genů (nemusí být všechny geny stejného typu), která může mít proměnnou délku.
- **Fenotyp**: Kandidátní řešení získané z obsahu genotypu za pomoci předem daných předpisů na vytvoření fenotypu.
- **Jedinec**: Reprezentace jednoho řešení prohledávaného prostoru.
- **Populace**: Množina obsahující předem dané množství jedinců.
- **Fitness funkce**: Matematická funkce hodnotící vlastnosti fenotypu vůči hledanému výsledku.

Princip evolučního algoritmu

Schéma algoritmu je zobrazeno na obrázku 2.1. Celý algoritmus začíná vytvořením počáteční populace o předem dané velikosti. Ta je ve většině případů vytvořena náhodně, v některých situacích lze využít předem známých řešení problému. Tato znalost nám umožňuje snížit počet generací potřebných k nalezení řešení. Následuje ohodnocení všech jedinců v populaci za pomocí fitness funkce. Pokud je nalezeno řešení s požadovanou hodnotou fitness, vyhledávání končí. Jinak následuje výběr rodiče pro další generaci. Pro výběr rodičů a vytváření potomků existuje celá řada algoritmů. Za jejich pomoci se následně vytváří nová populace. Po vytvoření nové populace následuje její hodnocení dle fitness funkce. Pokud je dosaženo požadovaného řešení, nebo byl dosažen maximální počet generací, je vyhledávání ukončeno, jinak je vytvořena další generace.



Obrázek 2.1: Zjednodušený algoritmus evolučního algoritmu.

Získání fitness hodnoty

Kvalitu jedince s ohledem na řešený problém popisuje hodnota fitness, která je vypočtena fitness funkcí. Hodnota fitness může byt vyjádřena mnohými způsoby.

- **Hrubá fitness** je vyjádření fitness hodnoty v hodnotách přirozených problémové doméně.
- **Standardizovaná fitness** je transformace hrubé fitness hodnoty tak, že nižší hodnota fitness značí lepšího jedince.
- **Přizpůsobená hodnota fitness** je získána převrácením hodnoty součtu standardizované fitness a čísla jedna. Díky tomuto výpočtu jsou hodnoty v intervalu $(0; 1\rangle$. Zde vyšší hodnota značí lepšího jedince.
- **Normalizovaná hodnota fitness** vznikne podílem hrubé hodnoty fitness jedince a součtu hrubých hodnot populace, kde jedinec s vyšší hodnotou je lepší. Tímto výpočtem zjistíme kvalitu jedince vůči populaci, avšak tato hodnota samotná neříká nic o kvalitě ve srovnání s hledaným řešením.

Výběr rodičů a generování další populace

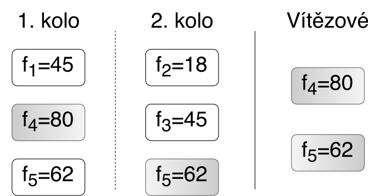
Tato část je srdcem evolučních algoritmů, protože při použití nevhodného algoritmu se může stát, že cílové řešení nebude nalezeno, nebo bude vyhledáváno podstatně déle, než za použití jiných algoritmů.

Roli zde hrají dvě části. Počet a výběr jedinců, kteří se dostanou ze staré populace do nové a způsob výběru rodičů, ze kterých jsou pomocí genetických operátorů tvořeni potomci. Nové populace mohou být nepřekrývající, což znamená, že do nové populace není

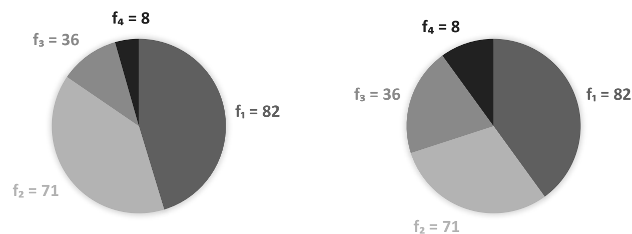
vložen jedinec z populace předchozí (což nevylučuje, že může vzniknout potomek, který je shodný s některým z jedinců v původní populaci), nebo překrývající, kdy se několik jedinců z původní populace dostane do populace nové. Pro výběr jedinců, kteří se stanou rodiči pro následující populaci, se používají různé algoritmy. Nejčastěji se používají tyto:

- **Deterministická selekce**: Je vybráno N jedinců s nejvyšší fitness hodnotou.
- **Turnajová selekce**: Nkrát je vybráno náhodně M jedinců. Z těchto skupin je z každé vybrán jedinec s nejlepší fitness hodnotou (obrázek 2.2a).
- **Proporcionální selekce**: Je vybáno náhodně N jedinců, přičemž pravděpodobnost výběru jedince je přímo úměrná jeho fitness hodnotě (obrázek 2.2b).
- **Selekce podle pořadí**: Populace je seřazena podle fitness hodnoty, následně je vybráno náhodně N jedinců, přičemž pravděpodobnost výběru jedince je přímo úměrná jeho pořadí. Rozdíl oproti proporcionální selekci je ten, že je zde vyšší pravděpodobnost výběru horšího jedince, tím je zaručena vyšší diverzita populace. To lze využít k prohledávání širšího prostoru řešení (obrázek 2.2c).

Za pomocí rodičů se následně generují potomci. Pro generování potomků je využito zejména křížení, po kterém může následovat mutace. Pro křížení je třeba dvou rodičů, ve kterých jsou vybrána místa, v nichž si tito rodiče vymění geny, čímž jsou vytvořeni noví potomci. U mutace nový potomek vznikne označením N genů mutovaného jedince, u nichž je náhodně změněna hodnota. Je třeba si dát pozor na to, aby nová hodnota nebyla mimo povolenou množinu hodnot pro daný gen.



(a) Turnajová selekce



(b) Proporcilnální selekce (c) Selekce podle pořadí

Obrázek 2.2: Algoritmy pro výběr rodičů.

2.1 Algoritmy vycházející z evolučních algoritmů

Počátky evolučních algoritmů jsou datovány k 50. letům minulého století. Ovšem algoritmy, které se nyní používají, nejčastěji vznikly na přelomu 60. a 70. let. Mezi hlavní algoritmy, spadající pod evoluční algoritmy, se řadí evoluční programování, evoluční strategie a genetické algoritmy a genetické programování [10].

Evoluční programování

Evoluční programování bylo vytvořeno v roce 1966 a používalo reprezentaci podle problému, který řešila. Například pro řešení problému optimalizace celočíselných hodnot, byli jedinci uloženi jako celočíselné vektory. Podobně řešeny byly za pomocí grafů například i stavové automaty. Jako operátor pro generování nových jedinců zde byla používána výhradně mutace. Pro výběr nových rodičů byly používány pravděpodobnostní funkce, zejména proporcionální selekce.

Evoluční strategie

Evoluční strategie (ES) byla původně představena jako algoritmus s velikostí populace jedna, kde byl vygenerován jeden potomek za pomoci mutace, a následně byl vybrán nejlepší z této dvojice pro další generaci. Později byly představeny dva algoritmy: ES$(\mu + \lambda)$ a ES(μ, λ), kde μ značí počet rodičů a λ značí počet potomků. Pro reprezentaci chromozomu byla opět použita reprezentace podle problému. ES používá jako operátor křížení, po němž následuje mutace. Výběr rodičů pro novou generaci je zde řešen vybráním μ nejlepších jedinců. U ES$(\mu + \lambda)$ jsou rodiče vybráni z množiny potomků a rodičů původní generace, u ES(μ, λ) je vybíráno pouze z množiny potomků.

Genetické algoritmy

Genetické algoritmy využívají reprezentaci nezávislou na řešeném problému. Mezi tyto reprezentace patří například binární řetězec nebo neuronová síť, přičemž tyto reprezentace jsou schopny pokryt širokou škálu problémů. Zde je pro generování nových jedinců klíčový operátor křížení, který je následně podpořen mutací s malou pravděpodobností. Selekce rodičů je zde nejčastěji implementována jako proporcionální selekce.

Genetické programování

Pracuje se spustitelnými stromovými strukturami. Tato metoda je podrobně popsána v podkapitole 2.2.

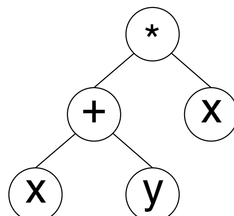
2.2 Genetické programování

Genetické programování je evoluční algoritmus, jehož cílem není pouze hledat optimální hodnoty parametrů zakódovaných v chromozomu, ale také generovat spustitelné struktury, jež jsou schopny řešit požadovanou úlohu. Pro řešení úlohy pomocí genetického programování je třeba provést přípravné kroky:

- Definovat množinu terminálů a funkcí,
- definovat způsob výpočtu hodnoty fitness,
- definovat parametry potřebné pro evoluci (jako jsou třeba velikost populace, počet generací atd.),
- definovat ukončovací podmínky.

Reprezentace programů

Programy jsou definovány jako struktury skládající se z terminálů a funkcí. Dále je definováno, jakým způsobem jsou terminály a funkce použity. Třemi základními strukturami jsou stromy, lineární struktury a grafové struktury. Podle typu struktury je jejím průchodem vykoná program. Na obrázku 2.3 lze vidět, jak je reprezentován matematický výraz ve stromové struktuře.



Obrázek 2.3: Reprezentace výrazu $(x + y) * x$ ve stromové struktuře.

Množina terminálů

Tato množina obsahuje primární vstupy, konstanty a funkce bez argumentů. V klasické variantě genetického programování je na počátku zvolena množina konstant. Tyto hodnoty nejsou měněny a nových hodnot je získáno za pomocí kombinací těchto konstant. Další možnost je například generovat konstanty zcela náhodně ve zvoleném intervalu.

Množina funkcí

Funkce v této množině zpracovávají vstupní argumenty a podle zadané funkce následně vytvoří výstup. Tyto funkce mají dva důležité požadavky. První z nich je uzavřenost. To znamená, že výstup funkce musí být použitelný jako vstupní argument pro ostatní funkce. Další požadavek je, aby byly použity chráněné varianty pro funkce, které mají někde nedefinované hodnoty. Například při dělení nulou je třeba zajistit návrat takové bezpečné hodnoty, kterou dokážou zpracovat následující funkce, popřípadě penalizovat fitness hodnotu daného jedince, ve kterém by se taková operace objevila. Dále je vhodné nedefinovat příliš složité funkce z důvodu optimalizace rychlosti evoluce.

Inicializace populace

Významná část genetického programování a její první krok je inicializace populace. Zde jsou vytvořeni první jedinci, ze kterých jsou následně tvořeny další populace. Mezi nejčastější způsoby generování stromové struktury patří metody *grow*, *full* a *ramped half-and-half*. Všechny tyto metody mají za společnou vlastnost použití limitované hloubky růstu.

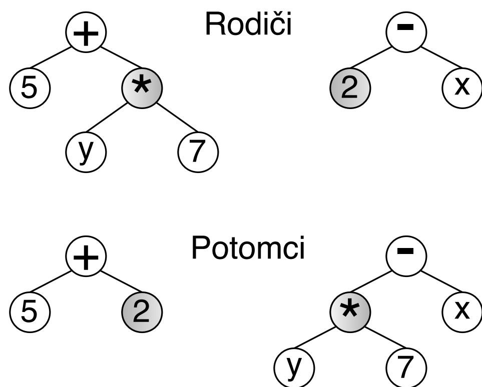
Metoda *grow* tvoří strom tak, že jako kořenový uzel vybere náhodnou funkci z množiny funkcí. Poté následuje doplňování listů. Pokud je již list v maximální hloubce stromu, je vybrán náhodný terminál, jinak je vybrán náhodný prvek ze spojené množiny terminálů a funkcí. Doplňování listů pokračuje do té doby, dokud existují nezaplněné listy.

Metoda *full* je metodě grow podobná. Využívá stejnou inicializaci kořenu, jediný rozdíl je v generování listů stromů. Pokud není list na maximální hloubce stromu, je vybrán náhodný prvek z množiny funkcí, jinak je vybrán prvek z množiny terminálů shodně s metodou grow.

Metoda *ramped half-and-half* využívá obou předchozích metod v poměru 1:1, přičemž pro každý strom je ještě navíc vybírána náhodná maximální hloubka stromu, která nemůže přesáhnout zadanou maximální hloubku stromu. Tato metoda generování stromu poskytuje vysokou diverzitu populace.

Operátory genetického programování

Pro vytvoření nové populace ze staré je třeba aplikovat na rodiče původní populace genetický operátor. Mezi základní operátory genetického programování patří operátor křížení a mutace. Pro oba operátory existuje mnoho variant. Například pro stromově orientované programování se základní varianta křížení chová tak, že je po výběru dvou rodičů z původní populace vybrán náhodný uzel a jemu náležící podstrom v obou rodičích. Následuje výměna těchto podstromů a výsledek této operace jsou dva nové stromy, viz obrázek 2.4. Oproti tomu operátor mutace pracuje jen s jedním jedincem. Tento operátor vybere náhodně jeden uzel a jeho podstrom nahradí novým náhodně vygenerovaným stromem. Pro generování stromu při mutaci jsou většinou aplikovány stejná pravidla jako pro generování počáteční populace.



Obrázek 2.4: Ukázka možného křížení dvou stromů.

Bloat

Bloat je situace v oblasti genetického programování, kdy začne po určitém počtu generací rapidně narůstat velikost stromů, přičemž tento nárůst většinou nemá kladný dopad na fitness hodnotu. Tento efekt má pak za následek zvýšení času potřebného pro získání řešení, protože narůstá počet výpočtů za generaci, tím pádem je nárust stromu většinou nežádoucí. S velikostí stromu se taky zvyšuje pravděpodobnost na vytvoření intronu. To je kód, který nijak neovlivňuje výstupní hodnotu. Příklad funkce větve, která splňuje definici intronu, je například $x/1$.

Řešení tohoto problému je několik, přičemž většina z nich si zakládá na omezení maximální hloubky stromů. Mezi tato řešení patří například snížení fitness hodnoty jedinců, kteří přesahují maximální hloubku stromů, nebo znemožnění těmto jedincům se dostat do další generace. Další varianta je ořezání, kde jsou všechny funkční uzly, nacházející se na úrovni maximální hloubky stromu, nahrazeny uzly z množiny terminálů.

2.3 Kartézské genetické programování

Tato metoda byla představena Millerem a Thomsonem [7]. Kartézské genetické programování (dále CGP) je varianta genetického programování, ve které jsou programy reprezentovány jako orientované acyklické grafy.

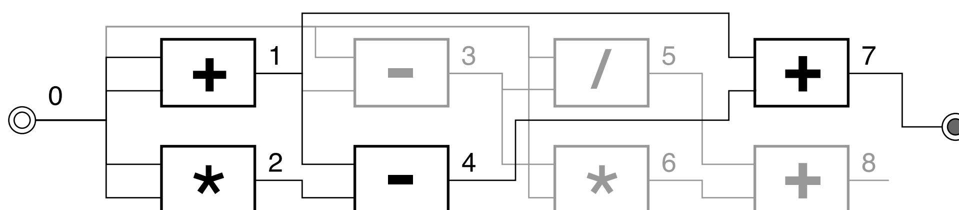
Tyto grafy jsou zobrazovány dvourozměrnou maticí výpočetních uzlů o předem daném počtu řádků a sloupců. Dále je ještě v CGP definován počet primárních vstupů a výstupů. Primární výstupy a vstupní argumenty uzlů mohou být připojeny na primární vstup, nebo na výstup libovolného z předchozích uzlů. S připojováním na předchozí uzly také souvisí parametr l-back, který určuje, o kolik sloupců zpět se může nanejvýš daný uzel připojit. Většinou je pro tvoření nové generace použit pouze operátor mutace. Byly provedeny pokusy s operátorem křížení, u kterých však ve většině případů bylo třeba více času, než když byl použit pouze operátor mutace. Vyskytly se však i případy, kdy po úpravě chromozomu CGP byla metoda křížení s mutací efektivnější, než mutace samotná [2].

Kódování chromozomu

CGP je uloženo v paměti jako pole celých čísel o předem dané velikosti, ve kterém jsou postupně uloženy hodnoty genů výpočetních uzlů. Každý uzel je složen z $n+1$ genů. Prvních n genů určuje vstupní parametry pro jeho funkci a poslední gen určuje, jaká funkce je uzlem vykonána. Po uložení uzlů do chromozomu jsou na konec pole vloženy indexy uzlů, na které jsou připojeny primární výstupy. Například pro obrázek 2.5 může mít chromozom tvar $[0, 0, +, 0, 0, *, 0, 1, -, 1, 2, -, 0, 3, /, 3, 0, *, 1, 4, +, 5, 6, +, 7]$. Výhoda, která plyne z tohoto kódování, je taková, že CGP nehrozí bloat, délka chromozomu je totiž pořád stejná.

Avšak při získání fenotypu z chromozomu už nemusí být použity všechny uzly. Získání fenotypu započne zjištěním, na které uzly jsou připojeny primární výstupy, a tyto uzly jsou následně prohlášeny za aktivní. Dále se u takto označených uzlů zjišťuje, které uzly jsou na tyto uzly připojeny jako parametr. Tyto uzly jsou také označeny za aktivní a znovu proběhne hledání aktivních uzlů.

Po prohledání celého chromozomu je ze všech aktivních uzlů vytvořen fenotyp. Uzly, které nebyly nepoužity ve fenotypu, se nazývají uzly neaktivní. CGP má tedy schopnost při konstantní délce chromozomu tvořit fenotypy o různé délce, záleží pouze na způsobu propojení uzlů v chromozomu.



Obrázek 2.5: Příklad jedince v CGP, šedě jsou znázorněny neaktivní uzly.

Průběh algoritmu

V algoritmu CGP je použita ES$(1 + \lambda)$, což znamená, že je vybrán jeden rodič, z něhož je vytvořeno λ potomků, kteří jsou generování pouze za pomoci operátoru mutace. Rozdíl je jen ve výběru rodiče pro další populaci. Primární faktor zde, stejně jako v evolučních algoritmech, tvoří fitness hodnota. Pokud se vyskytnou dva jedinci se stejnou fitness hodnotou,

jedinec, který byl pro danou generaci rodičem, se nemůže stát rodičem pro generaci další. Důvod takového výběru je, že nastala neutrální mutace rodiče. To je taková mutace, která mění chromozom, avšak nemění fenotyp jedince. Tento typ mutace je pro CGP důležitý, protože umožňuje lepší prohledávání prostoru všech možných řešení [6].

2.4 Kartézské genetické programování se souběžným učením

Doposud uvedené evoluční algoritmy neobsahují fázi učení, a tedy fitness hodnota jedince závisí pouze na jeho genotypu. V přírodě toto ovšem neplatí, zde jsou jedinci schopni se naučit co nejefektivněji využít zděděné vlohy, a tak jejich výsledná fitness hodnota může být vyšší, než by byla bez fáze učení. Vztahem mezi evolucí a učením se zabýval již J. M. Baldwin na konci 19. století [1].

Baldwinův efekt

Baldwin se ve svém výzkumu zabýval způsoby, jakými se mohou vyvinout složité instinktivní vlastnosti. Aby se takový instinkt vyvinul, je třeba učinit v Darwinově evoluční teorii malé kroky, přičemž tento instinkt jedinci nemusí dávat ve chvíli svého vývoje zvýhodnění. Baldwinův efekt tvrdí, že učením si jedinec může zvýšit šance na přežití, i když instinkt, který by mu mohl výrazně pomoci, je teprve ve vývinu. Například nedokonale vyvinuté oko může dát jedinci evoluční výhodu, pokud se s ním naučí pracovat [11]. Pro představu, zde by tento příklad mohl znamenat schopnost naučit se pracovat s okem, které umí rozeznat jenom světlo. Učení je však časově náročná činnost, která nemusí mít vždy užitečný efekt. Proto se naučené vlastnosti předávané po více generací mohou stát časem instinkty [3]. O schopnosti jedince se učit pojednává plasticita.

Plasticita

Plasticita je schopnost jedince změnit svůj genotyp v závislosti na jeho prostředí. Jako příklad může být použito domestikované zvíře chované v elektrickém ohradníku. Zvíře se pase na trávě a má volný přístup k ohradníku. Časem nastane situace, kdy se pokusí sníst trávu, při čemž se dotkne ohradníku. Zvíře dostane elektrický šok. Zvíře s vysokou plasticitou se naučí, že u ohradníků si má dávat pozor a do konce svého života si pravděpodobně tuto informaci uchová. Také existuje možnost předání informace dalším generacím, kdy se mládě může tomuto chování od rodiče naučit. V průběhu dalších generací se toto chování může stát instinktivním. Oproti tomu zvířata s nízkou nebo žádnou plasticitou si neuchovají během svého života informaci o tom, že si mají dávat pozor v blízkosti ohradníku. Například tuto vlastnost by mohla získat až po několika generacích, pokud by zvířata, která se nedotýkají ohradníku, byla nějak zvýhodněna při rozmnožování.

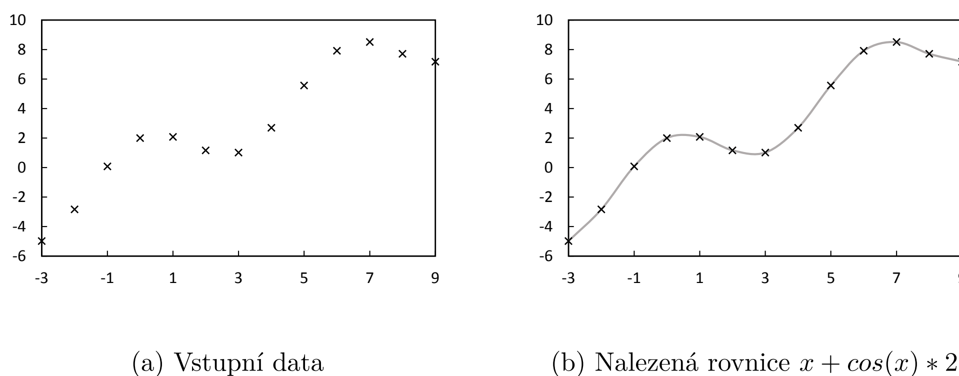
Baldwinův efekt a CGP

Základní verze CGP nemá implementovanou plasticitu. Primární výstup je připojen na určitý uzel, jenž je definován genotypem, a jediný způsob, jak toto změnit, je mutace, když je generována nová populace. Pro zavedení plasticity do CGP, je třeba najít způsob ovlivnění genotypu i za běhu generace, nejen při tvoření generace nové. Toho lze dosáhnout například upravením fáze vyhodnocování fitness hodnoty tak, aby bylo možné změnit za určitých podmínek primární výstup. Takto upravený algoritmus budeme nazývat CGP se souběžným učením. Vlastnostmi CGP se souběžným učením se už zabývaly některé práce,

například [12] se zabývala použití tohoto algoritmu pro rozeznání obrázků a práce [4], která se zabývala návrhem elektronických obvodů.

2.5 Symbolická regrese

Jednou z úloh, u které se ukázalo, že je ji vhodné řešit pomoci genetického programování, je symbolická regrese[13]. Úkolem symbolické regrese je vytvořit funkci, která mapuje vztah mezi nezávislými a závislými proměnnými s povolenou odchylkou, ukázka je zobrazena na obrázku 2.6. Hledání tohoto vztahu je řešeno za pomoci množiny funkčních bloků, které jsou skládány do funkcí. Do této množiny funkcí jsou vkládány operace nebo funkce, u kterých se předpokládá, že by mohly vést k řešení úlohy. Výhodou symbolické regrese je, že není omezená předpoklady ohledně vztahu vstupních dat. U jiných regresních metod se vychází z předpokládané rovnice, u které jsou upravovány parametry. Pro nalezení křivky, která alespoň přibližně procházející vstupními daty, musí být zvolena správná regresní rovnice. U symbolické regrese lze takto najít závislosti i bez jakýchkoliv znalostí ohledně vstupních dat [5]. Nevýhodou je, že neexistuje matematický aparát pro její řešení a je třeba použití prohledávacích metod.



(a) Vstupní data (b) Nalezená rovnice $x + cos(x) * 2$

Obrázek 2.6: Příklad vstupu a výsledku symbolické regrese.

Kapitola 3

Návrh

V této práci byly navrženy tři algoritmy. Algoritmus CGP, vůči kterému se porovnávaly dosažené experimentální výsledky a dvě varianty CGP se souběžným učením. První, u kterého si uživatel zvolí, na kolika uzlech má probíhat učení a druhý, který využívá učení nad všemi uzly.

3.1 Chromozom

Celý chromozom byl vytvořen s ohledem na řešení symbolické regrese. Výpočetní uzel obsahuje 2 geny pro vstupní parametry a 1 gen značící funkci tohoto uzlu. Tato konfigurace uzlu umožňuje pokrýt potřebné matematické funkce, ať už jednovstupé (druhý vstup je ignorován) nebo dvouvstupé. Množina funkcí byla vybrána shodně s článkem [9]. Seznam těchto funkcí a jim odpovídajících hodnot v genu je v tabulce 3.1. V každém uzlu jsou postupně zapsány dva geny vstupů a následně gen nesoucí informaci o funkci vykonávanou uzlem.

Číslo funkce	**Vykonaná operace**
0	$i_1 + i_2$
1	$i_1 - i_2$
2	i_1/i_2
3	$i_1 * i_2$
4	$sin(i_1)$
5	$cos(i_1)$
6	$exp(i_1)$
7	$log(i_1)$

Tabulka 3.1: tabulka funkcí pro vstupy i_1 a i_2.

3.2 Výpočet fitness hodnoty

Pro výpočet fitness hodnoty bylo na výběr ze dvou metod [9]. První možností bylo za fitness hodnotu považovat průměrnou odchylku vypočtenou ze získaných hodnot y a požadovaných hodnot t:

$$fitness = \frac{1}{n}\sum_{i=1}^{n}|y(i) - t(i)| \tag{3.1}$$

Vyhledávání v této variantě končí ve chvíli, kdy je odchylka menší, než uživatelem zadaná maximální odchylka.

Druhá varianta je použití metody *hit-miss*. Zde je vyhodnocování fitness hodnoty rozděleno na dvě části. Pro první část musí být definována velikost povolené odchylky ε, a pro druhou část musí být známo, kolik zásahů nám stačí k prohlášení, že nalezené řešení je přijatelné. Výpočet probíhá tak, že jsou všechny získané hodnoty y porovnávány s požadovanymi hodnotami t a je zjišťováno, zda je nalezený bod ještě v mezích odchylky, či nikoliv. Pokud se nachází v mezích odchylky, je jako výsledek přiřazena 1, v opačném případě 0. Nakonec proběhne součet výsledných hodnot, který nám určí fitness hodnotu nalezené funkce:

$$fitness = \sum_{i=1}^{n} \begin{cases} 1, & |y(i) - t(i)| < \varepsilon \\ 0, & |y(i) - t(i)| \geq \varepsilon \end{cases} \tag{3.2}$$

Pokud je daná fitness hodnota rovna nebo vyšší, než uživatelem definovaná hodnota fitness hodnota, tak bylo nalezeno konečné řešení.

3.3 Získání aktivních uzlů

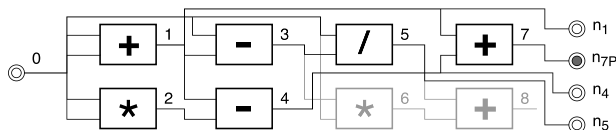
Pro urychlení výpočtu generací je vhodné neprovádět výpočty nad neaktivními uzly. Pro získání aktivních uzlů bylo vybíráno ze dvou metod. První metodou je rekurzivní sestup od uzlu, na který ukazuje primární výstup. Tento uzel je označen za aktivní a uzly, které jsou jeho parametry, jsou taktéž označeny za aktivní a stejným způsobem je tento proces opakován na jejich parametry. Druhá varianta je postupný průchod uzlů od posledního k prvnímu. Volba tohoto přístupu je možná, protože je zajištěno, že na vstupní parametry jsou připojeny pouze uzly předchozí. Na počátku této metody je označen uzel, na který ukazuje primární výstup. Poté následuje průchod polem uzlů, kde kdykoliv, kdy je nalezen aktivní uzel, jsou uzly, které má tento nalezený uzel jako vstupní parametr, označeny za aktivní. Zde zmíněná druhá varianta je výhodná v tom, že má konstantní paměťovou i časovou náročnost, a to je také důvod, proč byla v této práci použita.

3.4 CGP se souběžným učením

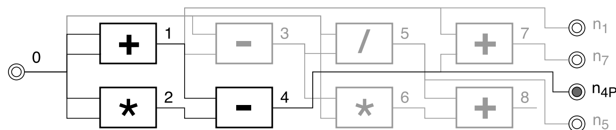
Algoritmus CGP se souběžným učením vychází z algoritmu CGP (podkapitola 2.3). Jediná změna v tomto algoritmu je ta, že fáze, kdy se má vyhodnocovat hodnota fitness jedince, je rozšířena o algoritmus aplikující souběžné učení. Díky tomu má jedinec schopnost změnit si chromozom v průběhu generace. Pro tuto úlohu byly vytvořeny dva algoritmy.

Algoritmus na bázi výběru několika uzlů

Zde je využito přístupu, který se snaží docílit co nejmenší zatížení jedince souběžným učením. Na počátku hodnocení fitness je k původně primárnímu uzlu náhodně vybráno několik dalších uzlů, u kterých se bude zjišťovat fitness hodnota. Následně je vytvořen fenotyp, který obsahuje všechny uzly z této množiny (obrázek 3.1a). Pro získání takového fenotypu je použit postupný průchod uzly od posledního k prvnímu, popsaný v 3.3, kde



(a) Fenotyp vytvořený z jedince pro získání fitness hodnot při souběžném učení.



(b) Výsledný chromozom po aplikaci souběžného učení.

Uzel	Fitness hodnota
1	$f = 24$
7	$f = 49$
4	$\boldsymbol{f = 63}$
5	$f = 51$

(c) Získané fitness hodnoty z vybraných uzlů, nejlepší je prohlášena za fitness hodnotu jedince.

Obrázek 3.1: Znázornění získání fitness hodnoty a chromozomu u algoritmu na bázi výběru několika uzlů. U jedinců jsou tmavě zvýrazněny jejich fenotypy a index P u výstupu značí aktuální primární výstup.

je algoritmus upraven tak, že na počátku jsou jako aktivní uzly označeny všechny uzly, u kterých se zjišťuje fitness hodnota.

Následně proběhne vyhodnocování fitness hodnot u vybraných uzlů (tabulka 3.1c). Po té je z takto získaných fitness hodnot vybrána ta nejvyšší, ta je nastavena jako fitness hodnota jedince a na uzel, kterému tato fitness hodnota přísluší, je připojen primární výstup (obrázek 3.1b). Následně algoritmus pokračuje stejně jako u standardního CGP.

Algoritmus na bázi zkoumání všech uzlů

Zde byl vybrán způsob prozkoumání celého jedince. Pro učení jsou zvoleny všechny uzly, takže není třeba zjišťovat strom fenotypu a lze tento krok přeskočit, stejně jako krok pro generování náhodných uzlů, potřebný u předchozí varianty (obrázek 3.2a). Následuje vyhodnocení fitness hodnot všech uzlů, ze kterého je vybrána nejlepší fitness hodnota (tabulka 3.2c), ta je nastavena jako fitness hodnota jedince a na uzel s touto fitness hodnotou je připojen primární výstup (obrázek 3.2b). Následovně algoritmus pokračuje stejně jako u standardního CGP.

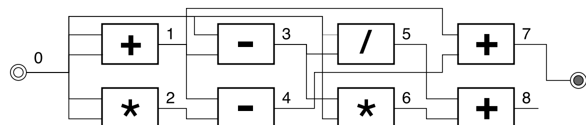
Tento přístup oproti předchozímu má tu výhodu, že je zde plně využito potenciálu jedince, protože se nemůže stát, že by byl v chromozomu nějaký nenalezený uzel s lepší fitness hodnotou. Oproti tomu tento algoritmus má nevýhodu v případě nízkých hodnot pravděpodobnosti mutace. Při takové pravděpodobnosti mutace je totiž vyšší pravděpodobnost, že tato mutace ovlivní jen pár uzlů z celého chromozomu, přičemž nová fitness hodnota by byla v této generaci opět počítána pro všechny uzly znova.

Vlastnosti takto implementovaného algoritmu

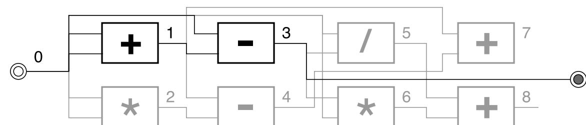
Mezi pozitivní důsledky této úpravy standardního CGP patří možné urychlení evoluce, tím pádem snížení počtu generací potřebných k získání řešení. Toto urychlení vychází z možnosti nalezení uzlu s lepší fitness hodnotou, než je fitness hodnota aktuálního primárního výstupu. U CGP by takové nalezení vyžadovalo vhodnou mutaci primárního výstupu při generování potomků.

Dále je zde využito toho, že při zkoumání fitness hodnoty více uzlů v jednom jedinci je možné vygenerovat fenotyp, který vyžaduje méně výpočtů pro získání fitness hodnot,

než kdyby byl pro každý takový uzel v jedinci vytvářen nový fenotyp. Díky tomu není obtížnost výpočtu více uzlů v jednom jedinci razantně náročnější, než obtížnost výpočtu uzlu jediného. Avšak poslední zmíněná vlastnost je i nevýhodou. Algoritmy se souběžným učením nemusí najít uzel s lepší fitness hodnotou, než ten, na kterém byl původně připojen primární výstup, což způsobí zvýšení časové náročnosti výpočtu bez jakéhokoliv zisku.



(a) Fenotyp vytvořený z jedince pro získání fitness hodnot při souběžném učení.



(b) Výsledný chromozom po aplikaci souběžného učení.

Uzel	Fitness hodnota
1	$f = 24$
2	$f = 8$
3	$\boldsymbol{f = 65}$
4	$f = 63$
5	$f = 51$
6	$f = 38$
7	$f = 49$
8	$f = 26$

(c) Nejlepší fitness hodnota ze všech nalezených je prohlášena za fitness hodnotu jedince.

Obrázek 3.2: Znázornění získání fitness hodnoty a chromozomu u algoritmu na bázi zkoumání všech uzlů. U jedinců jsou tmavě zvýrazněny jejich fenotypy. Pro jednoduchost byl jako výstup v obrázku zobrazen jen primární výstup.

Kapitola 4

Implementace

Jelikož jsou tyto algoritmy náročné na výpočetní výkon a optimalizaci, byla snaha vybrat programovací jazyk, který by byl pro tyto účely vhodný. Jako programovací jazyk byl zvolen jazyk C++ se standardem C++11, to hlavně z důvodu rychlosti a zabudované knihovny s kvalitními generátory náhodných čísel.

4.1 Parametrizace spouštěných programů

Jako místo pro nastavení většiny parametrů bylo zvoleno nastavení v hlavičkových souborech. Tento způsob zadávání parametrů byl zvolen z důvodu, že při překládání programu může překladač provést optimalizace, které by bez znalosti těchto dat nemohl provést. Nevýhodou tohoto přístupu je nutnost přeložit program po každé změně parametrů.

Výčet parametrů v hlavičkovém souboru shodný pro všechny implementace obsahuje:

- `VSTUPNI_SOUBOR` – cesta k souboru s trénovacími vektory.
- `POCET_UZLU` – určení počtu uzlů v chromozomu, minimální hodnota je jedna.
- `POCET_CHROMOZOMU` – určení počtu chromozomů v populaci, minimální hodnota je dva.
- `ODCHYLKA` – povolená odchylka, o kolik se může lišit nalezená hodnota od hodnoty požadované.
- `MAX_MUTACI` – maximální počet mutací, který je proveden při generování potomků, minimální hodnota je jedna.
  - Počet mutací je náhodná hodnota mezi jedničkou a zadanou hodnotou.
- `POSLEDNI_GENERACE` – maximální generace, které algoritmus může dosáhnout.
- `TOLERANCE` – procentuální určení minimální nutné fitness hodnoty pro nalezení výsledku. Zadávána jako desetinné číslo z rozsahu $\langle 0; 1 \rangle$.
- `CETNOST_ZALOHOVANI` – nastavuje pravidelnost zálohování v počtu generací.
- `VYBER_RODICE` – přepínač pro změnu algoritmu výběru rodiče, který upravuje jestli je použit další parametr po porovnání fitness hodnot, nebo ne.
  - 1 – výběr jen podle fitness hodnoty.
  - 2 – výběr podle fitness hodnoty a následně podle počtu uzlů ve fenotypu.
  - 3 – výběr podle fitness hodnoty a následně podle délky výsledné funkce.

- `TISK_PRUBEHU_HLEDANI` – přepínač pro změnu výpisu do souboru o průběhu hledání funkce.
  - 0 – vypnutí průběžného výpisu do souboru.
  - 1 – výpis průběžné fitness hodnoty rodiče, každý běh na jeden řádek.
  - 2 – podrobný výpis průběhu, jsou zde informace o fitness hodnotě, délce funkce, nebo počtu uzlů a kam je aktuálně připojen primární výstup.
- `CETNOST_TISKU` – nastavuje pravidelnost tisku průběhu hledání v počtu generací.
- `SOUBOR_PRO_TISK_PRUBEHU` – název souboru pro tisk průběhu hledání. Je ukládán do vytvořené složky s názvem použitého programu. Při podrobném výpisu je automaticky generováno číslování za název.
- `TISK_NALEZENE_FUNKCE` – přepínač výpisu nalezené funkce po dokončení programu na výstup. 0 tisk nalezené funkce vypíná, 1 zapíná.
- `POCET_UCICICH` (pouze cgpcl) – určuje počet uzlů, použitých pro učení. Maximální hodnota je `POCET_UZLU` − 1.

Dále je ještě možné využít parametru při spuštění programu. Parametr `-clean` způsobí, že se program chová, jako by byl minulý běh úspěšný a ignoruje zálohovanou konfiguraci.

4.2 Načítání souboru s trénovacími vektory

Pro bezchybné načtení trénovacích vektorů je třeba definovat strukturu souboru, ze kterého budou tato data načítána. Použito bylo řešení, ve kterém je první řádek ignorován (možnost pro uživatele popsat získaná data, popřípadě pro jeho další poznámky) a následně je každý řádek považován za jeden trénovací vektor. Tyto vektory jsou načítány, dokud se nenarazí na konec souboru. Struktura trénovacího vektoru byla zvolena jako nezávislá proměnná, za ní zapsána jedna závislá proměnná a mezi těmito položkami je nezbytné umístění alespoň jednoho bílého znaku. Na daném řádku je jakýkoliv další text za proměnnými ignorován. V případě, že načítané položky trénovacího vektoru nejsou číselnými hodnotami, načítání dalších vektorů je přerušeno, je vypsána se hláška o možné chybě ve vstupu a program se spustí s dosavadně načtenými hodnotami.

4.3 Generování náhodných čísel

Pro genetické algoritmy je velice důležité zvolit kvalitní algoritmus pro generování pseudonáhodných celých čísel, vzhledem k tomu, že je s nimi pracováno po celou dobu chodu programu. Bylo zde vybíráno mezi standardní funkcí z C++ `Rand()` a knihovnou *random*. Nakonec byl zvolen generátor náhodných čísel `mt19937` z knihovny *random*. Jedním důvodem byla schopnost u funkcí z knihovny random zaručit rovnoměrné rozložení dle našich potřeb. Oproti tomu u funkce `rand()` je pro získávání náhodných hodnot v zadaném rozsahu používána funkce modulo, eventuálně v kombinaci s posuvem přičtením. To zapříčiňuje, že rovnoměrné rozložení je zajistitelné jen u rozmezí, které mají počet hodnot o mocnině dvou. Toto je znatelné zejména při generování čísel ve větším rozmezí, protože na většině systémů `Rand()` může generovat číslo od 0 po 32767, při čemž s větším počtem čísel v rozložení je zvyšována jejich procentuální nerovnoměrnost.

Další důvodem výběru `mt19937` generátoru je podstatně delší doba, po které je znovu generována stejná posloupnost čísel. U `mt19937` to je $1-2^{19937}$ vygenerovaných čísel a u `Rand()`

je to ve většine případů 2^{32}. Tato skutečnost může být důležitá u dlouhých běhů s vysokou pravděpodobností mutace. Parametrické hodnoty u funkce `Rand()` se můžou lišit v závislosti na implementaci v systému, zmíněna zde byla nejčastěji implementovaná varianta.

Jako počáteční hodnota pro generátor `mt19937` je zvolena hodnota, kterou knihovna chrono vypočítá z aktuálního času. Tato hodnota je vypočítána získáním počtu uplynulých nejmenších časových jednotek podporovaných systémem (na většině systémů se pohybuje tato jednotka v řádech nanosekund, maximálně jednotek mikrosekund) od půlnoci dne 1. 1. 1970 časového pásma GMT. Díky rychlosti, jakou se tato hodnota mění, je velice nepravděpodobné, že dojde k vícenásobnému spuštění ve stejnou dobu.

4.4 Sběr dat a jejich záznam

Jako základní vlastnosti pro sběr byl zvolen čas potřebný pro běh programu spolu s počtem generací. Počet generací je v programu implementován jako proměnná a o měření času se zde stará funkce C++ `std::clock()`. Tato funkce byla zvolena z důvodu, že měří čas strávený během programu na procesoru a není závislá na platformě, na které je program spuštěn. Dále je uživateli umožněno nechat si na standardní výstup vytisknout předpis nalezené funkce společně s časem a generací ve, které to bylo řešení nalezeno. Pro vypsání nalezené funkce se používá in-order průchod získaným fenotypem.

Další možnost pro uživatele je nechat si ukládat data o průběhu běhů do souborů. V případě první implementace existuje jeden soubor s běhy, ve kterém je na jeden řádek pravidelně zapisována fitness hodnota. Interval zápisu fitness hodnoty je zadán uživatelem. Další běh programu je následně uložen na další řádek. V případě druhé implementace se jedná o podrobný výpis, ve kterém má každý běh svůj soubor, v němž jsou zaznamenávány informace o generaci, fitness hodnotě, délce fenotypu a aktuálním primárním výstupu rodiče.

4.5 Systém zálohování

Jelikož nelze zaručit, že program nebude z nějakého důvodu po dobu běhu ukončen, bylo třeba vytvoření možnosti průběžného zálohování. To je zde řešeno průběžným ukládáním dat potřebných pro chod programu do souboru. Četnost ukládání dat je zadávána v generacích jako parametr. Do souboru je ukládána doba běhu programu, generace, kolikátý běh programu je vykonáván a data chromozomu rodiče. Nevýhoda tohoto způsobu zálohování je, že existuje možnost ztráty prohledaných dat, která stihla vzniknout od doby poslední zálohy. Načtení ze zálohy probíhá tím způsobem, že pokud byl program přerušen, jsou načteny informace ohledně délky běhu a chromozom rodiče, z něhož se vygenerují potomci. Následuje pokračování programu stejným způsobem jako před přerušením.

4.6 Kompilace a spuštění

Pro překlad a spouštění programů byl využit program make a skripty napsané v bashi. `Make` přeloží všechny tří programy a vytvoří potřebné složky pro ukládání podrobných výsledků. `Make clean` vymaže všechny vytvořené složky, včetně vytvořených složek a jejich obsahu. Dále se zde nachází sada příkazu `make build_cgp`, `make build_cgpcl` a `make build_cgpcl_all`. Každý z těchto příkazů zkompiluje daný algoritmus.

Pro spouštění testů jsou vytvořeny typově dva skripty, `run_*.sh` a `tests_*.sh`, kdy za * je doplněn odpovídající algoritmus (`cgp`, `cgpcl`, nebo `cgpcl_all`). Skripty začínající na `run`, mají za úkol opakovaně spustit program dle nastavení počtu běhů. Skripty `tests` jsou schopné měnit parametr v průběhu běhu skriptu, a tak provést sadu testování nad jedním parametrem pro daný počet běhů pro každý parametr. Ukázkové nastavení je nachystáno ve skriptech, přičemž se dají parametry těchto skriptů změnit.

Kapitola 5

Experimentální vyhodnocení

V této kapitole jsou popsány jednotlivé experimenty, na kterých se úlohy testovaly, dále je zde zkoumáno chování algoritmů se souběžným učením a následně jsou zde tyto algoritmy porovnány s algoritmem vycházejícího ze standardního CGP (dále jen *cgp*). V této práci byly zkoumány dvě varianty CGP se souběžným učením. První varianta využívá algoritmus na bázi zkoumání několika uzlů (*cgpcl*) a varianta druhá využívá algoritmus na bázi zkoumání všech uzlů (*cgpcl_all*).

Veškeré experimenty byly spouštěny na školním unixovém serveru *edesign2*. Tento server obsahuje dva čtyřjádrové procesory Intel Xeon 5355 a má k dispozici 16GB RAM paměti. Operační systém na tomto serveru je CentOS 6.5 64bit Linux.

5.1 Úlohy použité pro experimenty

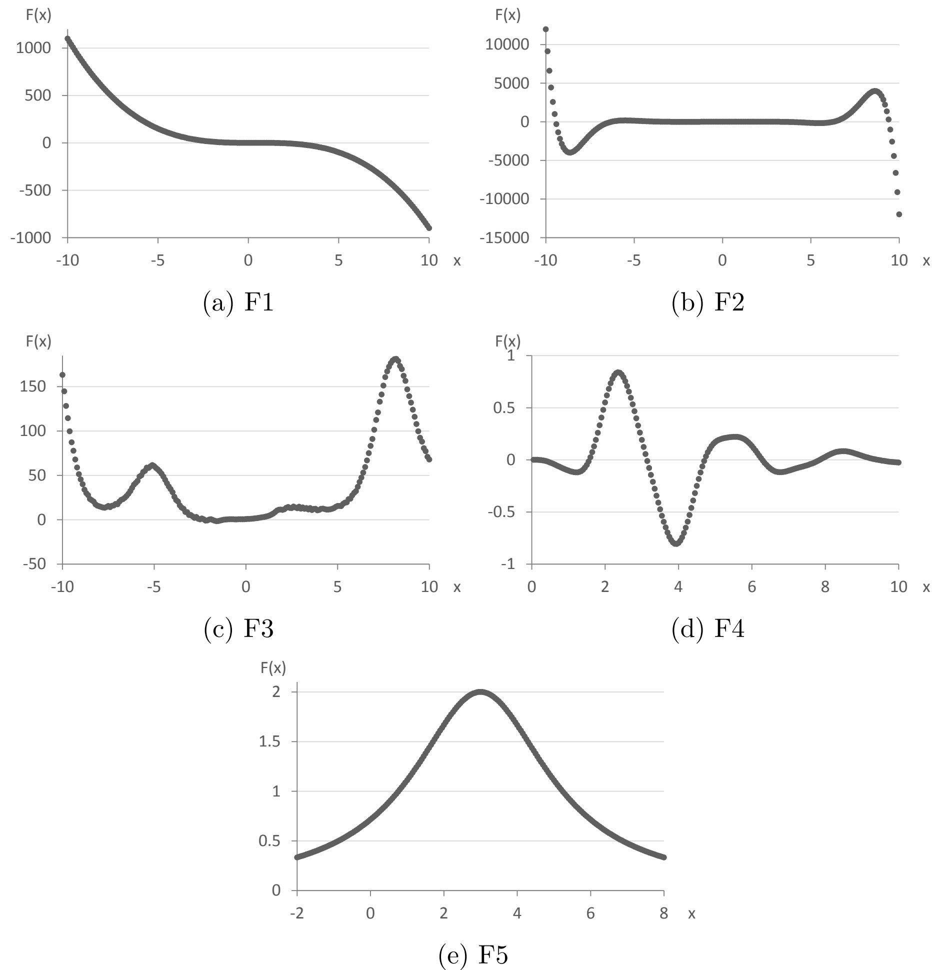
Pro provádění experimentů byla použita úloha symbolické regrese. Úlohy byly převzaty z článku [9], tato sada úloh (tabulka 5.1) obsahuje jak jednoduché úlohy pro CGP (F1, F2, F3), tak složité (F4, F5). Pro každou úlohu byla vytvořena sada s 201 trénovacími vektory z definičního intervalu. Trénovací vektory pro jednotlivé funkce z úloh lze vidět na obrázku 5.1.

Označení	**Funkční předpis**	**Definiční interval**
F1	$f(x) = x^2 - x^3$,	$x \in \langle -10; 10 \rangle$
F2	$f(x) = e^{\|x\|} \sin(x)$,	$x \in \langle -10; 10 \rangle$
F3	$f(x) = x^2 e^{\sin(x)} + x + \sin(\frac{\pi}{x^3})$,	$x \in \langle -10; 10 \rangle$
F4	$f(x) = e^{-x} x^3 \sin(x) \cos(x) (\sin^2(x) \cos(x) - 1)$,	$x \in \langle 0; 10 \rangle$
F5	$f(x) = \frac{10}{(x-3)^2+5}$,	$x \in \langle 2; 8 \rangle$

Tabulka 5.1: Seznam úloh pro jednotilvé experimenty.

5.2 Vlastnosti algoritmů se souběžným učením

Jako porovnávací úloha byla zvolena úloha F3. Zjišťování proběhlo nad parametry: počet chromozomů, počet uzlů, maximální počet mutací. Pro *cgpcl* byl zjišťován počet uzlů vhodných k aplikaci učení. Jako porovnávací parametr byl využit čas potřebný k nalezení řešení a pro každé nastavení byly algoritmy spuštěny 60krát.



(a) F1

(b) F2

(c) F3

(d) F4

(e) F5

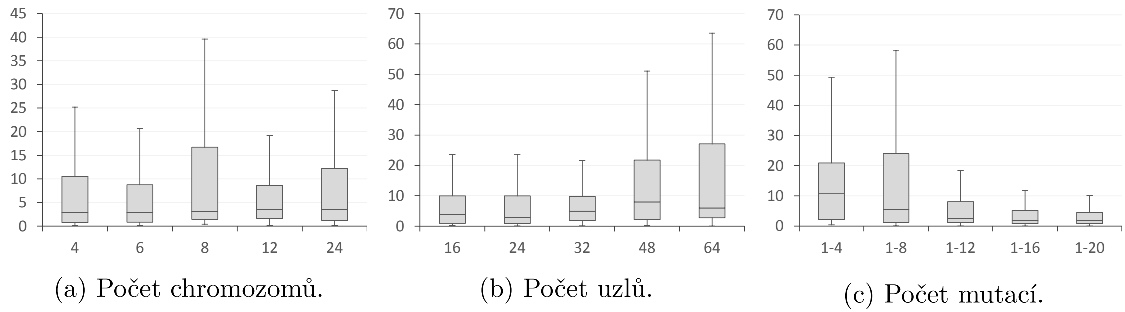
Obrázek 5.1: Rozložení trénovacích vektorů v jednotlivých úlohách.

Změna počtu uzlů a počtu chromozomů má u algoritmů se souběžným učením velice podobný efekt. V případě chromozomů je při vysokém počtu strávena delší doba prohledáváním užšího vygenerovaného prostoru možných řešení v jedné generaci, naopak při nízkém počtu chromozomů tento prostor nemusí být prohledán dostatečně.

U uzlů lze vidět, že při nižších počtech uzlů je třeba kratší časový úsek k vyhledání řešení. To se děje z důvodu, že je prohledáván menší prostor možných řešení a pro tuto úlohu existují hledaná řešení i v tomto prostoru. Od určitého bodu tato šance klesá kvůli narůstajícímu prostoru, kdy se snižuje poměr správných řešení vůči velikosti prostoru.

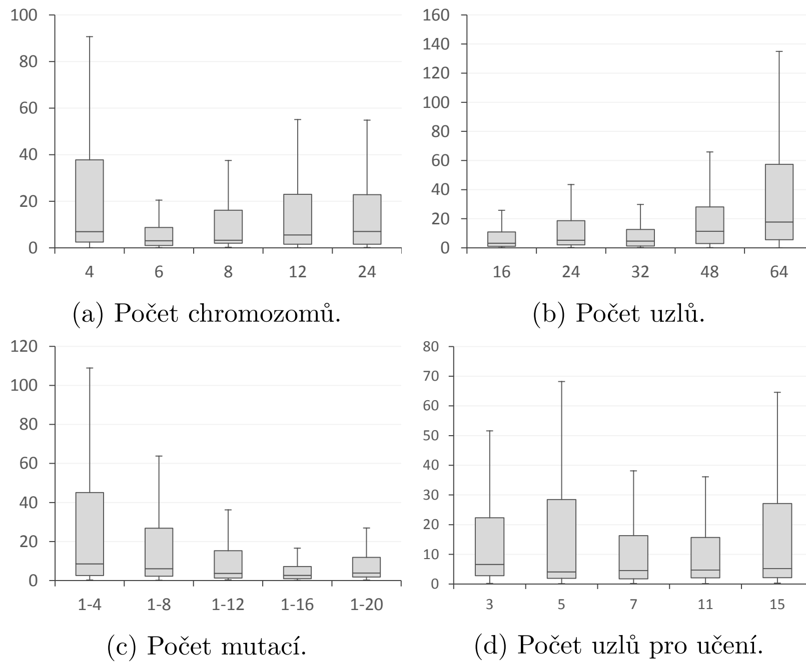
Z výsledků porovnání parametru mutace u *cgpcl* lze vidět, že tomuto algoritmu vyhovuje vyšší míra mutace. Při nízké mutaci může být souběžné učení na obtíž, kdy uzly, na kterých je prováděno učení, nemusí být od poslední generace změněny. Při příliš vysoké mutaci se začne algoritmus blížit náhodnému prohledávání, kdy evoluce začne mít nižší význam. Algoritmus *cgpcl_all* má ideální míru mutace položenou na ještě vyšší hodnotě, než cgpcl. To je zapříčiněno vyšším počtem uzlů, na kterých probíhá učení.

Pro učení u *cgpcl* lze pozorovat očekávané chování, kdy výhody učení stoupají s rostoucím počtem uzlů, na kterých je prováděno souběžné učení, a do jistého bodu následné začne doba strávená nad učením převyšovat čas, který může jedinci učení ušetřit.



(a) Počet chromozomů. (b) Počet uzlů. (c) Počet mutací.

Obrázek 5.2: Výsledky porovnání doby běhů parametrizace *cgpcl_all*. (s)



(a) Počet chromozomů. (b) Počet uzlů.

(c) Počet mutací. (d) Počet uzlů pro učení.

Obrázek 5.3: Výsledky porovnání doby běhů parametrizace *cgpcl*. (s)

5.3 Srovnání jednotlivých algoritmů

Parametry pro experimenty byly převzaty z článku [9]. Avšak oproti článku byla upravena hodnota mutovaných uzlů pro všechny algoritmy, to z důvodu, že tak nízká hodnota mutace znatelně zvyšuje čas pro získání řešení u souběžného učení. Dále byl vybrán počet uzlů pro aplikaci souběžného učení podle výsledků z grafu 5.3d. Pro úlohy F1, F2, F3 byl každý algoritmus spuštěn 100krát, pro úlohy F4 a F5 byl počet běhů stanoven z důvodu náročnosti výpočtu na 40. Přesné nastavení parametrů lze vidět v tabulce 5.2.

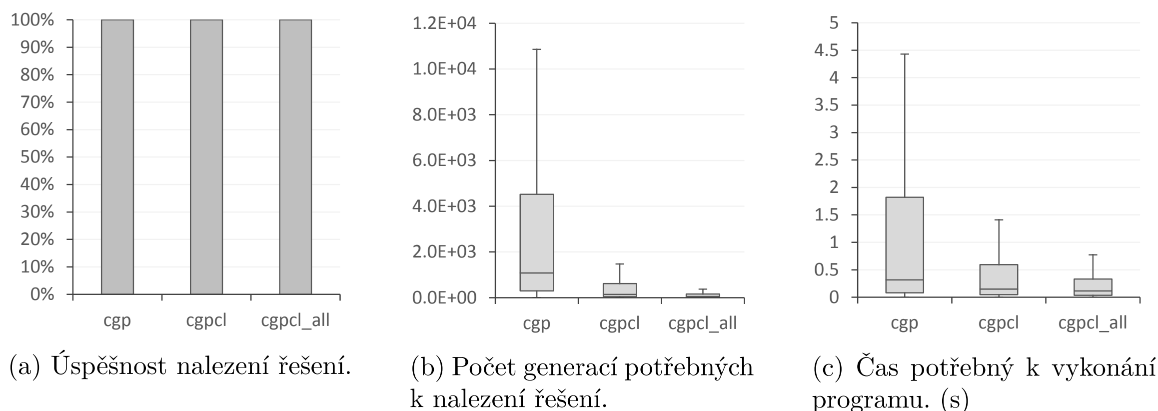
U experimentů byla zaznamenávána generace a čas, ve kterém bylo řešení nalezeno. Dále byla co 25 tisíc generací zaznamenávána aktuální hodnota fitness. Ve statistice o počtu generací jsou zaznamenány pouze úspěšné běhy, ve statistice o čase jsou zaznamenány běhy všechny. Záznam ze všech běhů u času byl vybrán z důvodu, že tímto způsobem je čas více vypovídající o reálné efektivitě algoritmu. Pro porovnání výkonnosti algoritmu byl vždy použit medián těchto hodnot.

Parametr	**Hodnota**
Počet jedinců v populaci	12
Počet uzlů v jedinci	32
Počet uzlů, na které se aplikuje souběžné učení (jen *cgpcl*)	7
Počet mutovaných genů	1–12
Množina funkcí uzlu	viz tabulka
Maximální počet generací	10 000 000
Minimální požadovaná hodnota fitness pro řešení	97% z maximální fitness
Povolená odchylka od trénovacího vektoru	F1, F2: 0,5; F3: 1,5; F4, F5: 0,025

Tabulka 5.2: Nastavení parametrů pro experimenty.

Úloha F1

S touto úlohou při daném nastavení parametrů neměl problém ani jeden z porovnávaných algoritmů. To lze vidět na grafu 5.4. Rozdíl je zde jen v počtu generací a času, které pro vyřešení této úlohy algoritmy potřebovaly. Počet generací byl 8krát nižší pro *cgpcl* a 20krát nižší pro *cgpcl_all*, při porovnání s algoritmem *cgp*. Oproti tomu časové zrychlení nebylo tak výrazné, algoritmus *cgpcl* byl rychlejší jen 3krát a algoritmus *cgpcl_all* 6krát.



(a) Úspěšnost nalezení řešení.

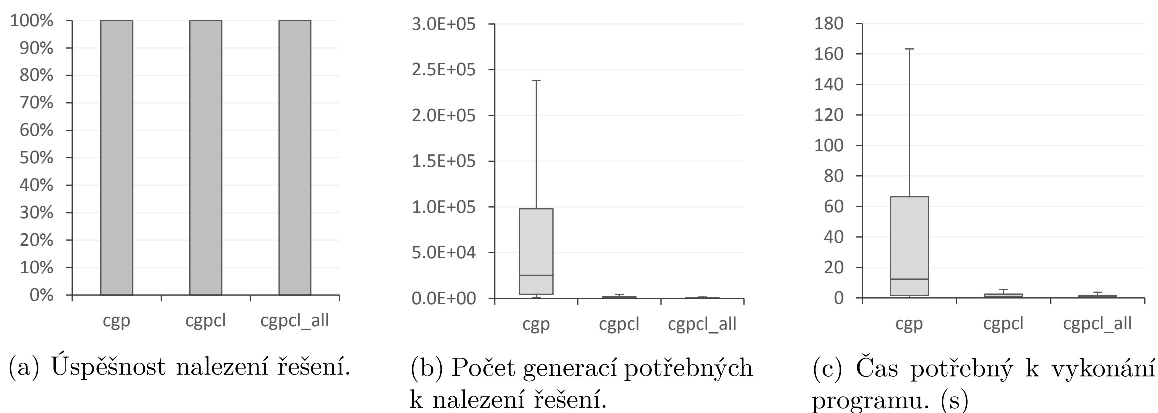
(b) Počet generací potřebných k nalezení řešení.

(c) Čas potřebný k vykonání programu. (s)

Obrázek 5.4: Porovnání jednotlivých algoritmů pro úlohu F1.

Úloha F2

V této úloze je oproti úloze F1 zrychlení podstatně výraznější. To lze vidět i na grafech 5.5. Počet generací potřebných pro získání řešení byl u *cgpcl* 30krát menší a u *cgpcl_all* bylo potřeba 70krát méně generací. Zrychlení bylo shodně u *cgpcl* a *cgpcl_all* 15násobné. Na této uloze lze vidět, že akcelerace vyhledání řešení je výraznější, pokud úloha není triviální. Dále se zde ukazují vyšší výpočetní nároky na generaci u algoritmu *cgpcl_all*, kdy sice tento algoritmus na rozdíl od *cgpcl* potřeboval pro nalezení řešení o více než polovinu méně generací, avšak čas pro nalezení řešení měly oba algoritmy skoro stejný.



(a) Úspěšnost nalezení řešení.

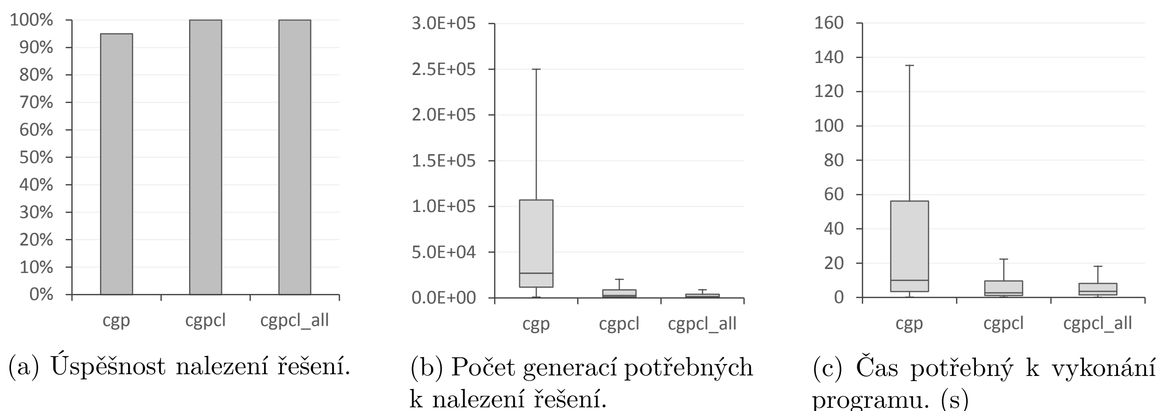
(b) Počet generací potřebných k nalezení řešení.

(c) Čas potřebný k vykonání programu. (s)

Obrázek 5.5: Porovnání jednotlivých algoritmů pro úlohu F2.

Úloha F3

Na této úloze lze vidět jeden z prvních nedostatků algoritmu *cgp*, konkrétně tu, že velice lehce uvázne v lokálním extrému. U tohoto se tento jev objevil ve dvou případech. Na grafech 5.6 lze vidět, že souběžné učení má stále pozitivní efekt na CGP. Oba algoritmy souběžného učení zde opět přinášejí vylepšení, kdy snížení počtu generací pro *cgpcl* je 10násobné a pro *cgpcl_all* je 17násobné. Zkrácení doby výpočtu je pro *cgpcl* a *cgpcl_all* zhruba 4násobné.



(a) Úspěšnost nalezení řešení.

(b) Počet generací potřebných k nalezení řešení.

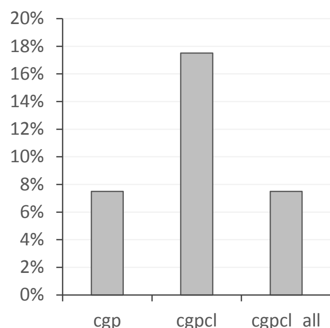
(c) Čas potřebný k vykonání programu. (s)

Obrázek 5.6: Porovnání jednotlivých algoritmů pro úlohu F3.

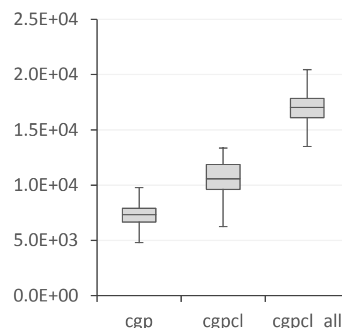
Úloha F4

Na grafech 5.7 můžeme vidět, že všechny porovnávané algoritmy měly s touto úlohou problémy. Algoritmy *cgp* a *cgpcl_all* úspěšně nalezly řešení ve 3 případech ze 40, algoritmus *cgpcl* našel řešení v 7 případech. U této úlohy měly všechny algoritmy tendenci uvíznout v lokálním extrému. Graf 5.7c vypovídá o tom, jak daný algoritmus umí řešit úlohu. I když by se z úspěšnosti algoritmů mohlo zdát, že *cgpcl* by na této úloze mohl být efektivnější, graf 5.7c ukazuje, že je to pouze statistická odchylka, protože má jen nepatrně vyšší průměrnou fitness hodnotu oproti algoritmu *cgpcl_all*. V případě, že by tento algoritmus byl lepší, měl by průměrnou fitness hodnotu během běhu blíže maximální hodnotě fitness.

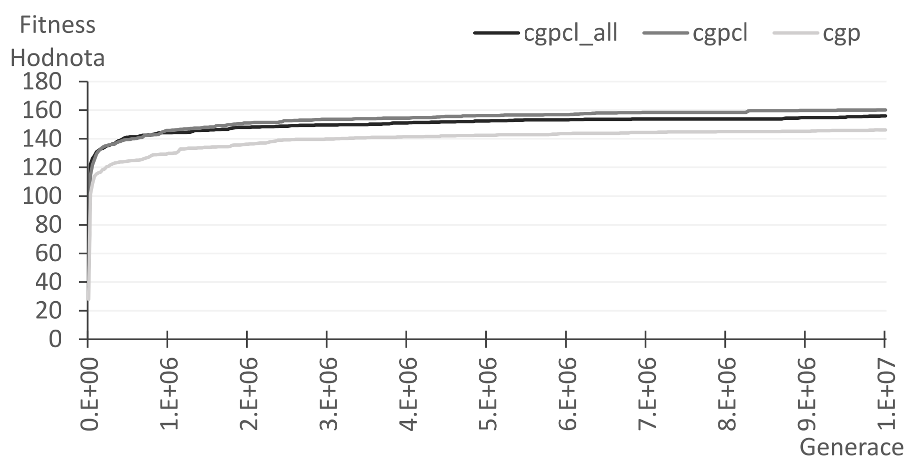
Dále na této úloze můžeme vidět časovou náročnost učení. Kdy *cgpcl* běželo 1,5krát déle a u *cgpcl_all* mělo zvýšenou dobu běhu 2,2násobně oproti *cgp*. Tyto hodnoty jsou tak vysoké z důvodu obtížnosti úlohy, kdy se algoritmy většinou dostaly na maximální počet generací a čas je zaznamenáván i z neúspěšných běhů.



(a) Úspěšnost nalezení řešení.



(b) Čas potřebný k vykonání programu. (s)

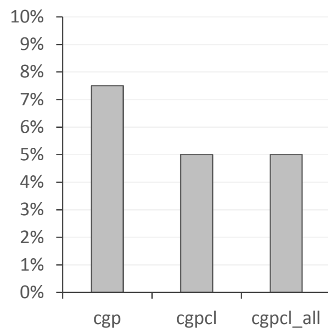


(c) Průměrná hodnota fitness v průběhu evoluce ze všech běhů.

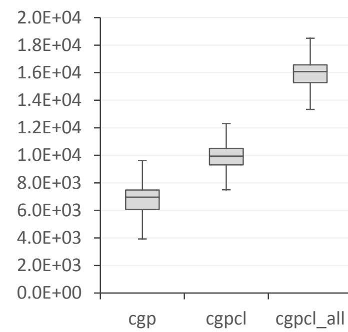
Obrázek 5.7: Porovnání jednotlivých algoritmů pro úlohu F4.

Úloha F5

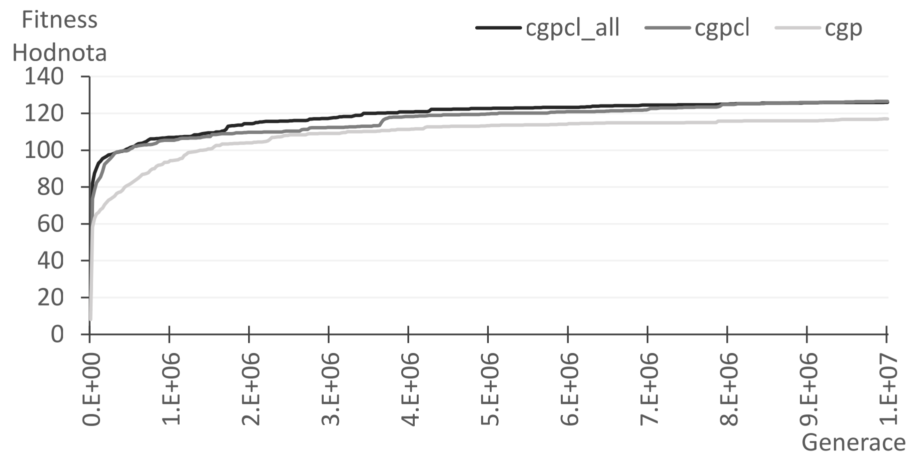
U úlohy F5 se projevil stejný problém jako u úlohy F4, kdy je úspěšnost nalezení řešení opět velice nízká. V grafu 5.8c můžeme vidět, jak algoritmy souběžným učením mají na začátku rychlejší růst fitness hodnoty oproti algoritmu *cgp*. Následně však opět všechny algoritmy uvíznou v lokálním extrému. Na grafech 5.8 lze vidět podobné výsledky jak u úlohy F4. Pro algoritmus *cgp* bylo řešení úspěšně nalezeno ve 3 případech ze 40, u algoritmů se souběžným učením to bylo v případech dvou.



(a) Úspěšnost nalezení řešení.



(b) Čas potřebný k vykonání programu. (s)



(c) Průměrná hodnota fitness v průběhu evoluce ze všech běhů.

Obrázek 5.8: Porovnání jednotlivých algoritmů pro úlohu F5.

5.4 Shrnutí výsledků

Při nastavení parametrů podle standardního CGP u úloh F1, F2 a F3 můžeme vidět výrazné vylepšení času pro nalezení řešení při použití CGP se souběžným učením, které je až 15krát rychlejší. Na těchto úlohách bylo potvrzeno, že i když výpočet pří CGP se souběžným učení každé generaci trvá déle, než u algoritmů bez souběžného učení, tak tato metoda dokáže výrazně urychlit vyhledání řešení. V čem ale CGP se souběžným učením má problém, stejně jako standardní algoritmus CGP, je špatná schopnost dostávat se z lokálních extrémů. Tato vlastnost je potvrzena v úlohách F4 a F5, kdy úspěšnost nalezení řešení je u všech algoritmů velice nízká.

Kapitola 6

Závěr

Tato práce se zabývá evolučními algoritmy, kartézským genetickým programováním a použitím souběžného učení na kartézském genetickém programování. Hlavním cílem zde byl návrh a implementace algoritmu se souběžným učením pro řešení úloh symbolické regrese. Navrhnuty byly dva algoritmy využívající souběžné učení. Jeden využívající učení pouze na vybraných uzlech a druhý využívající učení na všech uzlech chromozomu. Pro implementaci algoritmů byl zvolen jazyk C++.

Testování probíhalo na sadě 5 úloh symbolické regrese, přičemž parametry byly u všech algoritmů nastaveny na stejné hodnoty. Tato testovací sada prokázala, že pozitivní vlastnosti souběžného učení mohou překonat ty negativní. Na úlohách F1, F2, F3 byla významně urychlena evoluce nalezení řešení. Navržené algoritmy se souběžným učením byly až 15krát rychlejší v těchto úlohách než algoritmus bez souběžného učení. U těchto úloh se také projevila různá výpočetní náročnost algoritmů, kdy oba algoritmy, využívající souběžné učení, dosahovaly podobných časů potřebných k získání řešení, ale algoritmus, využívající učení jen na vybraných uzlech, stihnul vypočítat více generaci.

Avšak na úlohách F4 a F5 se projevil u algoritmů se souběžným učením stejný problém, jakým se potýká algoritmus bez něj. Konkrétně pro složité úlohy mají tyto algoritmy problém dostat se z lokálního extrému. Souběžné učení sice urychlilo růst fitness hodnoty na počátku, ale od určitého bodu se přes lokální extrém dostávalo velice špatně.

Během práce na projektu jsem získal nové znalosti ohledně soft-computingu, zejména části věnující se evolučním algoritmům. Při implementaci jsem si prohloubil znalosti ohledně některých knihoven C++. Dalšími novými zkušenostmi, spojenými s touto prací, pro mě bylo psaní vlastních skriptů v jazyce bash a používání sázecího programu LaTeX.

Literatura

[1] Baldwin, J. M.: A new factor in evolution. *The american naturalist*, ročník 30, č. 354, 1896: s. 441–451, ISSN 0003-0147.

[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, ISBN 978-1-59593-697-4.

[3] Ellefsen, K. O.: Balancing the Costs and Benefits of Learning Ability. In *Advances in Artificial Life, ECAL*, ročník 12, MIT Press, 2013, s. 292–299, ISBN 978-0-262-31709-2.

[4] Khatir, M.; Jahangir, A. H.; Beigy, H.: Investigating the Baldwin effect on Cartesian Genetic Programming efficiency. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, IEEE, 2008, s. 2360–2364, ISBN 978-1-4244-1822-0.

[5] Koza, J. R.: *Genetic programming: on the programming of computers by means of natural selection*, ročník 1. MIT press, 1992, ISBN 0-262-11170-5.

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

[7] Miller, J. F.; Thomson, P.: Cartesian genetic programming. In *Genetic Programming*, Springer, 2000, s. 121–132, ISBN 978-3-540-67339-2.

[8] Sekanina, L.: *Evoluční hardware: od automatického generování patentovatelných invencí k sebemodifikujícím se strojům.* Praha: Academia, 2009, ISBN 978-80-200-1729-1.

[9] Šikulová, M.; Sekanina, L.: Coevolution in cartesian genetic programming. In *Genetic Programming*, Springer, 2012, s. 182–193, ISBN 978-3-642-29138-8.

[10] Spears, W. M.; De Jong, K. A.; Bäck, T.; aj.: An overview of evolutionary computation. In *Machine Learning: ECML-93*, Springer, 1993, s. 442–459, ISBN 978-3-540-56602-1.

[11] Turney, P.: How to shift bias: Lessons from the Baldwin effect. *Evolutionary Computation*, ročník 4, č. 3, 1997: s. 271–295, ISSN 1063-6560.

[12] Ullah, F.; Khan, G. M.; Mahmud, S. A.: Exploiting developmental plasticity in Cartesian Genetic Programming. In *Computers & Informatics (ISCI), 2012 IEEE Symposium on*, IEEE, 2012, s. 180–184, ISBN 978-1-4673-1685-9.

[13] Vanneschi, L.; Poli, R.: Genetic Programming—introduction, applications, theory and open issues. In *Handbook of natural computing*, Springer, 2012, s. 709–739, ISBN 978-3-540-92909-3.