UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2020 Petr Kopic

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Hluboké genetické programování

Petr Kopic

Diplomová práce

2020

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Petr Kopic**
Osobní číslo:	**I17207**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Hluboké genetické programování**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem práce bude zmapovat a otestovat možnosti hlubokého učení v genetickém programování. V teoretické části student popíše ideje evolučních algoritmů, genetického programování, hlubokého učení a nalezne (případně navrhne) některé možné paralely v oblasti genetického programování, v části praktické provede experimenty a vyhodnotí jejich výsledky.

Rozsah pracovní zprávy: **50-60**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

HYNEK, Josef. Genetické algoritmy a genetické programování. Praha: Grada, 2008. Průvodce (Grada). ISBN 978-80-247-2695-3.
MITCHELL, Melanie. An introduction to genetic algorithms. Cambridge, Mass.: MIT Press, c1996. ISBN 978-0262133166.
POLI, Riccardo, W. B. LANGDON, Nicholas F. MCPHEE a John R. KOZA. A field guide to genetic programming. [S.l.: Lulu Press], 2008. ISBN 978-1409200734.

Vedoucí diplomové práce: **Ing. Jan Merta**
Katedra řízení procesů

Datum zadání diplomové práce: **5. listopadu 2019**
Termín odevzdání diplomové práce: **15. května 2020**

Ing. Zdeněk Němec, Ph.D.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2019

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne:

Petr Kopic

PODĚKOVÁNÍ

Mé poděkování patří panu Ing. Janu Mertovi za odborné vedení práce, cenné rady, trpělivost a ochotu, kterou mi při zpracování diplomové práce věnoval.

ANOTACE

Diplomová práce se zaměřuje na hledání nových možností při řešení úloh pomocí hlubokého genetického programování. Práce popisuje stručný vývoj umělé inteligence a hlavní metody, které se využívají – neuronové sítě, evoluční algoritmy a genetické programování. Dále se práce zabývá tématem hlubokého genetického programování, které se snaží kombinovat principy již zmíněných metod. Jsou popsány dosavadní přístupy a dále navrhnuty nové metody hlubokého genetického programování. Závěrem práce je popsána implementace metod pomocí různých nástrojů a jejich otestování.

KLÍČOVÁ SLOVA

Umělá inteligence, strojové učení, hluboké učení, neuronové sítě, evoluční algoritmy, genetické programování, hluboké genetické programování, jenetics, watchmaker

TITLE

Deep Genetic Programming

ANNOTATION

The Diploma Thesis is focused on the search on new possibilities in solving problems using deep genetic programming. The work describes a brief development of artificial intelligence and the main used methods like neural networks, evolutionary algorithms and genetic programming. Furthermore, the work describe the topic of deep genetic programming, which tries to merge the principles of the methods mentioned so far. Current approaches are described and new methods of deep genetic programming are proposed. The conclusion of the work describes the implementation of new methods using various tools and their testing.

KEYWORDS

Artifical inteligence, machine learning, deep learning, neural network, evolutionary algorithm, genetic programming deep genetics programming, jenetics, watchmaker

OBSAH

SEZNAM ILUSTRACÍ

SEZNAM TABULEK

SEZNAM ROVNIC

SEZNAM ZKRATEK A ZNAČEK

ČR	Česká republika
AI	Artifical Inteligence (umělá inteligence)
ANN	Artifical Neural Network
RNN	Redundant neural network
CNN	Convolution neural network
FFX	Fast Function Extraction
EA	Evoluční algoritmus
GA	Genetický algoritmus
GP	Genetické programování
SR	Symbolická regrese

0 ÚVOD

Tato práce se snaží shrnout a zmapovat možnosti vývoje algoritmů umělé inteligence jak již z teoretické stránky, tak z praktického pohledu. Jsou zde shrnuty možné přístupy k využití těchto algoritmů z hlediska použitelnosti nebo kvality v různých typech úloh. Hlavní cíl práce je zaměřen na rozšíření, nebo v některých případech spíše sloučení, klasických přístupů. Díky tomu by se mohly objevit nové přístupy nebo metody, které budou řešit optimalizační problémy. Pozornost je věnována hlavně hlubokému genetickému programování, což je přístup, který se snaží využít více přístupů k řešení úloh a vhodně je sloučit dohromady.

V principech hlubokého genetického programování lze spatřit přístupy podobné hlavně neuronovým sítím. Rozdílem ovšem je, že jsou aplikovány na problémy řešené pomocí evolučních algoritmů, a to zejména pomocí genetického programování. Práce tedy nabízí základní pohled na obor umělé inteligence, který je následně rozšiřován díky specifikování jednotlivých oblastí tohoto oboru. V první řadě se jedná o popis neuronových sítí, které jsou základním vzorem pro hluboké genetické programování. Jsou zde popsány jak jednoduché sítě, tak následně i hluboké. Hluboké neuronové sítě využívají proces *feature extraction*, který se snaží rozčlenit data do vrstev, a tím zjednodušit proces výpočtu. Přesně tento přístup se snaží napodobit i hluboké genetické programování.

Dále práce popisuje evoluční algoritmy, jež jsou velmi oblíbené a účinné pro velkou řadu problémů řešených pomocí AI. Do této skupiny patří i genetické programování, které je v práci rozšířeno do vrstev. Přidáním vrstev, kde se upravují vstupní data, získáváme model hlubokého genetického programování, které je popsáno v další kapitole.

V kapitole hluboké genetické programování jsou popsány některé z dosavadních existujících metod, které jsou považovány za hluboké. Jejich kombinacemi mohou vzniknout několikavrstvé modely, které by mohly pomoct zlepšení genetického programování. V kapitole je také navržena metoda, která některé tyto přístupy kombinuje.

V posledních kapitolách je popsána implementace jednoduchého genetického programování za použití dvou různých knihoven a následně implementace nově navržené metody. Následuje experimentální část, kde jsou tyto přístupy porovnány na různých pokusech, kde jsou měněna počáteční nastavení algoritmů. Výsledkem je zhodnocení kvality navržené metody oproti stávajícím přístupům v genetickém programování.

1 ZÁKLADY UMĚLÉ INTELIGENCE

Tato kapitola popisuje vývoj a základní principy umělé inteligence (AI). Jsou zde popsány důležité podmnožiny AI, jako je strojové a hluboké učení, které se v posledních letech velmi rychle rozvíjí díky velkému pokroku v technice. Dnešní počítače dokáží pomocí algoritmů AI řešit složité a komplexní problémy. Tato úvodní kapitola se tedy také věnuje popisu základních metod reprezentace dat, díky které mohou být algoritmy AI mnohem efektivnější a dále jednotlivým způsobům učení algoritmů, které je typickou vlastností strojového učení a jeho podmnožin.

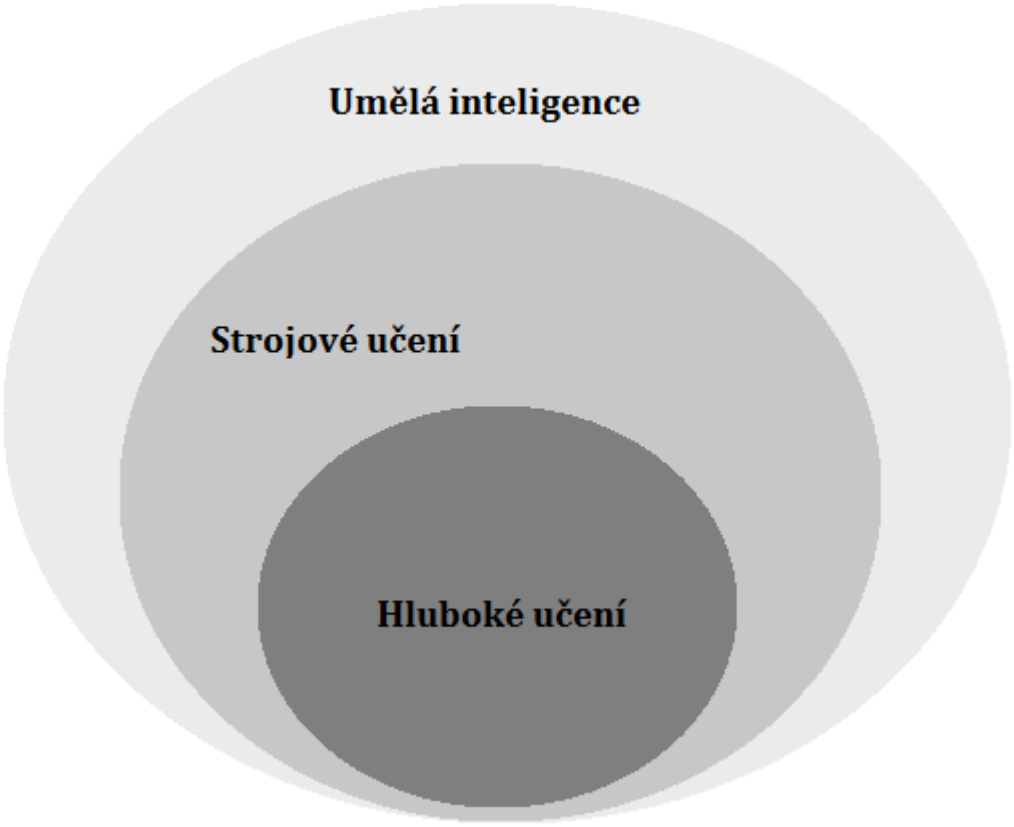
1.1 Umělá inteligence

Pod slovem inteligence si můžeme představit vlastnost živého organismu, která vznikla postupným vývojem. Inteligence je určena pro získání výhody při dosažení cílů živých organismů a umožňuje jim reagovat na měnící se prostředí a životní podmínky. Některé tyto principy jsou využity i v oboru umělé inteligence.[6]

Umělá inteligence je vědní obor, který se zabývá tvorbou strojů nebo systémů vykazujících lidskou inteligenci. AI se tedy snaží vytvořit stroj nebo systém, který k řešení problému využívá postupy, které by využil i člověk.[5] Tento vědní obor existuje již od roku 1956 a vznikl za účelem zjištění, zda lze naučit počítač myslet. Touto otázkou se AI zabývá dodnes. Stručně lze říci, že se AI snaží automatizovat intelektuální úlohy, které vytvářejí lidé.[4]

Jako každý vědní obor se i AI pořád vyvíjí. Do konce osmdesátých let byla hlavní myšlenkou AI tzv. symbolická umělá inteligence. Programátoři se domnívali, že mohou dosáhnout plného nahrazení lidského myšlení pomocí velkého souboru explicitních pravidel, čímž vznikaly expertní systémy. Systém s definovanými pravidly byl dobrým nástrojem pro řešení jasně definovaných logických problémů, jako jsou například šachy, ale byl již překonán novějšími technologiemi. Pro úlohy s nejasnými problémy, jako je například klasifikace obrazu, není možné vymyslet všechna pravidla. Jelikož postupem času vznikala potřeba řešit pomocí expertních systémů úlohy, jejich složitost se stupňovala, bylo jasné, že zhotovit tak velký soubor pravidel by bylo skoro nemožné. Právě proto začaly vznikat koncepty moderní AI, jejíž algoritmy umožní tyto úlohy řešit bez nutnosti vytvářet soubory pravidel.[4]

Pro řešení složitějších problémů bylo tedy nutné využít jiných přístupů, které jsou součástí AI. Dva hlavní směry, které budou popsány, jsou strojové a hluboké učení. Rozdíl těchto metod oproti tvoření expertních systému je, že nyní se programátor snaží naučit stroj tak, aby při pohledu na již známá data vytvořil pravidla, která nadále využije k rozhodování.[4]



Obrázek 1 - Umělá inteligence, strojové a hluboké učení

1.2 Strojové učení

Jak je vidět z obrázku umělá inteligence, strojové a hluboké učení, tak strojové učení je podmnožinou AI. Na rozdíl od expertních systémů, kde programátor zadával všechna pravidla ručně, se strojové učení zaměřuje na možnost vytvoření pravidel při pohledu na data. Systém využívající strojové učení by měl při znalosti vstupů a očekávaných výstupů nalézt taková pravidla, podle kterých bude schopen, při předložení určitého vstupu, vyřešit problém a vrátit očekávaný výsledek. Pro snadnější pochopení si můžeme představit příklad, kdy je nutné rozřadit obrázky do kategorií *auto* a *dům*. Systému předložíme několik obrázků, které máme rozřazeny správně, a tím ho naučíme rozpoznat kategorie. Když následně systému předložíme obrázek, měl by být schopný ho zařadit do správné kategorie *auto* nebo *dům*.[7]

Průkopníkem tohoto odvětví se stal již v roce 1959 Arthur Samuel, ale atraktivním se strojové učení stalo až po roce 1990. To bylo způsobeno hlavně technickým pokrokem. V těchto letech již byla přístupná technika, na které se algoritmy využívající principy strojového učení daly realizovat a výsledky byly dostupné v relativně přijatelném čase. S přicházející novou technikou se strojové učení těšilo větší oblibě a i nyní je velmi populární.[4]

Pokud je k řešení nějakého problému využit některý z algoritmů AI, není se možné spolehnout na stoprocentní správnost výsledku. K hledání řešení je totiž ve většině případů využito náhody a statistiky, což nemůže zaručit stoprocentní správnost nalezeného řešení. V těchto případech také velmi záleží na složitosti problému. Pokud je využit algoritmus správným způsobem, mělo by dosažené řešení být dostatečně věrohodné, nebo alespoň dobrým vodítkem k nalezení toho správného řešení.[8]

V této době se strojové učení vyskytuje hlavně v odvětvích herního průmyslu a dále je využito u systémů pro rozpoznání řeči a analýzu nebo kompresi obrazu. Většina lidí některé algoritmy využívá, aniž by o tom věděla, a to na některých internetových stránkách. V následujících desetiletích se ovšem předpovídá velký rozmach AI i do míst, kde využití techniky není hlavním zájmem. Může se jednat například o podporu rozhodování, ať už jako strategické rozhodování ve firmách nebo autonomní řízení automobilů.[7]

Oblast strojového učení obsahuje velkou řadu metodik a přístupů k tvorbě různých druhů algoritmů, které jsou dále vhodné pro řešení specifických druhů problémů. Jedno z nejdůležitějších rozdělení těchto algoritmů, je dělení podle druhu učení, kterému se věnuje jedna z následujících kapitol. Další rozsáhlou a důležitou podmnožinou strojového učení je hluboké učení, které je v dnešní době velice populární a velmi výhodné pro řešení některých problémů. Problematika hlubokého učení bude podrobně popsána v následujících kapitolách.[4]

1.2.1 Rozdělení algoritmů podle druhu učení

Jak již bylo řečeno, základní myšlenkou strojového učení je naučení systému, aby dokázal řešit konkrétní úlohu. Strojové učení nabízí několik způsobů, jak daný systém naučit a programátor si může zvolit nejvhodnější metodu pro úkol, který řeší. Volba učící metody je závislá na podobě dostupných dat a na úkolu, který je nutné řešit. Rozlišujeme učení s učitelem, bez učitele a zpětnovazebné učení. Každá z těchto metod má své využití a je vhodná pro specifické úlohy.[8]

Učení bez učitele znamená, že programátor má k dispozici jenom množinu vstupních dat. Proces naučení algoritmu se tedy musí uskutečnit bez množiny očekávaných vstupů. Nejčastějším případem učení bez učitele je shlukování objektů na základě nějakých podobností.[9]

Pravděpodobně nejpoužívanějším metodou je učení s učitelem. Tento model zahrnuje jak vstupní data, tak očekávané výstupy. Pro každou vstupní hodnotu algoritmus vrátí aktuální

výsledek, který je porovnán s očekávaným výstupem. Na základě tohoto rozdílu jsou upraveny parametry výpočtu algoritmu tak, aby byl kvalitnější. Je také vhodné rozložit množinu vstupních dat minimálně na dvě skupiny a to trénovací a testovací množinu dat. Trénovací množina dat je mnohem větší a algoritmus se na ní učí. Menší, testovací množina je použita pro kontrolu kvality algoritmu a pro zabránění přeučení.[4] Přeučení je jev, který vzniká v případě, že výsledky algoritmu jsou na trénovací množině čím dál lepší, zato na testovací množině se začnou zhoršovat. Z tohoto stavu lze vydedukovat, že jsou špatně zvoleny množiny dat, které nejsou z celého možného rozsahu dat.[19] Proto by měly množiny být disjunktní a jejich vstupy by měly být rozprostřeny po celém definičním oboru problému. Nikdy by tedy nemělo dojít k učení algoritmu na testovacích datech.[7]

Metoda zpětnovazebního učení spočívá v tom, že se algoritmus sám zdokonaluje za běhu programu. Podněty pro zdokonalení přichází většinou z okolí.[7]

Pomocí metod strojového učení lze řešit široká škála problémů. Tyto problémy lze shrnout do tří hlavních skupin, a to regrese, klasifikace a shlukování.

- Regrese

Podle vstupu odhaduje přibližnou hodnotu výstupu. Regresi je možné využít například v případě, že z náhodných bodů na číselné ose potřebujeme odhadnout předpis jejich funkce. V takovém případě použijeme známé body jako jejich vstupy a pomocí algoritmů AI zkoušíme najít nejpravděpodobnější předpis funkce. Využití je například pro predikce cen na burze.[8]

- Klasifikace

Jednoduchý příklad klasifikace může být rozdělení obrázků do různých kategorií. Jako vstupní data musí být několik příkladů správně rozřazených obrázků, na kterých se algoritmus naučí rozeznat jednotlivé kategorie. Výstupem může být například název kategorie.[8]

- Shlukování

Na základě podobnosti jsou shlukovány datové body do skupin, bez ohledu na obsah.[8]

1.2.2 Reprezentace dat

K sestavení algoritmu, který bude využívat strojové učení, je nutné určit tři základní součásti algoritmu.

- Vstupní data
- Očekávané výstupy

- Metoda, která změří kvalitu algoritmu

Žádný algoritmus se bez nich neobejde, jelikož jsou to základní součásti pro učení systému. Pro každý vstup musí existovat očekávaný výstup. Metodou, která měří kvalitu algoritmu, změříme rozdíl mezi aktuální a očekávanou hodnotou výstupu, čímž můžeme aktuální systém pozměnit (naučit) tak, aby byl příště tento rozdíl menší. Opakováním tohoto postupu můžeme docílit kvalitního systému, který bude správně naučený na konkrétní problém. Tento mechanismus bohužel nezaručí celý úspěch. Pro kvalitně fungující algoritmus je nutné se zamyslet i nad dalšími aspekty, jako je například reprezentace vstupních dat.[4]

Vhodná reprezentace dat vstupujících do jakéhokoliv algoritmu využívajícího AI je více než polovina úspěchu řešení daného problému. Tuto teorii je snadné ověřit jednoduchou dedukcí. Pokud jako vstup do algoritmu posíláme data, která nemají smysl pro řešení daného problému, tak nemůžeme dostat adekvátní výsledek. Navíc v tomto případě bude učení algoritmu trvat mnohem delší dobu, než kdybychom použili pouze adekvátní data.[4]

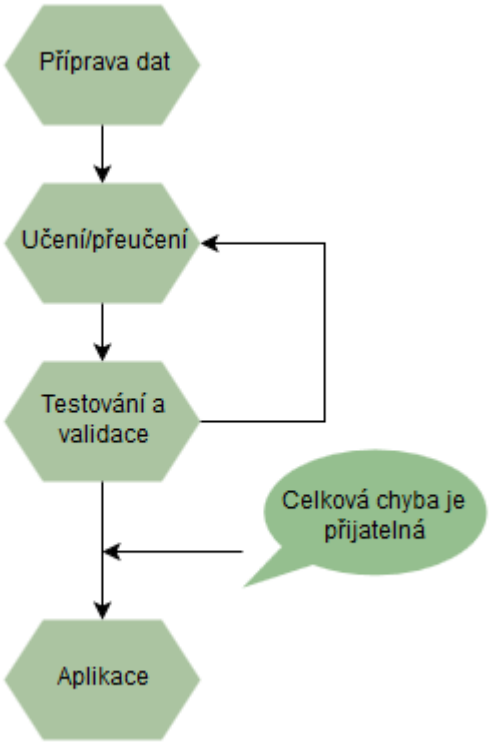
Nezáleží ovšem jen na očištění vstupních dat. Další důležitou technikou je vhodné zakódování dat. Tímto způsobem lze algoritmus také velmi zjednodušit. Jako příklad můžeme vzít jednoduchou klasifikaci obrazu. Pro některé problémy můžeme barevný obrázek uložený ve schématu RGB transformovat do odstínů šedi, čímž se sníží počet hodnot potřebných k identifikaci odstínu jednoho pixelu. U obrázku v RGB je nutné udržovat tři hodnoty, které určí barvu pixelu, u odstínů šedi tuto barvu určuje jediná hodnota. Tuto transformaci je vhodné použít v příkladech, kdy je účelem rozpoznat nějaký tvar, nebo obrys a není nutné brát v potaz jeho barvu.[4]

1.2.3 Fáze strojového učení

Fáze strojového učení lze chápat jako zjednodušený postup, jak vytvořit libovolný algoritmus, který bude využívat strojového učení. Základní fáze jsou naznačeny na obrázku 2. Základní a první fází je příprava vstupních dat. Tento krok je velmi důležitý, a pokud budou data dobře transformována pro daný problém, algoritmus se velmi zjednoduší.[8]

Dalším krokem je trénování. Systému jsou předložena vstupní data, a tím se realizuje učení algoritmu. Po předložení všech trénovacích vstupů je nutné zkontrolovat kvalitu algoritmu. K tomuto účelu je připravena testovací množina dat. Na těchto datech nemůžeme systém učit, aby nedošlo k naučení těchto konkrétních hodnot vstupů. Právě proto musí být trénovací a testovací množina disjunktní. Po testování se spočítá chyba modelu, která určuje, jaká je přesnost modelu na testovacích datech.[8]

Po testování je vhodné vypočítat celkovou chybu, a pokud je tato chyba menší než zvolená hodnota, která je blízká nule, je možné přejít na fázi aplikace. Je tedy možné aplikovat systém na ostrých datech a měl by být dostatečně spolehlivý. V opačném případě je vhodné systém začít učit od začátku, jelikož se v průběhu algoritmu nepodařilo nalézt dostatečně kvalitní řešení, které by bylo možné aplikovat. To znamená, že znovu aplikujeme fázi učení na trénovacích datech a následně testování a validaci. Tento cyklus se opakuje, dokud systém nevykazuje očekávané hodnoty na testovacích datech. Po nalezení adekvátního řešení je možné algoritmus ukončit.[8]



Obrázek 2 - Fáze strojového učení

1.3 Hluboké učení

Hluboké učení bylo poprvé představeno v roce 1986 Rinou Detcher jako podmnožina učení strojového. První experimenty ovšem nebyly efektivní, protože k řešení takto složitých modelů zatím neexistovala vhodná výpočetní technika. V dnešní době lze pomocí hlubokého učení řešit komplexnější a složitější úlohy než pomocí strojového učení. Tento obor se nyní těší velké oblibě a jeho techniky se nadále vyvíjí a zlepšují.[10]

Tato metoda je specifickou podmnožinou strojového učení. Cílem je dosáhnout funkcí lidského mozku pomocí výpočetní techniky. V lidském mozku je propojeno miliony neuronů,

které spolupracují na řešení určitého problému [11]. Systémy využívající hluboké učení používají podobné principy. Typickým algoritmem, který využívá hluboké učení je hluboká neuronová síť, na které budou zmíněné principy vysvětleny.[4]

Neuronová síť je složena z bloků, které se nazývají neurony. Neurony jsou slučovány do vrstev a tyto vrstvy dohromady tvoří model sítě. Počet vrstev udává hloubku modelu dané neuronové sítě. Takovýto model může obsahovat desítky až stovky vrstev a každá vrstva může obsahovat i tisíce neuronů. Velikost modelu závisí na obtížnosti řešené úlohy.[10] [12]

Princip hlubokého učení je v tom, že každá vrstva se učí z té následující, to znamená, že výstup jedné vrstvy je vstupem do vrstvy následující. Vstupní hodnota projde všechny vrstvy a daný model vrátí výslednou hodnotu. Následuje učení sítě, které probíhá od konce na začátek modelu (zpětné šíření chyby). Takto naučený model lze následně aplikovat na řešení konkrétního problému.[12] Při průchodu vstupní hodnoty modelem se každá vrstva modelu určitým způsobem podepíše na změně této hodnoty. Tímto způsobem dojde k transformaci vstupní hodnoty na výstupní. Poslední vrstva transformuje hodnotu na očekávaný výsledek modelu.[4][10]

Při tomto výkladu se může zdát, že se jedná o souhrn velmi složitých operací a transformací. To ovšem není pravda. Transformace se skládají z mnoha velmi jednoduchých operací s čísly. Jak již bylo vysvětleno, kvalita a výpočetní rychlost algoritmu také velmi hodně záleží na počáteční přípravě vstupních dat. Čím jednodušší budou vstupní data, tím jednodušší budou také operace při transformaci na výstupní hodnotu a tím bude učení daného algoritmu rychlejší a kvalitnější.[4][12]

1.4 Algoritmy využívající AI

Umělou inteligenci využívá velké množství algoritmů, které se dělí do několika skupin. Kromě toho, že je můžeme rozdělit podle typu učení, jako je strojové, hluboké, s učitelem nebo bez učitele, můžeme je také rozdělit podle způsobu řešení problémů. Každý algoritmus, nebo skupina algoritmů má své výhody při řešení specifických úloh.[4]

Jednou z prvních oblastí algoritmů, které se používaly, jsou expertní systémy. Tyto systémy jsou naprogramovány jako souhrn pravidel, podle kterých se program řídí a na jejichž základně se vyhodnotí výsledek. Tyto systémy jsou požívány jako podpora rozhodování.[13]

Další oblastí je prohledávání stavového prostoru. Tyto algoritmy se používají hlavně při řešení klasických her, jako jsou například šachy. Algoritmus má nadefinovanou množinu

stavů a množinu možných akcí neboli přípustných tahů. Cílem algoritmu je najít cestu mezi počátečním a koncovým stavem pomocí procházení stavů a nadefinovaným akcím.[13]

Početnou a oblíbenou skupinou jsou evoluční algoritmy. Tato oblast se snaží při řešení problémů používat přístupy vycházející z biologické evoluce. Mezi evoluční algoritmy patří například genetické algoritmy, mravenčí kolonie nebo genetické programování.[13]

Poslední představenou oblastí jsou neuronové sítě, kde se řešení úlohy snaží napodobit funkci lidského mozku. Nesnaží se ji však napodobit věrně, jelikož takový algoritmus by byl velmi složitý a ani v této době není zcela jasné, jak přesně mozek funguje.[13] Proces učení je zjednodušen tak, aby byl algoritmus rychlejší, ale model sítě byl obdobný jako u lidského myšlení. Neuronové sítě jsou v poslední době velmi populární oblastí AI, a to díky možnostem, které nabízí.[14] Pomocí hlubokých neuronových sítí lze řešit i velmi složité a komplexní úlohy, které mají využití například při rozpoznávání nebo kompresi obrazu, generování fotorealistických obličejů nebo scén pomocí GAN, deep fake výměny obličejů, nebo předvídání časových řad.[4]

2 UMĚLÉ NEURONOVÉ SÍTĚ

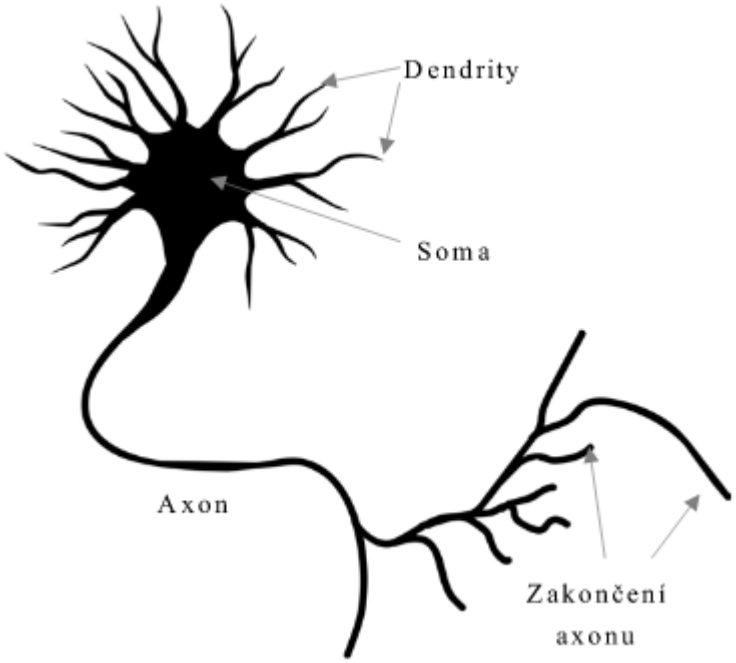
Neuronová síť je velmi komplexní nástroj pro řešení složitých úloh, které nejsou schopny vyřešit klasické algoritmy. Tato kapitola popisuje princip fungování neuronových sítí od jednoduchého perceptronu až po složité hluboké neuronové sítě.

2.1 Úvod do neuronových sítí

Umělá neuronová síť (ANN – Artifical neural network) je prostředkem především pro řešení úloh, kde je nutné zpracovávat komplexní data. ANN jsou inspirovány biologickými neuronovými sítěmi v lidském mozku.[14] Lidské myšlení pracuje na principu komunikace velkého množství malých jednoduchých výpočetních jednotek, které paralelně zpracovávají data. I umělé neuronové sítě se skládají z těchto malých jednotek, kterým se říká umělé neurony nebo perceptrony. ANN jsou používány hlavně v oblastech analýzy hlasu, obrazu nebo textu, dále v oblastech predikce časových řad a také v automatizaci výroby nebo diagnostice.[15]

2.1.1 Neuron

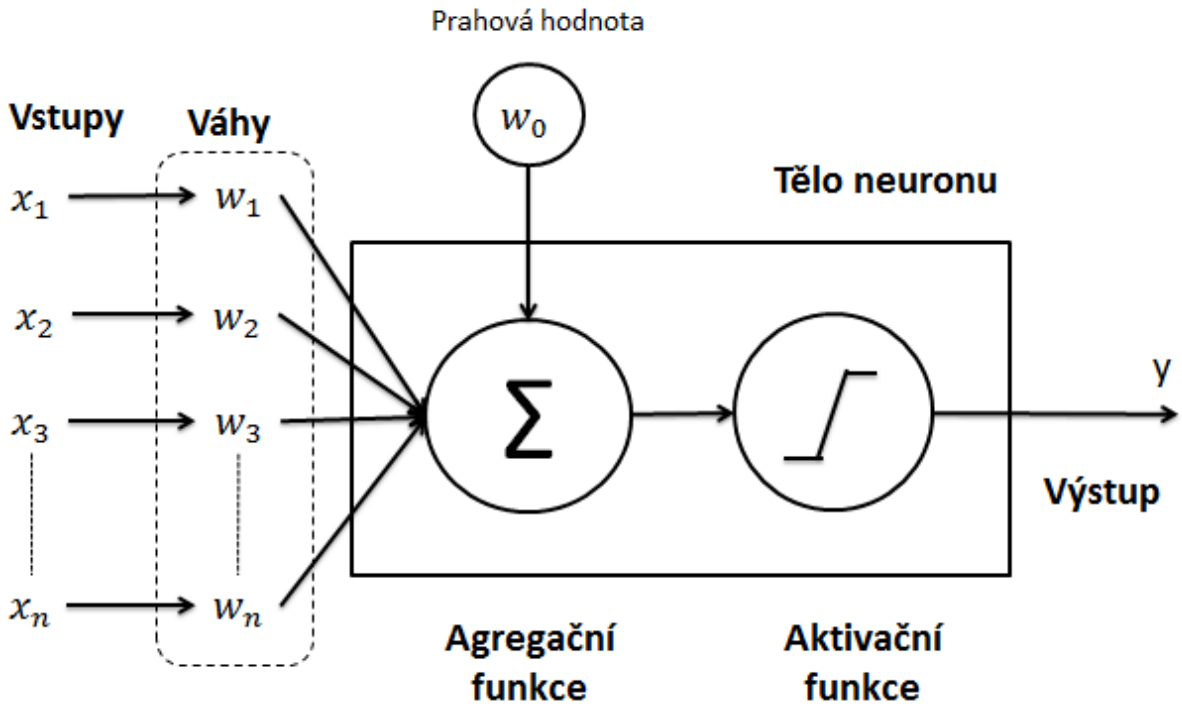
Biologický neuron, který je vyobrazený na obrázku 3, je buňka, která je potřebná k zajištění životních funkcí člověka. Jako každá buňka je i biologický neuron ohraničený, aby byl oddělený od okolí. Buněčné jádro se nazývá soma a je centrem celého neuronu. Z centra vychází dva druhy vláken, a to vstupní vlákna, dendrity, které vedou informaci do jádra neuronu a axon, který vede informaci z jádra. Vstupů do jádra neuronu neboli dendritů může mít neuron několik, zato výstup neboli axon má každý neuron pouze jeden. Umělý neuron napodobuje principy biologického neuronu, také se skládá z několika vstupů, těla a jednoho výstupu. Pomocí jednoho či více takových neuronů vzniká umělá neuronová síť.[20]



Obrázek 3 - Biologický neuron[20]

První umělý neuron se objevil v roce 1948 a vynalezli ho neurofyziolog Warren McCulloch a logik Walter Pits. Jedná se o jednoduchou výpočetní jednotku (matematickou funkci), která zpracovává vstupní hodnoty a transformuje je na výstupní. Nejjednodušší umělá neuronová síť je složena právě z jednoho nebo z několika nezávislých neuronů a nazývá se perceptron. ANN typu perceptron bude popsána v následující kapitole.[16]

Umělý neuron funguje na jednoduchém principu. Skládá se ze souboru vstupů, vah, prahové hodnoty, aktivační a agregační funkce a výstupní hodnoty. Tyto součásti dohromady tvoří celou buňku, tak jak je vidět na obrázku 4. První krok k získání výstupní hodnoty je výpočet agregační funkce, která sloučí vstupní hodnoty a váhy pomocí předem dané metody. Výsledek agregační funkce je dále porovnán s prahovou, neboli mezní hodnotou. Pokud hodnota agregační funkce překročí prahovou hodnotu, což může znamenat, že při hledání předpisu funkce je překročen definiční obor, nebo obor hodnot této funkce, je použita jako výsledek agregační funkce použita prahová hodnota. Dále je na tuto sumu aplikována aktivační funkce, a tím je získána výstupní hodnota z neuronu.[20]



Obrázek 4 - Umělý neuron, Zdroj: vlastní

Vstupní číselné hodnoty představují buď výstupní hodnoty z neuronů, které se nachází v předešlé vrstvě neuronové sítě, nebo to jsou hodnoty podnětů z vnějšího světa, které jsou vstupy do celého modelu ANN. Hodnoty, které mohou vstupy nabývat, jsou závislé na typu řešené úlohy.[17][20]

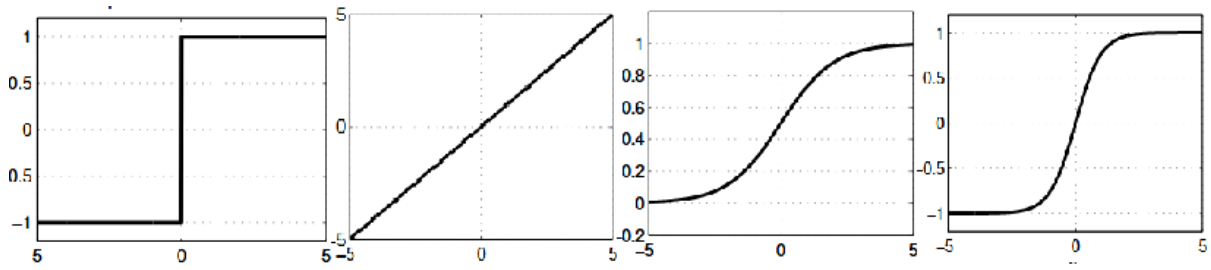
Váhy spojení nejvíce ovlivňují transformaci vstupních hodnot na výstup. Každý vstup je ohodnocen váhou, a tím je zaručeno, že se vstupní hodnota transformuje na požadovaný výstup. Váhy jsou nejdůležitějším parametrem při učení sítě, neboť to jsou hodnoty, které se v průběhu učení ladí, čímž se daná ANN zpřesňuje a vylepšuje svoje výsledky.[20]

Účelem agregační funkce je sloučit vstupy do neuronu do jedné hodnoty. Nejčastěji se používá suma vážených vstupů. Tato funkce vynásobí každý vstup příslušnou váhou a následně tyto hodnoty sečte. Výstup z agregační funkce pokračuje na vstup aktivační funkce.[20]

Prahovou hodnotu si lze představit jako bariéru, kterou vstupní hodnota neuronu nesmí překročit. Většinou se používá tak, aby vstupní hodnota nepřekročila obor hodnot aktivační funkce.[20]

Aktivační funkce převádí vstupní hodnotu na výstupní. K tomuto účelu je možné využít velké množství funkcí. Jedny z nejpoužívanějších jsou bipolární skoková, lineární, sigmoidální,

nebo hyperbolicko-tangenciální aktivační funkce. Každá funkce má své výhody a je vhodnější ji využít v určitých vrstvách neuronové sítě anebo může mít výhodné použití vzhledem k typu řešené úlohy.[17] [20]



Obrázek 5 - Bipolární skoková, lineární, sigmoidální a hyperbolicko-tangenciální aktivační funkce[20]

2.1.2 Trénování

Žádná neuronová síť, ani jiný algoritmus AI, není ihned po naprogramování připravena pro aplikaci na reálný příklad. Každá síť se musí nejprve natrénovat na řešení konkrétní úlohy.[14] Proces trénování, nebo učení sítě spočívá především v odladění vah tak, aby síť dokázala transformovat vstupní hodnoty na výstupní. Váhy mají podobu reálných čísel a podílí se na sloučení a transformaci vstupních hodnot v agregační funkci neuronu.[20]

K natrénování umělé neuronové sítě je nutné připravit vstupní množinu dat. Tato množina obsahuje vstupní hodnoty a k nim očekávaný výsledek. Vzhledem k povaze úlohy se tedy může jednat například o historická data vývoje trhu, obrázky správně rozřazené do kategorií, nebo například o souřadnice bodů v rovině nebo prostoru. Některá vstupní data je nutné před použitím upravit do číselné podoby, aby s nimi mohla síť pracovat, což platí zejména u obrázků, které je nutné převést do číselné podoby. Nebo u velmi komplexních dat, které je vhodné nejdříve očistit od nepotřebných parametrů.[18][20]

Připravená vstupní data je vhodné rozdělit do tří vzájemně disjunktních skupin. První skupinou je trénovací množina. Tato množina je nejpočetnější a ANN se na ní učí. Jelikož asi nikdy nebudeme mít k dispozici tak velkou trénovací množinu dat, síť se na těchto datech učí opakovaně a postupně vylepšuje své výsledky. Další množiny se nazývají validační a testovací. Tyto množiny slouží ke kontrole, jestli síť nekonverguje ke špatným výsledkům, proto je nutné, aby všechny množiny byly vzájemně disjunktní. Je tedy také vhodné, aby všechny množiny byly, co možná nejvíce rozprostřené po definičním oboru všech vstupních dat. Tím je možné zajistit, aby síť byla připravená pro všechny možné vstupní hodnoty, a to i na extrémní hodnoty.[20]

Pro natrénování sítě existuje velké množství algoritmů, z nichž každý je vhodný pro různé druhy neuronových sítí, úloh a podobě vstupních dat. První algoritmus, který vznikl a stal se základem pro mnoho dalších, je Hebbovo učení, které poprvé publikoval psychiatr Donald Hebb v roce 1949. Hebbův zákon učení říká, že pokud jsou dva propojené neurony aktivní, tak je jejich vazba, kterou reprezentuje váha, zesílena. Pokud jsou oba neurony neaktivní, tak je vazba mezi nimi zeslabena. Tento způsob učení je vhodný pro jednodušší ANN, jako je například perceptron. Složitější algoritmy, které vycházejí z Hebbova zákona učení jsou určeny spíše pro složitější modely umělých neuronových sítí. Mezi tyto učící algoritmy patří například zpětné šíření chyby.[20]

Na kvalitu, rychlost a účinnost učení neuronové sítě má velký vliv také topologie této sítě. Tento problém se týká sítí s více vrstvami a při hledání vhodné topologie se většinou používá forma pokusů. Při hledání vhodné topologie se hledá vhodný počet vrstev, počet neuronů v jednotlivých vrstvách a také vhodné aktivační funkce jednotlivých vrstev. Počet vrstev a neuronů v nich se většinou určuje metodou pokus, omyl. Většinou jsou prvně zvoleny náhodné hodnoty, které jsou následně upravovány dle výsledků učení. Podle pozorování je sestavena nejvhodnější topologie neuronové sítě. Pro výběr vhodné topologie sítě je také možné použít některý z dalších algoritmů strojového učení, nejčastěji některý ze skupiny evolučních algoritmů.[19][20]

2.2 Perceptron

Umělá neuronová síť typu jednoduchý perceptron se většinou skládá pouze z jednoho neuronu, ale může se stát, že je použito více nezávislých neuronů. Tento typ sítě se tedy hodí pouze pro úlohy, které jsou lineárně separovatelné. Takovým problémem může být například jednoduchá kategorizace do dvou skupin.[10] [17]

Jelikož se tato síť skládá pouze z jednoho neuronu, tak je topologie stejná jako u neuronu, tak jak je vidět na obrázku 4. Jako agregační funkce je zde většinou použita suma vážených vstupů a jako aktivační funkce se velmi často využívá funkce bipolární skoková, která rozděluje definiční obor úlohy na dvě skupiny. Díky této funkci se odezva přikloní buď k jedné, nebo ke druhé skupině, výsledek tedy může nabývat dvou hodnot (0, 1; -1, 1;…).[20]

Učení jednoduchého perceptronu lze provést pomocí Hebbova učení, nebo, v praxi více používaném, chybovém učení. Tento druh učení patří do kategorie učení s učitelem, kde je nutné používat vstupní data s očekávanými výstupy, podle kterých se ladí váhy na vstupech do perceptronu.[20] Váhy jsou aktualizovány po přiložení každého vzoru ze vstupních dat.

K aktualizaci vah je nutné vypočítat rozdíl mezi vzorem a odezvou perceptronu, který se spočítá odečtením očekávané hodnoty od aktuálně vypočítané. Přírůstky váhy *Δw* jsou potom přepočítány podle vzorce:[20]

Rovnice 1 - Výpočet přírůstku váhy perceptronu[20]

$$\Delta w_i = \alpha x_i e,$$

Kde α je koeficient rychlosti učení, který je v intervalu (0, 1) a v průběhu učení se většinou zmenšuje, čímž se postupem času zjemňuje přírůstek k váze. Hodnota x_i je vstupní hodnota do perceptronu a hodnota e je chyba na výstupu, která vzniká odečtením odezvy od očekávané hodnoty.[20]

Algoritmus učení je spuštěn na určitý počet epoch, což uvádí kolikrát je síť učena na všechny vzory z trénovací množiny dat. Postupem času je vhodné snižovat koeficient rychlosti učení, tak, aby při běhu posledních epoch nedocházelo k tak výrazným změnám vah. Po každé epoše je vhodné zkontrolovat průběh celkové chyby a otestovat síť na testovacích množinách, čímž můžeme ověřit správný vývoj učení sítě. V ideálním případě se celková chyba na testovací množině rovná nule (nebo alespoň hodně blíží). Takto natrénovaná síť je připravená na aplikaci do provozu.[10][17][19]

2.3 Vícevrstvá neuronová síť

Vícevrstvá neuronová síť přinesla do oboru umělé inteligence velmi výrazný pokrok. Tento druh sítě významně vylepšuje síť typu jednoduchý perceptron a umožňuje řešit mnohem složitější úlohy. Nejčastěji se využívá k řešení predikce, klasifikace, nebo aproximace funkcí.[17]

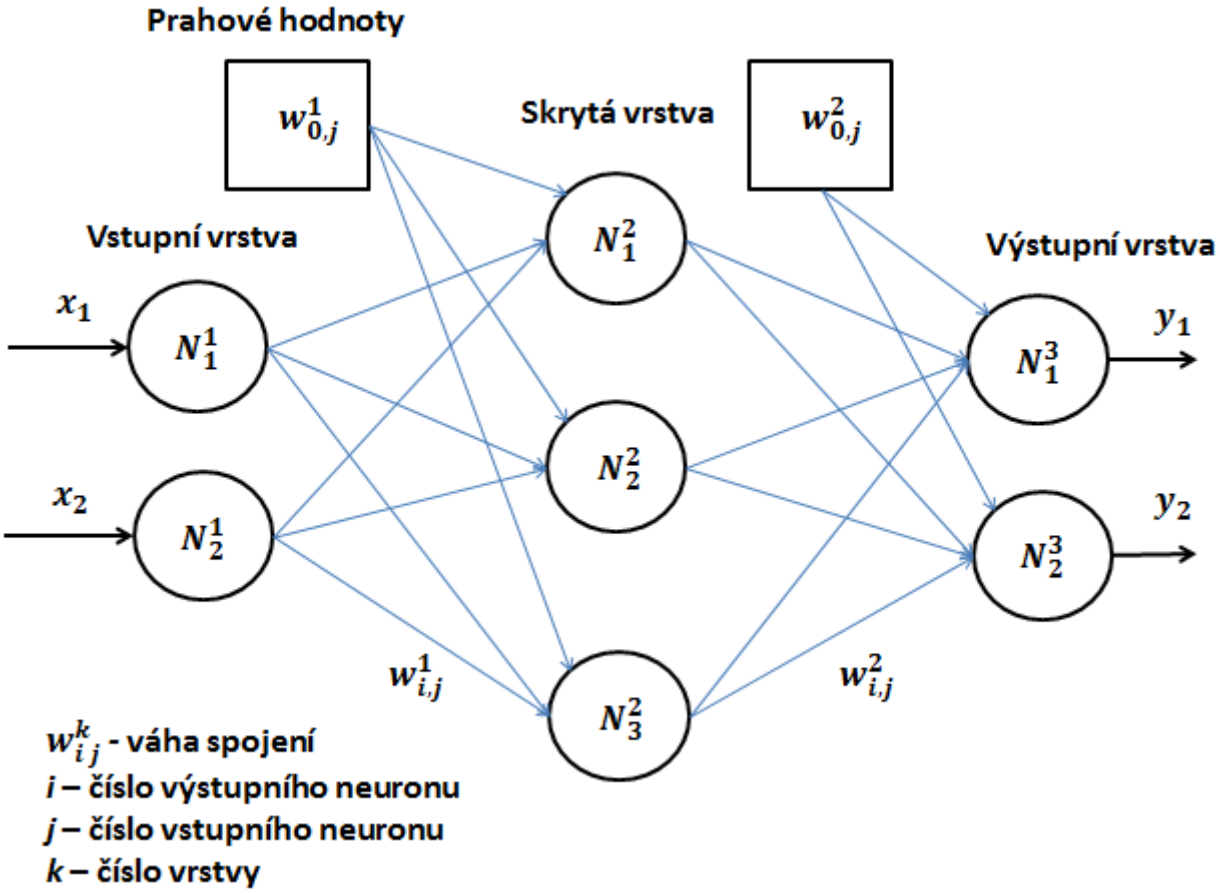
2.3.1 Model vícevrstvé sítě

Základními stavebními prvky sítě jsou neurony, které jsou poskládány do vrstev. Každá síť je složená z několika vrstev, které dělíme na tři druhy. První je vrstva vstupní, na kterou jsou přivedeny vstupy do sítě.[14] Cílem této vrstvy je rozvést každý vstup na všechny neurony v následující vrstvě. Tato vrstva tedy obsahuje takový počet neuronů, kolik existuje vstupů. Pokud jsou jako vstupy posílány trojrozměrné vektory, vstupní vrstva bude obsahovat právě tři neurony.[20]

Dalším vrstvám se říká skryté. Těchto vrstev může síť obsahovat více, typicky však vícevrstvá síť obsahuje jednu skrytou vrstvu. Účelem skrytých vrstev je naučit se transformovat vstupní data. K tomu jsou využity agregační a aktivační funkce neuronu. Jako

agregační funkce se většinou využívá suma vážených vstupů, což znamená, že ke každé skryté vrstvě musí existovat sada vah a prahových hodnot, podle kterých je možné vypočítat tuto agregační funkci. Jako aktivační funkce se ve skrytých vrstvách využívají většinou funkce sigmoidální nebo hyperbolický tangens. Ve většině případů se využívá stejná funkce pro všechny neurony v jedné vrstvě.[20]

Poslední vrstva se nazývá výstupní. Zde se provádí konečná transformace na očekávanou výstupní hodnotu. Tato vrstva také obsahuje sadu vah a prahových hodnot, tudíž je zde také využita agregační funkce suma vážených vstupních hodnot. Pro výstupní vrstvu je podle druhu úlohy zvolena aktivační funkce, například pro regresi je ideální použít lineární funkci. Počet neuronů ve výstupní vrstvě musí být shodný s počtem očekávaných výstupních hodnot. Pokud tedy očekáváme dvourozměrný vektor jako výstup, musí výstupní vrstva obsahovat právě dva neurony.[20]



Obrázek 6 - Model vícevrstvé neuronové sítě, zdroj: vlastní

2.3.2 Výpočet odezvy vícevrstvé sítě

Pro stanovení odezvy je potřeba připravit vstupní vektor dat, který je následně přiložen na vstupy *x* do neuronové sítě. Vstupní vektor dat reprezentuje vstupní hodnotu a je dobré ho připravit co nejjednodušší. Tím můžeme eliminovat zbytečné části vektoru a výpočet a učení bude značně rychlejší. Po přiložení vstupů se vypočítají vstupní hodnoty první skryté vrstvy pomocí agregační funkce a výstupní hodnoty pomocí aktivační funkce. Následně je výsledek rozeslán na vstupy další skryté vrstvy a postup se opakuje, dokud není dosaženo výstupní vrstvy. Z výstupní vrstvy je vytvořen vektor, který je označen jako potencionální řešení. Pomocí tohoto potencionálního řešení je v dalším kroku vypočítán rozdíl mezi skutečnou hodnotou a odezvou sítě. Pokud je tento rozdíl menší než zadaná malá hodnota, tak je síť prohlášena za naučenou a je možné ji používat v praxi. V opačném případě je odezva využita pro naučení sítě. Tento proces je popsán v následující kapitole.[10] [20]

Rovnice 2 - Výpočet agregační funkce neuronu[20]

$$y_{a,j}^{k} = \sum_i w_{i,j}^{k} * y_i^{k-1},$$

Rovnice 3 - Výpočet výsledku aktivační funkce neuronu[20]

$$y_j^k = \Phi^k(y_{a,j}^k),$$

kde

k – číslo vrstvy
i – číslo vstupního neuronu
j – číslo výstupního neuronu
Φ^k – aktivační funkce vrstvy k

2.3.3 Učení vícevrstvé sítě

Učení sítě probíhá jako učení s učitelem, je tedy nutné mít připravenou trénovací množinu dat, popřípadě ještě validační a testovací množiny, u kterých budou známé očekávané hodnoty.[17] Jak již bylo řečeno v předchozích kapitolách, je vhodné, aby všechny množiny obsahovaly data rovnoměrně rozesetá po celém definičním oboru vstupních dat, aby bylo možné síť natrénovat co možná nejlépe. Cílem trénování sítě je minimalizovat rozdíl mezi očekávanou a skutečnou odezvou sítě.[19] Tohoto cíle je dosaženo pomocí adaptace vah, která funguje na principu:

1. Zjištění skutečné odezvy sítě na určitý vstup,
2. Výpočtu rozdílu mezi skutečnou a očekávanou hodnotou,
3. Adaptace vah podle vzniklého rozdílu,
4. Přesunu na krok jedna s další vstupní hodnotou.

Problémem je, že ve vícevrstvé síti nejsou laděny pouze váhy na jednom neuronu, jako tomu bylo u jednoduchého perceptron. Zde je potřeba naladit váhy na několika neuronech a navíc ve více vrstvách. K tomuto účelu slouží hned několik poměrně složitých algoritmů, z nichž nejznámější je algoritmus zpětného šíření chyby (*backpropagation*).[10]

Pomocí tohoto algoritmu se vypočítají přírůstky vah na každém neuronu, které se následně přičtou k aktuálním vahám. Algoritmus zpětného šíření chyby byl pojmenován podle toho, že se přírůstky vah počítají směrem od výstupní ke vstupní vrstvě. K získání přírůstků vah je nutné vypočítat takzvané lokální gradienty, které se počítají z výsledků následující vrstvy. U výstupní vrstvy se lokální gradient vypočítá pomocí rozdílu mezi očekávanou a skutečnou hodnotou.[20] U skrytých vrstev ovšem není možné použít tento rozdíl, jelikož by jeho použití nedávalo smysl a síť by byla natrénovaná chybně. Z tohoto důvodu se k výpočtu používá gradient následující vrstvy. Po výpočtu gradientů všech vrstev je možné začít počítat přírůstky vah pro jednotlivé neurony a to směrem od vstupní vrstvy k výstupní.[10] [19]

U trénování se sleduje celková chyba, která by se měla postupem času snižovat a pokud je na konci trénování chyba blízká nule, tak je síť připravena na použití.[10]

2.4 Hluboká neuronová síť

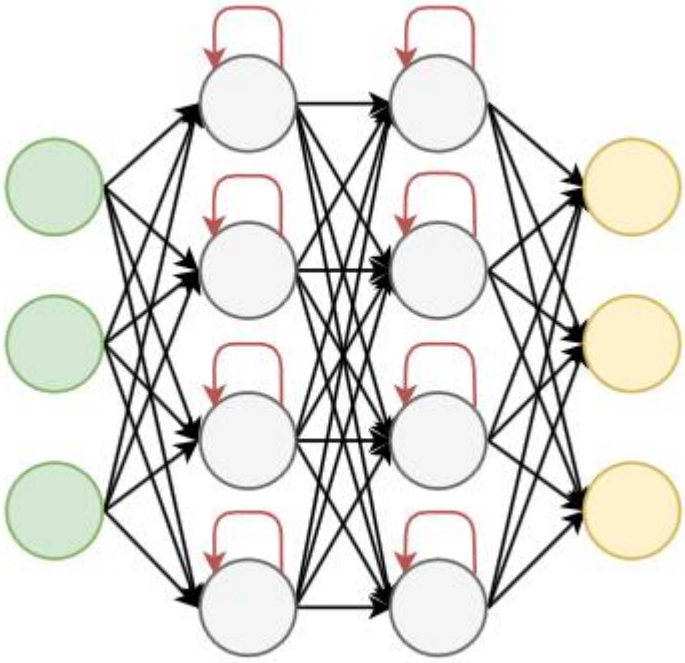
Hluboké neuronové sítě jsou dalším důležitým milníkem ve vývoji oboru umělé inteligence. Hlavně kvůli své náročnosti a požadavkům na rychlost hardwaru při velmi dlouhých procesech učení se dostaly oblíbenosti až v posledních letech. Pomocí těchto sítí lze popsat složité komplexní problémy postupně. Nejdříve jsou popsány jednoduše a postupně se tyto jednoduché příznaky kombinují v dalších vrstvách. Tímto způsobem lze jednodušeji modelovat složitější problémy.[17]

Obecně lze tedy hlubokou neuronovou síť popsat jako vícevrstvou síť s více jak jednou skrytou vrstvou. Dalším důležitou vlastností je takzvaná *feature extraction*. Tento pojem znamená, že síti nepředkládáme upravená data. Jako vstupy jsou předána surová, neopracovaná data, která obsahují nepodstatné informace pro proces učení a stanovení odezvy. Proces *feature extraction* tedy očistí data od nepodstatných informací a zároveň vybere ty, které jsou důležité. Tato metoda je realizovaná pomocí několika speciálních vrstev a další vrstvy slouží k samotnému stanovení odezvy stejně jako u vícevrstvé sítě. Model hluboké neuronové sítě může být tedy velmi složitý a může obsahovat i stovky vrstev. Z toho vyplývá, že proces učení může být velmi časově a výpočetně náročný. Hluboké neuronové sítě lze rozdělit do dvou hlavních skupin, a to na rekurentní a konvoluční sítě.[17]

2.4.1 Rekurentní neuronová síť

Všechny dosud popisované neuronové sítě, i konvoluční sítě, které budou popsány v následující kapitole, se nazývají dopředné. Dopředné, protože signál jimi putuje pouze jedním směrem, a to od vstupu na výstup. V těchto sítích tedy nemůže signál vstoupit do žádné smyčky ani se vrátit o několik vrstev zpět. Výjimkou jsou právě rekurentní sítě, kde se využívá takzvaná zpětnovazební smyčka. Díky využití smyčky se rekurentní sítě více podobají lidskému mozku, protože jim smyčka dodává zásadní vlastnost, a to je uchování informace, neboli paměť.[21] Díky této vlastnosti může rekurentní síť dosáhnout možná i lepších výsledků než síť dopředná, a to hlavně v oblastech sekvenčního zpracování nebo generování textu a také například při úpravě nebo doostření obrazu.[10] [21]

Problémem u těchto sítí je ovšem proces učení, kde je nutné vyřešit problém zpětnovazební smyčkou. Díky této smyčce je učení ještě složitější než u sítí dopředných, proto se rekurentní sítě tolik nevyužívají a existuje zatím málo přístupů, které by byly dostatečně efektivní.[21]



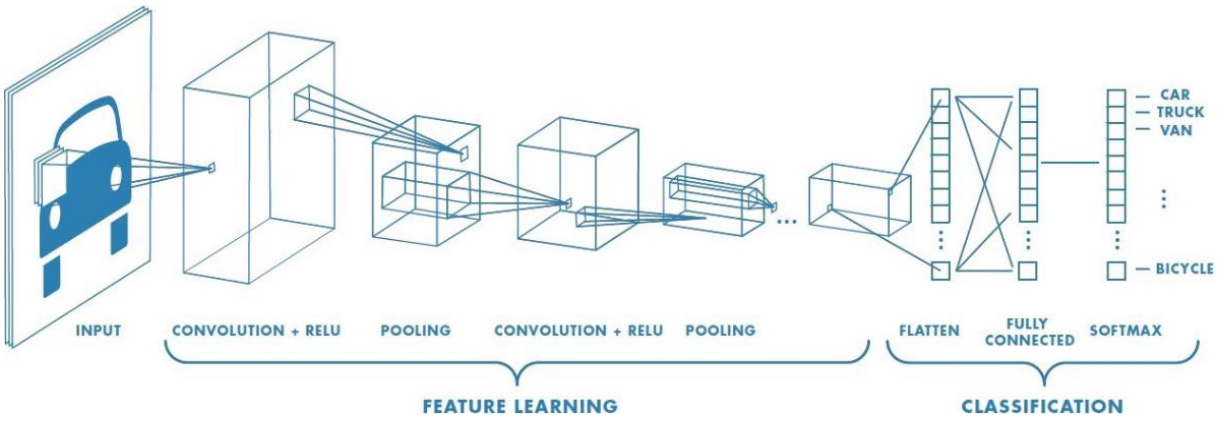
Obrázek 7- Rekurentní neuronová síť[10]

Rekurentní neuronová síť je složena z klasických neuronů. Neurony ve skrytých vrstvách navíc obsahují zpětnovazební smyčku, jak je vidět na obrázku 7. Díky smyčce si neurony nesou informaci ze všech předchozích vstupů. To je praktické pro vstupní data, která nemají konstantní délku, jako například text, různě velké obrázky, nebo také časově závislé údaje, které mohou disponovat různým množstvím parametrů.[22]

K učení rekurentní sítě se využívá zpětné šíření chyby, ovšem problém nastává při ladění vah zpětnovazební smyčky. Jelikož je toto spojení rekurzivní, nastává situace, kdy se gradient předchozí vrstvy opakovaně násobí rekurentní váhou. Z toho plyne, že pokud bude váha menší než jedna, laděná váha se bude pořád snižovat a blížit k nule a naopak pokud bude váha větší než jedna, bude při ladění nekontrolovatelně růst. Tomuto jevu se dá zabránit dvěma způsoby. Buď je ladění rekurentních vah vynecháno, tzv. *echo state*. Další možnost je pomocí metody *long short term memory*, kdy při učení neuron pracuje se svým stavem (pamětí) a rekurentní váhy jsou zafixovány na hodnotě jedna. Výpočet nového stavu neuronu je tedy spočítán pomocí nynějších i předchozích vstupů, předchozího stavu neuronu, prahů a vah.[22]

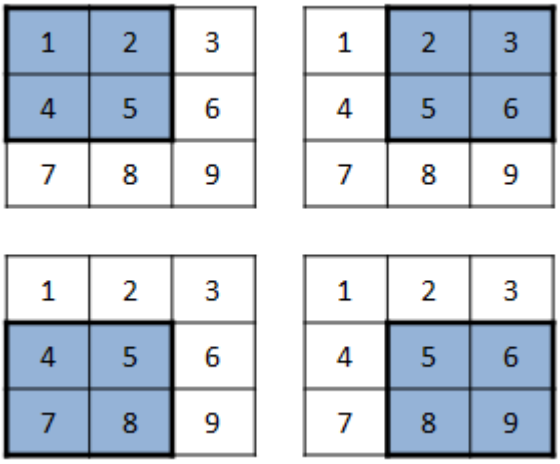
2.4.2 Konvoluční neuronová síť

Konvoluční síť (CNN) je hluboká neuronová síť, která kromě principů klasické vícevrstvé sítě obsahuje ještě několik dalších vrstev, které zajišťují proces *feature extraction(detekce příznaků)*. CNN tedy dokáží zpracovat předem neupravená a neočesaná vstupní data pomocí detekce příznaků a získat potřebná vstupní data například pro klasifikaci.[17] Jak je vidět z obrázku 8, CNN může být rozdělená na dvě části, a to právě část, která zajišťuje detekci příznaků (*feature extraction*, nebo *feature learning*), a část, která provádí konečné vyhodnocení, v tomto případě klasifikaci obrázku.[23]



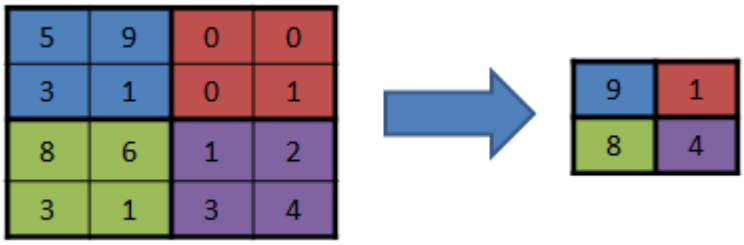
Obrázek 8 - Příklad CNN[23]

Proces detekce příznaků se skládá z konvolučních a pooling (shlukovacích) vrstev, kterých může síť obsahovat i více. Konvoluční vrstva slouží k rozložení vstupních dat na konvoluční filtry. V případě obrázku je možné si filtr představit jako menší část vstupního obrázku. Díky tomu se zmenší počet neuronů ve vstupní vrstvě a množství vah, čímž se zrychluje dopředné šíření i algoritmus učení (zpětné šíření). Při dopředném šíření jsou nejdříve vytvořeny všechny filtry, ze kterých jsou spočítány výsledné hodnoty vrstvy.[23][30]



Obrázek 9 - Příklad vytvoření konvolučního filtru při velikosti okna 2x2, zdroj: vlastní

Dále signály putují do pooling vrstvy, která slouží pro další snížení náročnosti výpočtu. Po výstupu z konvoluční vrstvy totiž pořád existují parametry, které jsou při hledání výsledku sítě nepodstatné. Pooling vrstvy se tedy snaží eliminovat nepodstatné parametry. Tento proces může být realizován například výběrem nejvyšší hodnoty ze zadaného výběru (max-pooling), nebo průměrnou hodnotou (average pooling), atd.[23][30]



Obrázek 10 - Příklad metody max-pooling, zdroj: vlastní

Po projití všech konvolučních a pooling vrstev je proces *feature extraction* u konce a jeho výsledky vstupují do druhé části CNN. Tato část se stará o získání požadovaného výsledku celé sítě. Ve většině případů se tato část dá přirovnat ke klasické vícevrstvé neuronové síti. Pomocí CNN sítí se dají řešit problémy klasifikace, segmentace obrazu, zpracování obrazu, atd. Jelikož CNN řešící tyto většinou poměrně složité úlohy, jejich topologie může nabývat velkých rozměrů. Proces učení může být tedy velmi složitý a zdlouhavý. Proto jsou pro různé takto složité problémy k dispozici různé, již předučené sítě, které je možné využít a „doučit“ podle potřeby. Příkladem takových sítí může být například AlexaNet.[23][30]

3 GENETICKÉ ALGORITMY

Velmi obsáhlou a oblíbenou skupinou algoritmů AI a strojového učení jsou genetické algoritmy. Přednost těchto algoritmů spočívá v jejich jednoduchosti, síle a prvku náhody, který je přítomný skoro ve všech fázích hledání optimálního řešení. Většina těchto algoritmů využívá jako svůj vzor principy evoluční biologie. Genetické algoritmy jsou založeny na principu evoluční biologie, přesněji na darwinovské evoluci. Tyto algoritmy se využívají k numerické optimalizaci a rozhodování, strojovému učení, učení konceptů z hlediska dolování dat, nebo k optimalizaci modelů neuronových sítí.

Kapitola popisuje základní principy průběhu genetického algoritmu, základní pojmy a využívané metody pro jednotlivé fáze hledání optimálního řešení daného problému.

3.1 Základní pojmy

K pochopení jednotlivých fází a principů genetických algoritmů je nutné zavést několik základních pojmů, které se objevují v každé fázi hledání řešení. Tyto pojmy představují základní kameny každého algoritmu a bez nich by evoluce nemohla proběhnout. Jedná se o jedince, neboli chromozom, následně gen, populaci, generaci a následně fitness funkci.

3.1.1 Jedinec (Chromozom)

Jedinec je nositelem genetické informace. Z hlediska genetických algoritmů jedinec obsahuje takzvaný chromozom, pod kterým si lze představit řešení daného problému. Toto řešení ovšem není konečné, ale je to pouze návrh na optimální řešení problému.[27]

Chromozom má ideálně podobu sekvence nul a jedniček. V tomto případě je hledání optimálního řešení nejjednodušší a nejrychlejší. Ne v každém případě lze hledané řešení zakódovat do této podoby a v tom případě je možné se setkat i s jinými reprezentacemi tohoto chromozomu, jako například celé nebo reálné číslo, matice, vektor, křivka, atd.[2]

Jedinec č. 1									
0	0	1	1	0	1	0	1	1	1
Jedinec č. 2									
396	45	75	87	555	789	533	1	14	43

Tabulka 1 - Příklady jedinců genetického algoritmu

3.1.2 Gen

Gen je nejmenší jednotka chromozomu. Pokud se tedy podíváme na příklady jedinců genetického algoritmu, tak jeden gen si lze představit jako jednu buňku tabulky. Tyto geny již nejdou v rámci algoritmu dále dělit a jako celek tvoří chromozom. Gen může nabývat různých hodnot, a to v závislosti na druhu řešené úlohy. Ze základních podob genů se nabízí uvést znak, číslo, nebo také specifičtější příklady jako barva, akce (dopředu, otočit, doprava, atd.) a další. Celý soubor hodnot, kterých může gen v řešené úloze nabývat, se nazývá alely.[2]

3.1.3 Populace a generace

Populace je skupina jedinců, kteří figurují v jedné generaci. Pokud tedy algoritmus pracuje s populací o velikosti sto jedinců, tak každá generace bude obsahovat sto, většinou rozdílných, jedinců. Dále tedy algoritmus pracuje s počtem generací, díky kterému lze po určité době ukončit běh algoritmu. Je možné na počátku algoritmu definovat spuštění evoluce na konkrétní počet generací. [1][27]

3.1.4 Fitness funkce

Zde se jedná nejspíše o nejdůležitější pojem. Fitness funkce je taková funkce, která dokáže ohodnotit každého jedince číslem, které vyjadřuje jeho kvalitu, a tím tedy i kvalitu řešení problému. Při ukončení algoritmu je tedy právě jedinec s nejlepším fitness ohodnocením prohlášen za hledané řešení daného problému.[27]

Ke každému problému je třeba sestavit odpovídající fitness funkci, která dokáže spravedlivě ohodnotit každého jedince, a tím získat spolehlivé řešení daného problému. Jako příklad lze uvést problém, kdy se snažíme dosáhnout situace, kdy bude jedinec obsahovat všechny geny rovné hodnotě jedna. Ohodnocení bude tedy vyjadřovat počet jedniček, které jedinec obsahuje.[27]

Jedinec										Fitness
0	0	1	1	0	1	0	1	1	1	6
1	1	0	0	1	1	0	0	1	0	5
0	0	0	0	1	1	1	0	0	0	3
1	1	1	1	1	0	1	1	1	0	8
1	1	1	1	1	1	1	1	1	1	10

Tabulka 2 - Příklad ohodnocení jedinců pomocí fitness funkce

3.2 Obecné principy

Průběh genetického algoritmu lze rozdělit do několika základních fází, které se podílí na hledání optimálního řešení problému. Tyto fáze většinou napodobují již zmíněnou darwinovskou evoluci, podle které se genetické algoritmy řídí. Výsledkem běhu tohoto algoritmu je optimální, nebo alespoň dostačující řešení daného problému. Při použití genetického algoritmu ovšem není nikdy zaručeno dosažení tohoto optimálního globálního řešení. Je velmi pravděpodobné, že algoritmus zkonvertuje k některému z lokálních, částečných řešení, ve kterém uvázne. V některých případech může být i toto částečné řešení považováno za dostačující, ale v případě opačném je nutné průběh algoritmu opakovat do nalezení řešení dostačujícího. [1]

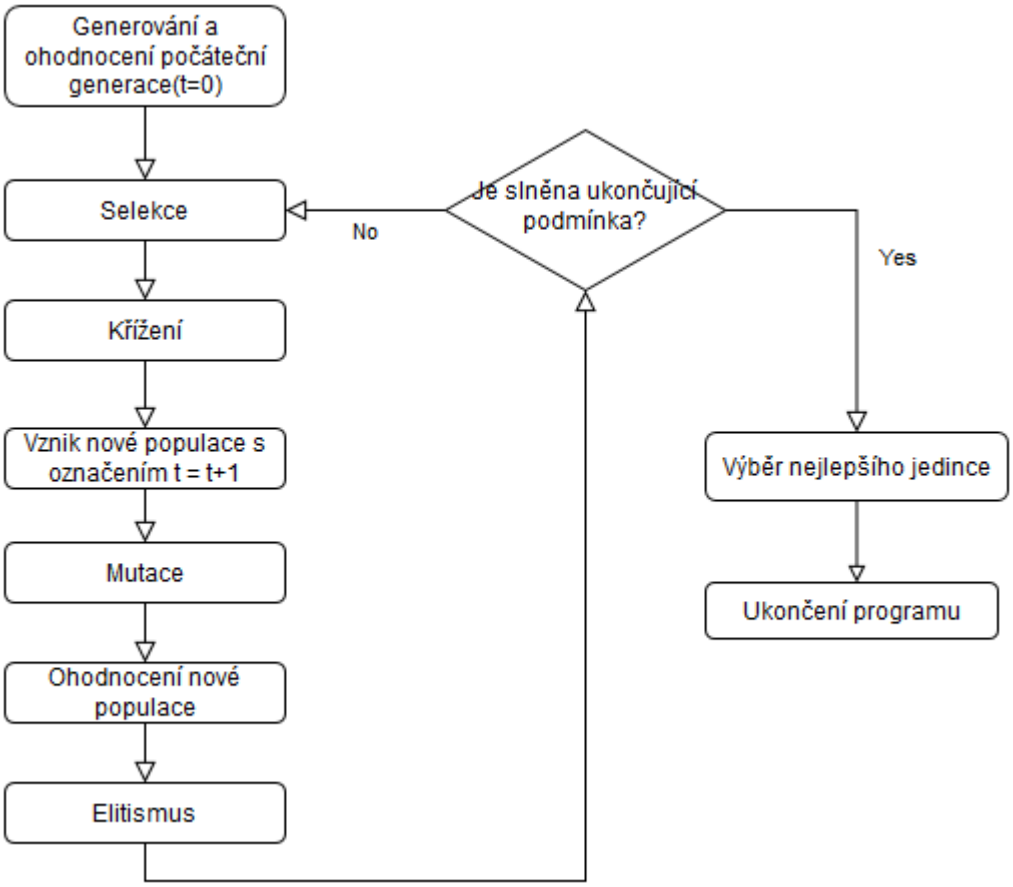
První, ale často opomíjenou fází tvorby každého algoritmu by mělo být porozumění problému a transformace dat do podoby, která bude pro průběh algoritmu snesitelnější. Toto samozřejmě nemusí platit v každém případě, existují i problémy, kde již žádná transformace není třeba. V případě, že jsou data kvalitně upravena, je možné výpočet algoritmu velmi zjednodušit a zrychlit.[1] Dále lze do této fáze také zahrnout nastavení genetických konstant, mezi které můžeme zařadit například počet generací, počet jedinců v generaci a pravděpodobnosti křížení, mutace a selekce, které budou popsány v následujících podkapitolách. Jako další můžeme nastavit již zmíněné podmínky ukončení algoritmu. Základní podmínkou je, že ohodnocení fitness dosáhne určité hodnoty. Další ukončovací podmínky mohou být maximální počet generací, stagnující hodnota fitness funkce, nebo uplynutý čas trvání algoritmu.[2][26]

Další fáze se již týkají samotného průběhu algoritmu. V první řadě je nutná počáteční inicializace. V této fázi je nutné vygenerovat a ohodnotit počáteční populaci. Generování probíhá volbou náhodných hodnot a jedinci budou tedy zcela náhodní. Po ohodnocení je tedy velmi malá pravděpodobnost, že mezi jedinci bude některý vykazovat přijatelnou hodnotu fitness.[1]

Dalším krokem podle evoluce je výběr vhodných kandidátů, pomocí kterých bude tvořena následující generace. Tomuto procesu se říká selekce, která zvolí kandidáty, kteří pomocí následujícího kroku, křížení, vytvoří nové jedince následující generace. Jedná se tedy o stejný princip jako u lidí, kteří spolu mají děti a každé dítě vlastní část genetické informace otce a část informace matky.[1]

V další fázi do algoritmu mohou vstoupit některé genetické operátory. Mezi nejznámější patří například křížení, mutace nebo elitismus. Pomocí operátoru křížení je vytvořena nová generace jedinců. Dále je možné použít mutaci, která má za úkol náhodně přetvořit nově vzniklého jedince podle toho, jak to určí programátor.[2]

Následuje ohodnocení nové generace pomocí stejné fitness funkce a následně ještě genetický operátor elitismus, který dokáže uchovat nejlepší jedince do dalších generací, tak aby v průběhu algoritmu nedošlo ke ztrátě některého z potenciálních řešení. Pokud je po ohodnocení splněna některá z ukončovacích podmínek, tak je algoritmus ukončen a za výsledek je prohlášen jedinec s nejlepším dosaženým ohodnocením. Pokud žádná podmínka splněna není, tak algoritmus pokračuje fází výběru jedinců ke křížení. Tento postup se opakuje, dokud není splněna některá z podmínek.[1][2]



Obrázek 11 - Fáze průběhu genetického algoritmu, zdroj: vlastní

3.3 Selekce

Proces selekce slouží k výběru vhodných jedinců ke křížení. Tyto jedince tedy nazýváme rodiče. Většinou jsou vybírány dvojice jedinců, u kterých jsou následně aplikovány genetické

operátory. Výběr dvojic se provádí náhodně, ale většinou jsou zvýhodněni jedinci s lepším ohodnocením fitness. Pro selekci vhodných jedinců slouží několik zavedených principů. Každý způsob má své výhody a podle jeho pravidel se určuje, jaká část populace se bude podílet na tvorbě populace nové.[25][27]

Mezi nejoblíbenější algoritmy selekce patří turnajové schéma, které napodobuje středověké rytířské souboje. Existuje více variant turnajového schématu. Nejjednodušší z nich je výběr nejlepšího jedince z několika náhodně vybraných, typicky jeden ze dvou, nebo ze čtyř. Další, poněkud složitější, je provedení více kol turnaje. V prvním kroku je určeno, na kolik kol se bude turnajové schéma konat a dále je nutné říci, s jakou pravděpodobností kolo vyhraje silnější jedinec. Podle počtu kol je vybráno několik náhodných jedinců, kteří utvoří dvojice. Každá dvojice provede souboj. S určenou pravděpodobností do dalšího kola postoupí jedinec s lepším ohodnocením fitness. Takto se v turnaji pokračuje do té doby, než není znám vítěz. Pravděpodobnost postupu silnějšího jedince je vhodné zvolit blízko hodnotě jedna, například 0,9. Tím se zaručí, že do tvorby nové populace mohou zasáhnout i jedinci, kteří nemají tak dobré hodnocení, ale po provedení genetického operátoru křížení mohou přinést potřebnou variabilitu do nové populace.[1][2][26]

Další oblíbený selekční algoritmus je ruletové kolo. Tento algoritmus pracuje pouze na principu náhodného výběru jedinců. Častým a efektivním vylepšením je vážené ruletové kolo, kde jedinci s lepším ohodnocením mají větší šanci, že budou vybráni. Tento algoritmus také zaručuje potřebnou variabilitu vybraných jedinců.[1][26]

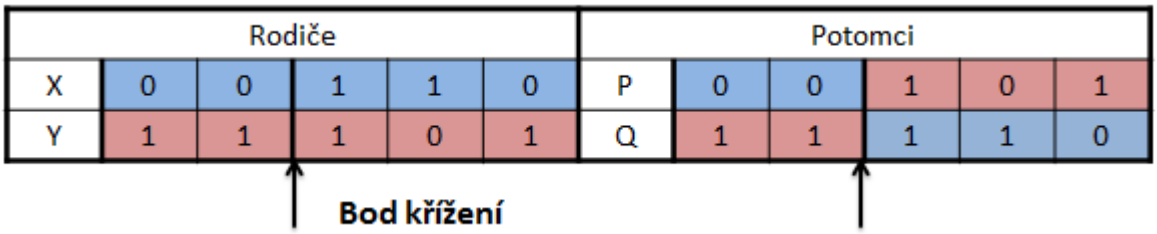
Po provedení selekce je velmi pravděpodobné, že někteří jedinci budou vybráni vícekrát a někteří naopak zahynou, tedy nebudou se podílet na tvorbě nové generace. Tento proces také odráží přirozenou evoluci. I v přírodě většinou přežije silný jedinec na úkor slabšího, ne tolik přizpůsobeného okolnímu prostředí, který předá genetickou informaci svým potomkům. V některých případech může přežít i slabší jedinec a i jeho genetická informace může být ve výsledku užitečná pro další generace potomků.[1]

3.4 Genetické operátory

Po dokončení selekce jsou aplikovány takzvané genetické operátory. Těchto operátorů existuje mnoho, ale nejdůležitější je křížení, které vytvoří z vybraných jedinců nové, tedy novou generaci. Další důležité a prakticky vždy používané jsou operátory mutace a elitismus.

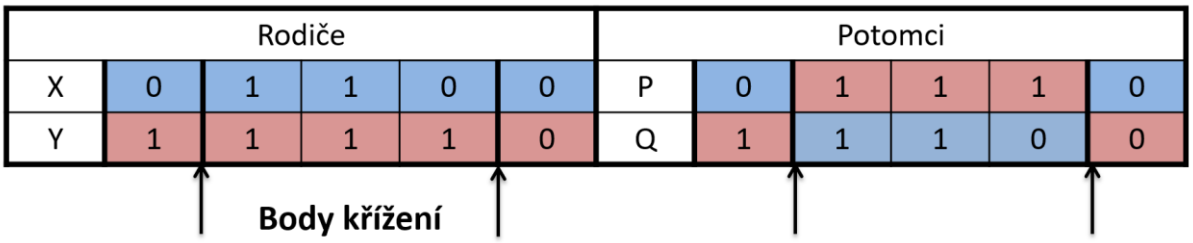
3.4.1 Křížení

Jak již bylo řečeno, výsledkem operace křížení je nově vytvořená generace z jedinců vybraných pomocí selekce. Stejně jako v přírodě si i zde rodiče vymění část genetické informace (genů) a tím vzniknou jejich potomci. Existuje několik druhů křížení, z nichž nejjednodušší je jednobodové křížení, tak jak je vidět na obrázku 12. Nejprve je náhodně zvolen bod křížení a následně se vymění části chromozomu za tímto bodem, a tím vzniknou dva noví potomci. Dále záleží, jestli do nové generace necháme postoupit oba, nebo jen jednoho nově vytvořeného potomka.[1][2][27]



Obrázek 12 - Jednobodové křížení, zdroj: vlastní

Kromě jednobodového křížení lze použít ještě vícebodové, kde je zvoleno více bodů křížení a je vyměněna například jen prostřední část chromozomu. U vícebodového křížení je také možné nové potomky skládat z více než dvou rodičů.[26]



Obrázek 13 - Vícebodové křížení, zdroj: vlastní

Stejně jako u selekce se i zde využívá náhody, a to při výběru bodu křížení. Pro výběr bodu křížení je vhodné zaručit, aby nebyl na začátku ani na konci chromozomu. To by totiž znamenalo, že oba rodiče postoupí do další generace beze změny. I tento jev může být žádoucí, ale je lepší ho určit takzvanou pravděpodobností křížení, která se většinou nastavuje na 95%. Tento údaj vypovídá, že ke křížení dojde s pravděpodobností 95% a s pravděpodobností 5% nedojde ke křížení u dané dvojice vůbec a rodiče postoupí do nové generace beze změny.[1]

3.4.2 Mutace

Dalším hojně využívaným operátorem je mutace. Princip tohoto operátoru je také odvozen z biologické evoluce. I v přírodě se může stát, že některý z genů potomka zmutuje a změní se. To samé se děje i v genetickém algoritmu po využití operátoru mutace. Nejčastější formou mutace je, že s určitou pravděpodobností, která zpravidla bývá velmi malá (například 1%), zmutuje náhodný gen chromozomu. Je ale na uvážení programátora a na podobě řešené úlohy jakou formu mutace zvolit. Musí se však dbát na fakt, že mutace má za úkol zabránit ztrátě potenciálně užitečné informace, zjednotvárnění vlastností v rámci populace a předčasné konvergenci populace.[1][2][25][27]

Mutace č. 1					
X	0	0	**1**	1	0
	0	0	**0**	1	0
Mutace č. 2					
Y	1	**0**	1	0	1
	1	**1**	1	0	1

Tabulka 3 - Příklad mutace

3.4.3 Elitismus

Princip elitismu je velmi jednoduchý. Předem zvolený počet nejlepších jedinců (elita) přežijí beze změny do další generace. Tyto jedinci se vybírají až na konci každého cyklu algoritmu, po ohodnocení nově vzniklé generace pomocí fitness funkce. Díky elitismu je tedy možné zachovat některé z dosavadně nalezených jedinců napříč několika generací, a tím zabránit ztrátě jedinců, kteří by mohli být potencionálními řešeními celého algoritmu.[1][2]

Po ohodnocení nově vzniklé generace *t* se tedy vybere požadovaný počet nejlépe ohodnocených jedinců a uloží se pro použití v další generaci, tak jak je naznačeno na obrázku 14. Po vytvoření a ohodnocení generace *t+1* se elitní jedinci vloží nejčastěji jako náhrada nejhorších jedinců v této generaci.[2]

Generace č.1						Fitness	Generace č. 2						Fitness
P	0	0	1	1	0	2	P	1	0	1	1	1	4
Q	1	0	1	0	1	3	Q	0	0	1	1	0	2
R	1	0	0	0	0	1	R	0	0	0	1	1	2
S	0	0	1	1	0	2	S	1	1	0	1	1	4
T	1	1	0	1	1	4	T	1	1	0	0	0	2

Obrázek 14 - Příklad elitismu, zdroj: vlastní

4 GENETICKÉ PROGRAMOVÁNÍ

Genetické programování je problematika, která stejně jako genetické algoritmy vychází z biologické evoluce. Genetické programování je však úzce specifikováno na řešení úloh, zabývajících se tvorbou počítačových programů, které dále řeší zadanou úlohu.[1]

V kapitole je tedy popsána základní terminologie i prvky, se kterými algoritmus pracuje. Dále principy fungování a hledání ideálního jedince, který co nejlépe vyřeší zadanou úlohu.

4.1 Základní pojmy

Jak již bylo řečeno, genetické programování také vychází z biologické evoluce. Fungují zde podobné principy jako u genetických algoritmů. V genetickém programování se také setkáme s pojmy jedinec, gen, populace, generace atd. Jediné, co se liší od genetického algoritmu, je ovšem podoba jedince. Jedinci zde nemají podobu pevného pole genů, ale takzvaného syntaktického stromu, kde jsou geny uspořádány do stromové struktury a jedinci mohou mít rozdílné počty genů.[1][24]

I další pojmy jsou stejné jako u genetického algoritmu. I zde musí být každý jedinec ohodnocen pomocí fitness funkce a toto ohodnocení většinou chybu programu od hledaného stavu.

Genetické programování také používá selekci pro výběr ideálních jedinců, dále také genetické operátory, křížení, mutace a další. Jejich princip zůstává stejný, je však nutné přizpůsobit funkčnost hlavně genetických operátorů pro jedince, které mají geny uspořádány v hierarchické struktuře. Dalším problémem může být různá velikost této struktury u každého jedince. Další změny a problémy budou popsány v následujících kapitolách, které se detailněji věnují problematice jednotlivých částí evoluce a základních částí genetického programování.[1][3]

Tyto algoritmy jsou většinou používány pro případy, kdy je potřeba vytvořit nějaký program, který řeší konkrétní úlohu. Lze je tedy použít například při řešení symbolické regrese, což je asi nejčastější způsob využití. Jedná se o problém, kdy jsou známé body například v prostoru, a pomocí algoritmu je nutné najít vhodný předpis funkce, která co nejvěrněji aproximuje všechny body. Takto lze nalézt úplné, či jen uspokojivé řešení zadaného problému. Využití tedy existuje v oblastech jako je například predikce počasí, vyšetření průběhu časových řad, hledání trendů nebo pouhé hledání průběhu funkce apod.[1][3]

4.2 Syntaktický strom

V genetickém programování má každý jedinec geny uspořádány do hierarchické struktury, která se nazývá syntaktický strom. Tyto stromy obsahují dva druhy genů a to terminály a funkce. Jelikož tyto geny mají jinou podobu a mohou představovat různá data, je syntaktický strom tvořen jako *k*-cestný strom. To znamená, že existuje jeden kořen a každý uzel může obsahovat *k* potomků.[1][3]

Syntaktický strom omezuje počet potomků podle pravidel, které mají jednotlivé neterminálové geny. Například pokud strom obsahuje gen sčítání, je jasné, že musí mít 2 potomky. Pokud je ovšem přítomen gen například odmocnina, tak počet jeho potomků musí být roven jedné. Tomuto jevu se říká arita funkce, která určuje, kolik potomků mají jednotlivé funkce přítomné v syntaktickém stromě.[1][3]

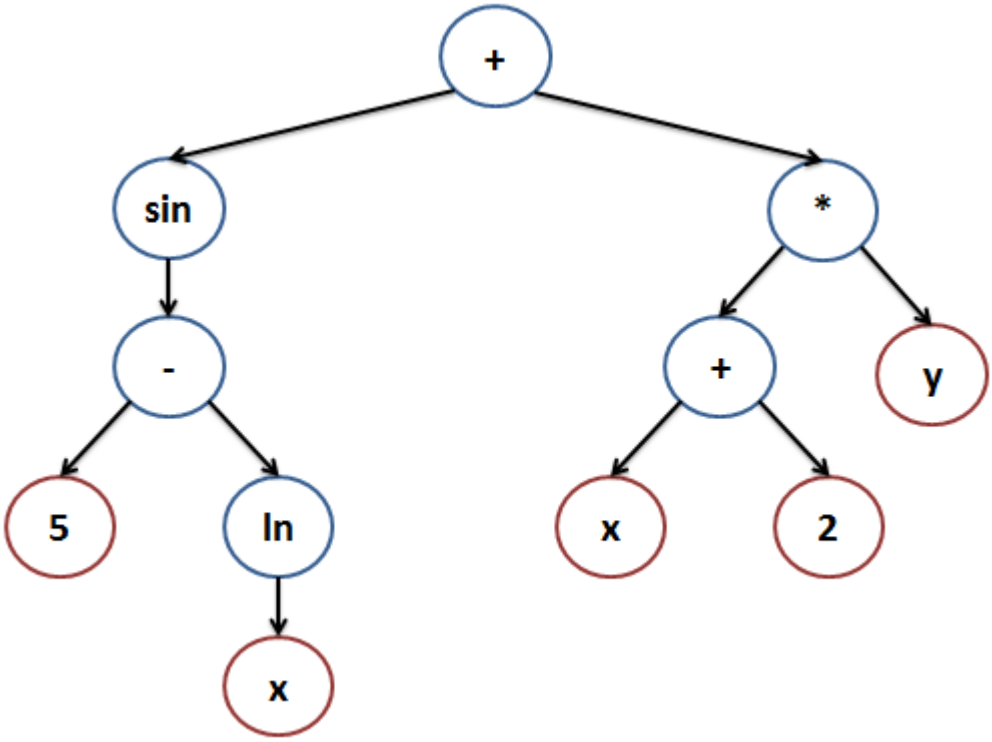
4.2.1 Terminály

Terminál je tedy gen, pod kterým je možné představit si například číslo. Jedná se o soubor proměnných a konstant. Tento soubor je definován před spuštěním algoritmu a obsahuje všechny přípustné hodnoty, se kterými může program pracovat. Ve většině případů se jedná o kombinaci číselných hodnot a proměnných, ale záleží na typu řešené úlohy. Například pro účely prohledání bludiště jsou terminály definovány jako akce, které mohou být použity, jako otočení, nebo pohyb dopředu. V syntaktickém stromě jsou terminály vždy ve všech listech tohoto stromu.[1][28]

Terminály je možné definovat jako konstanty, tedy například čísla 1, 3, 8 a žádné jiné nemůže být vloženo na místo terminálu do syntaktického stromu. Na druhou stranu lze také definovat jako proměnnou (x, y, z, rychlost apod.), což se využívá zejména při symbolické regresi, nebo náhodnou hodnotu ze zadaného rozsahu, takže je možné vygenerovat (většinou již při tvorbě stromu) náhodnou hodnotu v rozsahu 0–10 a na místo terminálu do stromu bude vždy vloženo číslo v tomto rozsahu.[3]

4.2.2 Funkce

Interní uzly syntaktického stromu tvoří takzvané funkce. K provedení těchto operací musí mít funkce na vstupu vstupní hodnoty a po provedení funkce je vypočítán výstup. Tyto funkce mají většinou podobu matematických operací, které využívají jako vstupy terminály obsažené v listech stromu, nebo výstupy jiných funkcí obsažených v podřazených uzlech. Uzel typu funkce tedy může obsahovat operace sčítání, odčítání, ale i například goniometrické funkce, mocninu, odmocninu, atd.[1][28]

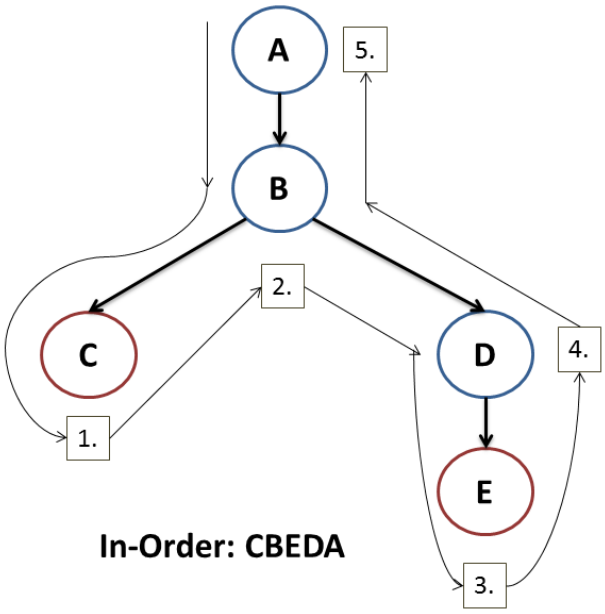


Obrázek 15 - Příklad syntaktického stromu, zdroj: vlastní

4.2.3 Průchod syntaktickým stromem

Na obrázku 15 je vidět syntaktický strom, který představuje určitý předpis funkce. Po prohlédnutí je možné určit minimální množiny terminálů a funkcí, které byly použity pro tvorbu tohoto stromu. Množina terminálů tedy obsahuje minimálně proměnné *x, y* a konstanty 2 a 5. Je možné, že obsahuje více prvků, které ale nebyly použity pro tvorbu tohoto stromu. Množina funkcí obsahuje alespoň operace sčítání, odčítání, násobení, sinus a logaritmus. Pokud se snažíme vytvořit ze stromu předpis funkce, který je pro lidské oko lépe čitelný, je nutné použít na stromovou strukturu in-order prohlídku.[3][29]

Pro in-order prohlídku platí, že jako první se vezme nejlevější list, následně jeho otec a poté pravý bratr. Tímto způsobem se vytvoří rovnice, která věrně popisuje syntaktický strom. Pokud je jako příklad využit levý podstrom z obrázku 15, tak se jako první vezme gen 5 a poté jeho otce, tedy operátor odečítání. Následně se do rovnice přidá jeho bratr. Tím je však logaritmus, který obsahuje ještě jednoho potomka. Je tedy nutné aplikovat stejný mechanismus, tedy levý list, otec a následně bratr. V tomto případě je jen jeden list a otec, proto bude dosavadní výsledek roven *5-ln(x)*. Následně prohlídka přechází na gen sinus, tudíž předpis tohoto podstromu bude roven *sin(5-ln(x))*. Obecný princip prohlídky je znázorněn na následujícím obrázku 16.[29]



Obrázek 16- Příklad principu in-order prohlídky stromu, zdroj: vlastní

Pokud tedy aplikujeme mechanismus in-order prohlídky na syntaktický strom vyobrazený na obrázku 15, vznikne předpis funkce, který je popsaný v rovnici 4.

Rovnice 4 - Předpis funkce obsažené v syntaktickém stromu

$$f(x,y) = \sin(5 - \ln x) + (x + 2) * y$$

Existují i jiné způsoby prohlídek syntaktického stromu, které ale nejsou tolik přehledné a ve výstupu se velmi těžko orientuje. Takovými prohlídkami jsou post-order, pre-order a prohlídka do šířky.[29]

4.3 Fitness funkce

Fitness funkce určuje kvalitu každého jedince v populaci. Kvalita je vyjadřovaná číselnou hodnotou, která je závislá na typu úlohy a definici fitness funkce. Mohou vzniknout dvě situace a to čím větší číslo, tím lepší jedinec, nebo naopak, čím bližší nule, tím lepší jedinec.[1][28]

Výpočet ohodnocení v genetickém programování spočívá většinou v opakovaném vyhodnocování syntaktického stromu. Nejlepším příkladem je symbolická regrese, kde se ve stromě většinou nachází jedna nebo více proměnných, za které jsou v průběhu výpočtu dosazovány různé vstupní hodnoty a výsledky jsou porovnávány s očekávanými hodnotami, které jsou známé.[1] Rozdíly mezi výsledky a očekávanými hodnotami jsou následně zprůměrovány například pomocí střední kvadratické chyby. Tím je zjištěna celková chyba

na všech zadaných vstupních hodnotách. Tento proces se opakuje pro každého jedince v populaci.[3]

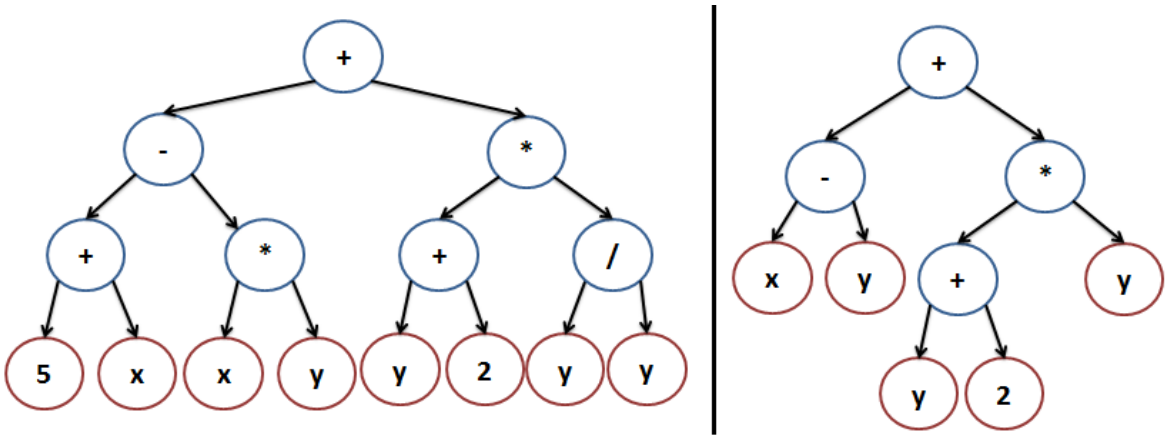
4.4 Tvorba počáteční populace

Stejně jako u genetického algoritmu je i zde nutné náhodně vygenerovat první generaci jedinců. Generování jedince probíhá náhodným výběrem z předem definovaných množin terminálů a funkcí. Dále je nutné definovat genetickou konstantu, která bude udávat maximální hloubku generovaných stromů. Dále je při generování stromů nutné dbát na pravidla, které mají jednotlivé funkce, tedy na aritu funkcí.[28]

Je zřejmé, že takto náhodně vygenerovaní jedinci nebudou mít potenciál, který bude řešit úlohu. Přesto je pravděpodobné, že alespoň část populace bude obsahovat nadějné genetické informace, které budou základem pro další fáze algoritmu a pomocí nichž se budou tvořit další generace kvalitních jedinců.[1]

Při tvorbě první generace se dbá na rozmanitost vygenerovaných jedinců. Je vhodné, aby byli v první generaci zastoupeni jedinci s různými hloubkami syntaktických stromů a aby byli různě stavěni. K tomuto účelu slouží úplná a růstová metoda tvorby syntaktických stromů. Úplná metoda generuje plné stromy maximální hloubky. To znamená, že všechny větve každého stromu mají maximální hloubku.[3][28]

Růstová metoda generuje neúplné stromy proměnné hloubky, ale maximální hloubka je omezena podle nastavené konstanty. Pomocí této metody je možné generovat lépe použitelné jedince. Je však vhodné tyto metody kombinovat v poměru 50:50, tím se zaručí požadovaná rozmanitost první generace jedinců. Dále je také vhodné v průběhu generování populace měnit maximální hloubku stromů tak, aby byly jedinci více rozdílní a byla větší šance, že jejich genetická informace bude kvalitnější pro následující generace jedinců.[3][28]



Obrázek 17 - Příklad stromů vygenerovaných úplnou a růstovou metodou, zdroj: vlastní

4.5 Průběh evoluce

Po vygenerování počáteční populace je možné spustit samotný algoritmus genetického programování, který je rozdělen do několika opakujících se fází. Průběh genetického programování je takřka stejný jako u genetického algoritmu a pro orientaci je tedy možné použít obrázek 11.[1]

V první řadě je nutné pomocí selekce vybrat vhodné jedince a následně na vybrané n-tice jedinců aplikovat genetický operátor křížení. Dále je vhodné na nově vzniklé jedince aplikovat další genetický operátor mutace a následně nově vzniklé jedince ohodnotit pomocí fitness funkce. [1]

Ve většině případech je žádoucí použít i další genetické operátory, jako například elitismus. Po dokončení cyklu jsou zkontrolovány ukončující podmínky. Pokud je některá z podmínek splněna, tak je algoritmus ukončen a jako výsledek vrácen nejlépe ohodnocený jedinec. Pokud není žádná z podmínek splněna, tak algoritmus pokračuje dalším cyklem, tedy selekcí, křížením, atd.[1]

4.6 Selekce

Proces selekce má stejný princip jako u genetických algoritmů. Jedná se o proces výběru například dvojic, které se mezi sebou budou křížit a tvořit novou generaci jedinců. U genetického programování se nejvíce využívá metody turnajového schématu, nebo ruletového kola.[3][28]

Jediným rozdílem je, že do metody nevstupují jedinci ve formě sekvence genů, ale ve formě syntaktického stromu. Je tedy nutné metodu mírně upravit. Ovšem podstatou této metody je práce s ohodnocením jedinců, podle kterého výběr probíhá.[3]

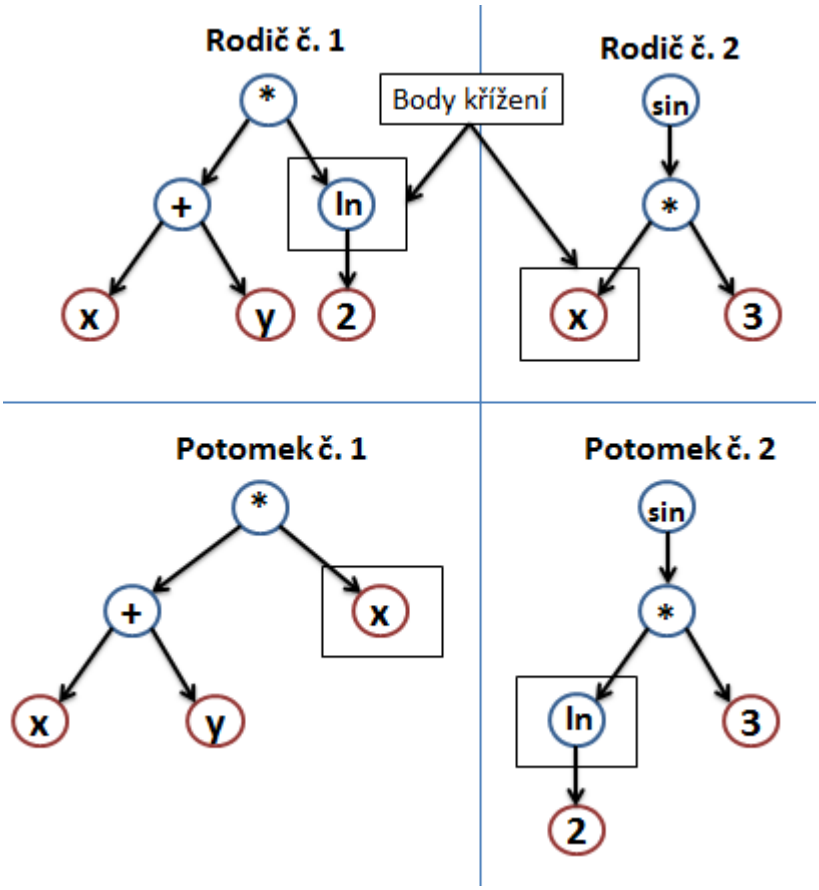
4.7 Genetické operátory

I principy genetických operátorů zůstávají stejné. Je však nutné si uvědomit, že použití genetických operátorů na syntaktické stromy bude mít jiná pravidla a bude přinášet jiné problémy než použití na sekvence genů.

4.7.1 Křížení

Nejdůležitější operátor je křížení. Účelem křížení je vytvoření nové generace jedinců vytvořených z vybraných jedinců předešlé generace. Rozdílem ovšem je, že nyní je třeba křížit syntaktické stromy. Je nutné zvolit uzel stromu, který bude sloužit jako bod křížení. Tento bod je vybrán náhodně v každém rodiči zvlášť a následně jsou vyměněny jejich podstromy začínající v těchto uzlech. Tento postup se nazývá jednobodové křížení. Tím, že je vyměněn jen jeden podstrom bodu křížení, je vždy zajištěna syntaktická správnost nově vzniklých jedinců, to znamená, že nikdy nemůže být porušena maximální arita daných funkcí.[1][3]

Problém ovšem nastává v hloubce nově vzniklých stromů. Může nastat situace, kdy podstatná část jednoho rodiče bude vyměněna například jen za list rodiče druhého. Tímto způsobem je možné, že hloubka stromů bude po několika generacích velmi velká a výpočetní doba algoritmu se bude velmi prodlužovat. Je tedy vhodné určit maximální hloubku, které může syntaktický strom při řešení daného problému dosáhnout. Většinou se maximální hloubka určuje na trojnásobek hloubky stromu generovaného v počáteční populaci, ale na základě složitosti řešené úlohy a dalších kritérií je možné tuto hodnotu zvolit jiným způsobem. Při překročení maximální hloubky stromu se většinou používá postup zamítnutí tohoto potomka a nahrazení jedním z rodičů.[3][28]

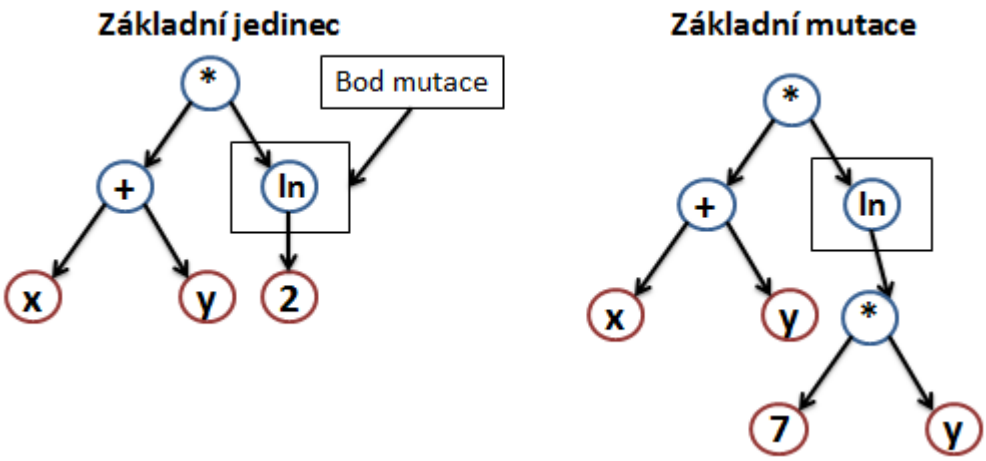


Obrázek 18 - Křížení syntaktických stromů, zdroj: vlastní

4.7.2 Mutace

Mutace je v genetickém programování obsažena z důvodu přínosu náhodné změny. Pravděpodobnost mutace se většinou nastavuje kolem jednoho procenta. Zmutování syntaktického stromu spočívá ve výběru náhodného uzlu, tedy bodu mutace. Celý podstrom bodu mutace je smazán a nahrazen nově vygenerovaným stromem. Ke generování nového podstromu se taktéž využívá úplná, či růstová metoda generování stromu. Nastavení maximální hloubky takto vygenerovaného podstromu se nastavuje, tak aby výsledný strom nebyl moc hluboký.[1][3]

Existují i jiné metody, které se dají využít pro mutaci jedinců. Výběr této metody záleží na řešené úloze a volbě programátora. Jednou z takových metod je uzlová metoda. Pomocí této metody je vyměněn terminál jiným terminálem, nebo funkce jinou funkcí se stejnou aritou. Pomocí vyzvedávající mutace se vymění celý strom jeho náhodným podstromem a pomocí smršťující mutace se vymění náhodný podstrom jediným terminálem. Každý druh mutace přináší pro zmutované jedince jiné výhody. Je tedy na úvaze jakou, či případně jaké metody využít.[1][3]



Obrázek 19 - Příklad mutace syntaktického stromu, zdroj: vlastní

4.7.3 Další genetické operátory

V genetickém programování lze využít další, sekundární, genetické operátory. Tyto operátory nemají takový vliv na tvorbě nové generace, takže jsou využívány společně s těmi základními. Mezi tyto operátory patří například permutace, editace, zapouzdření a decimace.[1]

První možnost je permutace, kde do křížení vstupuje pouze jeden rodič. Jedná se tedy o asexuální operátor, kde se vybere náhodný uzel, který odpovídá funkci s argumenty, které jsou následně vyměněny. Výměna musí proběhnout podle pravidel jednotlivých funkcí, tedy hlavně na základě arity funkce, která určuje počet potomků. Tato metoda je sice jednoduchá na implementaci, ale není tolik účinná.[1]

Pomocí editace je možné automaticky zjednodušit a upravit jednotlivé syntaktické stromy. Za běhu evoluce lze tedy aplikovat na strom některá pravidla, díky nimž mohou být různé části stromu zjednodušeny. Tento mechanismus spočívá ve výpočtu částí stromu, které je možné vypočítat, například pokud strom obsahuje podstrom *2*3*, je možné tyto 3 uzly zjednodušit na jeden terminál s číslem *6*. Je však nutné tento operátor používat s mírou, jelikož aplikace pravidel na syntaktické stromy velmi prodlužuje celkovou dobu evoluce. Na druhou stranu může použití pozitivně ovlivnit evoluci lepším vývojem populace.[1]

Zapouzdření dokáže identifikovat potenciálně důležitý podstrom a ten uložit a pojmenovat. Dále může být tento podstrom využit jako nový terminál. Je tedy možné se na tento podstrom odkazovat jako na terminál a jeví se tedy jako jeden uzel. Díky tomu je chráněný před rozdělením pomocí jiných genetických operátorů. Díky zapouzdření je možné opakovaně využívat stejný kód, a tím zjednodušit syntaktický strom.[1]

Poslední možnost je decimace. Pomocí této operace je možné zredukovat počet jedinců v populaci. Decimace využívá faktu, že první generace obsahuje mnoho jedinců, kteří nemají význam pro další generace. Proto se tedy zvolí počáteční populace, která obsahuje velké množství jedinců, která se po určitém počtu generací díky decimaci zredukuje. Jedinci, kteří budou odstraněni, jsou vybíráni náhodně, tak aby byla zaručena rozmanitost zredukované populace.[1]

4.8 Podmínky ukončení evoluce

Evoluce je ukončena po splnění některé ze zadaných ukončovacích podmínek. Je možné využít několik těchto podmínek najednou. Nejčastěji využívané je nalezení úplného nebo částečného řešení. Další hojně využívanou ukončovací podmínkou je maximální počet generací.[3]

Dále je možné evoluci zastavit v případě, kdy se po určitý počet generací nezmění nejlepší jedinec. Tedy hodnota fitness funkce nejlepšího jedince v určitém počtu po sobě jdoucích generacích stagnuje.[3][26]

Poslední podmínky jsou podmínky konvergence, kdy je možné hlídat, zda se v nejlepších jedincích neobjevují stejné geny. To je důkazem konvergence genů, a je to důvod pro zastavení evoluce. Je také možné evoluci ukončit při konvergenci populace. Je možné pohlídat vývoj evoluce napříč populacemi. Pokud není určitý rozdíl mezi průměry ohodnocení všech jedinců v několika po sobě jdoucích populacích, znamená to, že populace dokonverguje, což je důvod k zastavení evoluce.[3][26]

5 HLUBOKÉ GENETICKÉ PROGRAMOVÁNÍ

Název této kapitoly obsahuje spojení dvou pojmů, a to hluboké učení a genetické programování. Cílem této kapitoly a zároveň i této práce je popsat a vyzkoušet některé principy hlubokého učení a konvolučních neuronových sítí aplikovaných na genetickém programování.

Jak již bylo řečeno, hlavní myšlenkou hlubokého učení a CNN je rozdělení výpočtu do více vrstev. CNN se dělí na dvě hlavní vrstvy, *feauture extraction* a následné dořešení problému pomocí klasické neuronové sítě. Myšlenkou hlubokého genetického programování je rozložit algoritmus genetického programování do jakýchsi pomyslných vrstev, které by následně jako celek řešily zadanou úlohu. Tento princip by měl zvýšit výkon genetického programování při řešení složitých a komplexních vícerozměrných úloh.[4][30]

Kapitola obsahuje popis symbolické regrese, jako základního problému řešeného pomocí GP, na kterém budou všechny metody implementovány. Nadále kapitola obsahuje popis některých již existujících metod hlubokého genetického programování, nebo jiných metod zvyšování výkonu genetického programování, na které se dá nahlížet jako na hluboký. Takové metody mohou být například metoda FFX, nebo některé z ensemble metod. V poslední řadě zde budou popsány návrhy na nově vymyšlené metody hlubokého genetického programování.

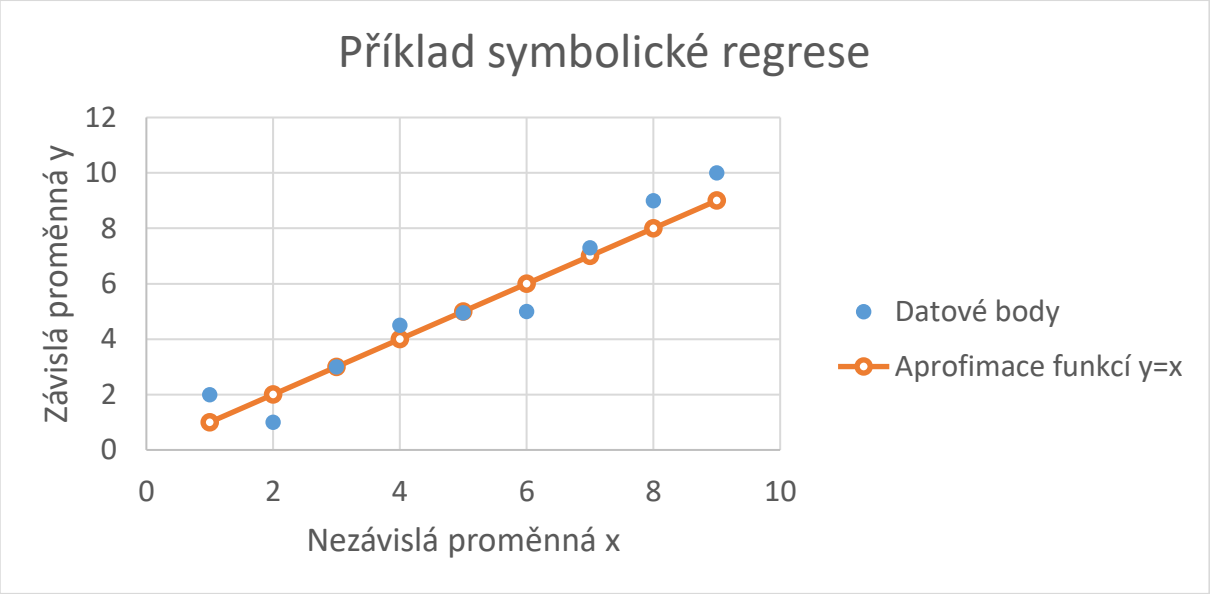
5.1 Symbolická regrese

Problém symbolické regrese je jedním ze základních problémů, která může být řešena pomocí genetického programování. Základní myšlenkou symbolické regrese je nalézt matematický popis závislosti, většinou předpis funkce, z dat, které většinou představují body na číselné ose. Příklad je naznačen v tabulce 4 a následně na obrázku 20. Trénovací množina dat tedy obsahuje několik nezávislých proměnných, podle počtu dimenzí problému a závislou proměnnou. Takový soubor může obsahovat i tisíce bodů. Čím větší soubor je, tím je hledání řešení náročnější operace. Na druhou stranu, pokud je tento soubor malý, není k dispozici dostatečný rozptyl hodnot k nalezení optimálního řešení daného problému.[32]

Symbolická regrese (SR), oproti lineární, která vyhledává pouze správné hodnoty jednotlivých parametrů, vyhledává i správnou matematickou strukturu. Jedná se tedy o výběr vhodné kombinace parametrů, konstant a matematických operací vybraných z předem určených množin. Cílem je minimalizovat chybu, která vzniká při aplikaci modelu na vstupní data, která se SR snaží aproximovat na závislou proměnnou. Pro řešení problému SR je možné využít i jiné metody než GP, například FFX.[32]

Nezávislá proměnná x	1	2	3	4	5	6	7	8	9
Závislá proměnná y	2	1	3	4,5	4,95	5	7,3	9	10

Tabulka 4 - Příklad trénovací množiny dat



Obrázek 20 - Příklad symbolické regrese

Testování metod v této práci je aplikováno výhradně na problém symbolické regrese. Pro testování metod jsou použity různé, i vícerozměrné funkce.

5.2 FFX metoda

Tuto metodu objevil a popsal ve své literatuře Trent McConaghy[31]. Jedná se o techniku, která řeší výhradně problém symbolické regrese. Tato metoda představuje rychlou, škálovatelnou a deterministickou funkci, která vyhledává vhodné kandidáty z mnoha vygenerovaných funkcí. FFX nepracuje na bázi evoluční strategie genetického programování, ale zároveň je jimi tato metoda velice ovlivněna.[31]

V prvním kroku se vygeneruje velké množství základních náhodných funkcí. Generování probíhá z předem známé množiny terminálů a neterminálů, které jsou nakombinovány do předpisů funkcí. V první fázi se generuje každý terminál na každý povolený exponent, následně je na každý takto vygenerovaný předpis aplikován unární matematický operátor a nakonec se vytvoří páry z předešlých dvou kroků.[33] Dále probíhá metoda částečně regulovaného učení, která zajišťuje odladění koeficientů ve vygenerovaných funkcích tak, aby co nejlépe odpovídaly hledanému řešení. Učení je tedy velmi rychlé, jelikož je jeho složitost podobná metodě nejmenších čtverců. Díky tomu se minimalizují rozdíly mezi výsledky

modelů a očekávanými výsledky. Algoritmus je zastaven po dosažení chyby, která je menší než předem zadaná hodnota.[31]

Pro běh FFX je nutné definovat pouze množiny terminálů a neterminálů, neexistují žádné jiné parametry, které je nutné nastavovat. Neexistuje tedy žádná obdoba genetických konstant, které je nutné ladit a nastavovat, aby byl algoritmus účinnější. Lze definovat ještě ukončující podmínky, ale ani ty nejsou povinné. Slouží pouze pro předčasné ukončení učení například v případě, že se chyba modelu nezlepšuje. Dále je možné omezit počet vygenerovaných základních funkcí, díky čemuž se výrazně zkrátí výpočetní doba.[33]

Výhodou FFX je, že metoda je velmi rychlá díky využití regularizovaného učení, která se dokáže naučit pro několik funkcí i tisíce koeficientů poměrně snadno. Dále má FFX srovnatelně kvalitní predikci jako GP a je deterministický.[31] Výhodou, ale zároveň nevýhodou se může stát i fakt, že FFX vytváří mnohem menší a jednodušší modely než GP díky omezené maximální složitosti predikční funkce. Výhoda se může projevit při hledání jednoduššího modelu, ovšem při hledání složité funkce, se může problém projevit. Tento problém lze celkem snadno odstranit na úkor výpočetní doby algoritmu, čímž se ovšem mohou vytratit výhody FFX oproti GP. Dalším problémem, který již tak snadno odstranit nelze je, že se FFX nemůže naučit vnitřní parametry funkce, tudíž nemůže vygenerovat například funkce $sin(2x)$, nebo $ln(x+2)$.[33]

5.3 Ensemble learning

Ensemble techniky slouží v oboru strojového učení ke zlepšení kvality řešení daného problému pomocí kombinace více modelů a minimalizace vzniklé chyby. Použité modely, které se kombinují, mohou být například klasifikátory nebo experti a jejich použití pomáhá zlepšit stabilitu a přesnost algoritmů strojového učení.[38] Ensemble learning tedy ve strojovém učení slouží k zlepšení predikce, klasifikace, nebo, hlavně v tomto případě, ke zlepšení aproximace funkce. Dále může sloužit ke zlepšení ohodnocování některých modelů nebo k výběru optimálních řešení. Mezi základní metody patří max voting, průměrování a vážené průměrovaní. Další, složitější metody, využívají redukci chyby nebo zlepšení predikce modelu.[36]

Při použití jakéhokoliv algoritmu strojového učení bude ve výsledných řešeních problému přítomna chyba, budou tedy nedokonalé. Tato chyba, jak je již zmíněno v minulých kapitolách, je rozdíl mezi skutečnými a vypočítanými hodnotami a nazývá se chyba predikce. Hlavním úkolem každého algoritmu AI je minimalizovat tuto chybu, a tím získat optimální

řešení daného problému. Chybu predikce lze rozdělit na chybu redukovatelnou a neredukovatelnou. Jedním z hlavních myšlenek ensemble technik je tedy dále minimalizovat redukovatelnou chybu.[37]

Neredukovatelná chyba je taková, která z modelu nelze odstranit. Na druhou stranu redukovatelnou chybu lze zmenšit nebo odstranit úplně. Redukovatelná chyba se dále dělí na chybu rozptylu a chybu zkreslení (bias).[37] Tato chyba lze odstranit metodou zvanou bagging (pytlování). Chyba zkreslení, neboli bias, lze zredukovat pomocí metody boosting.[36] Poměr všech typů chyb v modelu lze vyjádřit pomocí vzorce:

Rovnice 5 - Poměr chyb v predikčním modelu[37]

$$celková\ chyba = (bias)^2 + rozptyl + neredukovatelná\ chyba$$

Poslední metodou, která se netýká redukce chyby, ale soustřeďuje se na zlepšení predikce, je stacking (stohování).[36]

Všechny ensemble techniky lze rozdělit do dvou skupin, a to sekvenční a paralelní. Sekvenční techniky spočívají v postupném generování modelů. Každý nově vzniklý model vychází ze svého předchůdce a snaží se zmenšit jeho chybu. Tento postup se opakuje, dokud není nalezeno dostačující řešení daného problému. Tyto metody se také nazývají posilovací (boosting).[36]

Paralelní metody naopak využívají nezávislost jednotlivých modelů. Podle vybrané metody se dále volí strategie vylepšení modelů. Může se jednat například o výběr nejlepšího z *n* modelů nebo generování modelů z různých podmnožin tréninkových dat. Do této skupiny jsou zařazeny například hlasovací metody nebo metoda pytlování (bagging).[38]

5.3.1 Max voting

Metoda používaná hlavně při klasifikaci. Pro jeden vstupní datový objekt je vytvořeno více predikčních modelů. Výsledek každého modelu, v případě klasifikace se jedná o označení skupiny, do které je objekt zařazen, je považován za hlasování. Po vytvoření všech modelů daného objektu se hlasy sečtou. Skupina, která pro daný objekt získá nejvíce hlasů je brána jako konečná předpověď.[36]

5.3.2 Průměrování

Podobně jako u hlasování je zde vytvořeno více modelů k jednomu datovému objektu. U metody průměrování však neprovádíme součet hlasů, ale odpovědi všech modelů jsou zprůměrovány. Výsledný průměr je brán jako konečná odpověď. Tuto metodu lze využít

pro predikce v regresních problémech nebo u pravděpodobnostních problémů s klasifikací.[36]

5.3.3 Vážené průměrování

Jedná se o vylepšení metody průměrování, kdy je pro konečný výpočet předpovědi použit vážený průměr. Každý z vygenerovaných modelů má přiřazenou váhu, která určuje důvěryhodnost jeho předpovědi. Čím vyšší váha je, tím je model důvěryhodnější a jeho předpověď má větší vliv na utváření konečné predikce.[36]

5.3.4 Pytlování (bagging)

Metoda pytlování je paralelní ensemble metoda, která se snaží zkombinovat několik vygenerovaných modelů, za účelem získání kvalitnějšího řešení daného problému. Jak již bylo zmíněno, pytlování používá pro generování modelů různé sady vstupních dat. Při použití stejné sady pro každý model by mohlo dojít k situaci, kdy budou pro generování použita stejná data a výsledek modelů bude tedy stejný.[38]

Pro každý nový model je tedy z původní sady dat vytvořena podmnožina, ze které je následně vygenerován. Pro příklad symbolické regrese je možné vytvořit podmnožiny terminálů nebo neterminálů. Důsledkem toho je možné omezit počet dimenzí a počet přípustných neterminálů. Dále je možné vytvořit podmnožinu vstupních datových bodů. Použití pytlování znamená vytvoření jednodušších funkcí, které jsou následně sjednocovány do jedné. V GP je možné tímto způsobem vytvářet subpopulace, z nichž jsou vybíráni jedinci, kteří následně vytvoří složený výstup.[39]

5.3.5 Posilování (boosting)

Metoda boosting byla původně vyvinuta pro řešení binárních problémů. Časem byl ovšem algoritmus adaptován i na složitější klasifikační problémy a následně i pro řešení regrese pomocí GP.[36]

Varianta vyvinutá pro řešení binárních a klasifikačních problémů byla nazvána AdaBoost. Jedná se o sekvenční algoritmus, kde každý model vychází ze svého předka a snaží se vylepšit jeho nedostatky tak, aby nový model byl o něco lepší. K tomu je využito ohodnocení tréninkových dat pomocí váhy.[38] Při tvorbě nového modelu se algoritmus snaží soustředit na data, která byla díky ohodnocení vybrána jako slabá místa. Nově vzniklý jedinec by měl být vždy o něco kvalitnější než jeho předchůdce.[39]

Pro využití v GP byl následně rozšířen tento algoritmus na BoostGP, který je určen pro řešení problémů týkajících se regrese. V prvním kroku se nastaví všem datovým bodům stejná váha *1/N*, kde *N* je počet datových bodů. Následně je vypočítána chyba pro každý datový bod, dále celková a průměrná chyba. Díky těmto údajům lze nadále upravit váhy jednotlivých datových bodů. Použití tohoto algoritmu by mělo v každém případě zaručit lepší výsledky než náhodné hledání řešení, tedy klasické GP. Navíc je možné BoostGP vhodně spojit s BagGP (pytlování) a získat ještě lepší algoritmus pro řešení regresních problémů[39]

5.3.6 Stohování (stacking)

Stohování je metoda, která se využívá hlavně při problému klasifikace. Základní myšlenkou je kombinace modelů vzniklých různými metodami pomocí například metaklasifikátoru nebo metaregresoru.[38]

Řešení pomocí stohování kombinuje několik algoritmů učení, například neuronová síť, GA, GP, nebo jiné metody. Pomocí těchto základních klasifikátorů jsou utvořeny modely, které jsou následně slučovány pomocí metaklasifikátoru do výsledného modelu. Při použití na problémy rozřazení dat do skupin se jedná o jeden z nejlepších algoritmů, které se v této době dají použít.[36] Základní klasifikátory vygenerují modely, které jsou sice kvalitní, ale jejich sloučením dojde k prolnutí hranic jednotlivých skupin. Díky tomu je možné vytvořit tyto hranice přesněji a výsledný model tedy dokáže určit správnou skupinu s větší pravděpodobností.[38]

5.4 MultiTree metoda

Představa řešení této metody spočívala ve faktu rozdělení syntaktického stromu na více menších podstromů, které budou uloženy zvlášť. Záměr tohoto řešení spočíval ve zmenšení syntaktickým stromů, a tím by mělo docházet ke zjednodušení operací, kdy se prochází velký syntaktický strom, jako jsou operace fitness vyhodnocení, křížení, mutace a další operátory. Dále byly navrhovány možnosti rozšíření operátoru křížení a mutace na více úrovní.

Tato operace spočívá ve výměně dvou celých podstromů, čímž se ve výsledku syntaktický strom hodně změní a v dalších krocích může docházet ke křížení na úrovních podstromů. Podobné mechanismy byly zkoušeny i u operátoru mutace. Následně měla být použita ještě vrstva s váhami, kde by měl každý podstrom svojí váhu a ta by v konečném výsledku vylepšila celkový výsledek. Při výpočtu fitness bylo implementováno ladění těchto vah.

Během implementace bylo bohužel zjištěno, že ladění vah není možné udělat tak, že se zlepší částečný výsledek daného podstromu. Nebylo tedy možné ladit samostatné podstromy, které by ve výsledku byly účinnější pro řešení problémů SR pomocí GP. Jelikož by tato metoda bez vážené vrstvy nepřinesla požadovaný přínos pro práci, byla z dalších částí, jak implementačních, tak experimentálních, vyřazena a dále již není používána.

5.5 MultiEngine metoda

Tato metoda se snaží vhodně zkombinovat principy FFX metody a ensemble technik a tuto kombinaci využít v GP. Cílem je v prvním kroku vygenerovat malé modely, tak jako tomu je u FFX. V dalších krocích by tyto jednoduché modely měly posloužit pro tvorbu konečného, výsledného modelu. Rozdělení algoritmu do vrstev je tedy zřejmé. První vrstva se stará o tvorbu malých, jednoduchých modelů a druhá vrstva je klasický algoritmus GP, který má rozšířenou množinu terminálů o modely vzniklé v první vrstvě.

Generování malých modelů probíhá jako odlehčený algoritmus GP. Základní GP vytváří jedince a na základě fitness ohodnocení a genetických operátorů se tyto jedince snaží zdokonalovat, a tím nalézt řešení daného problému. Účelem první fáze metody MultiEngine ovšem není nalézt řešení problému, ale vygenerovat několik nových, malých modelů, které poslouží pro další fáze algoritmu. Základním principem je spuštění několika algoritmů GP, kde jedinci nejsou ohodnocováni pomocí fitness funkce, protože toto ohodnocení není pro danou fázi podstatné. Dále je každý tento GP spuštěn s jinou, náhodnou podmnožinou neterminálů, což by mělo zaručit různorodost vygenerovaných modelů. Každý algoritmus je spuštěn typicky jen s několika jedinci v populaci a na jednotky až desítky generací. Každý takový GP tedy vytvoří první generaci z určité podmnožiny neterminálů a následně proběhne několik generací GP, aniž by se jedinci hodnotili pomocí fitness funkce. Následuje výběr jedinců ke křížení. Jelikož není k dispozici žádné ohodnocení, je selekce prováděna pomocí ruletového kola, kdy se vždy zcela náhodně vyberou dva jedinci, kteří se mezi sebou budou křížit. Vždy je vybíráno ze všech jedinců v generaci, tudíž je možné, že někteří jedinci se budou křížit vícekrát a někteří se křížení nebudou účastnit vůbec. Operace křížení je realizována pomocí jednobodového křížení, kdy si dva vybraní jedinci mezi sebou vymění náhodné podstromy. Dále je na jedince aplikována mutace a následně je náhodně vybrán pouze jeden ze dvojice nově vzniklých potomků, který postoupí do následující generace.

Tímto způsobem je vygenerován požadovaný počet modelů, které mají formu syntaktického stromu. Tyto modely jsou následně vloženy do množiny terminálů a je spuštěn GP druhé fáze.

Zde již existuje fitness funkce, pomocí které lze ohodnotit každého jedince. Algoritmus probíhá, dokud není kladně vyhodnocena některá z ukončujících podmínek a za řešení problému je následně prohlášen nejlepší prvek z daného průběhu algoritmu.

Účelem této metody je dosáhnout rozmanitosti vstupních terminálů druhé fáze, díky které by mělo být jednodušší a rychlejší nalézt dostatečně kvalitní řešení. V ideálním případě by měla nová množina terminálů obsahovat částečná řešení daného problému a jejich vhodnou kombinací i s původními terminály by se mělo rychle a efektivně dosáhnout kvalitního řešení daného problému. Jak je ale zřejmé z popisu metody, působí zde velká porce náhody, tudíž není zaručeno, že některý z nově vygenerovaných modelů z první fáze, bude představovat toto částečné řešení. Právě proto jsou v množině terminálů přítomny i původní prvky, díky kterým je možné nalézt řešení i bez použití nových modelů.

6 IMPLEMENTACE

Tato kapitola popisuje implementaci vybraných metod a algoritmů při použití GP na problém symbolické regrese. Nejprve jsou popsány technologie, které je možné použít a jsou zde uvedeny některé knihovny pro usnadnění práce s vývojem algoritmů využívajících AI. Dále jsou detailněji popsány technologie a knihovny, které jsou použity v této práci.

Další část této kapitoly se věnuje detailnějšímu popisu implementace jednoduchého GP, který je nadále použit pro rozšíření z pohledu hlubokého genetického programování. Dále jsou uvedeny jednotlivé testovací sady dat, na kterých jsou testovány všechny metody, které zde budou uvedeny. Je tedy možné si utvořit obrázek o kvalitě jednotlivých algoritmů a metod a vyhodnotit jejich použitelnost, či možnosti pro zlepšení.

V posledních částech jsou popsány implementace nových metod, které jsou složené z několika vrstev. O těchto metodách se dá říci, že využívají hluboké genetické programování, a to právě proto, že hledání optimálního řešení daných regresních problémů je složeno z několika částí a rozděleno do několika vrstev.

6.1 Použité technologie

V dnešní době je možné naprogramovat jakýkoliv algoritmus využívající AI skoro v každém programovacím jazyce. Spousta z nich obsahuje mnoho metod nebo připravených knihoven, které tuto činnost velmi zjednoduší. Velké množství připravených knihoven (TensorFlow, Keras, NumPy) pro tvorbu neuronových sítí nebo genetických algoritmů je možné využít v jazyku Python.[35]

Mezi další jazyky, které podporují práci s umělou inteligencí, patří například Java se svými frameworky pro práci s genetickými algoritmy a GP Jenetics a Watchmaker, C++, R, Lisp, atd. Za zmínku stojí také program Matlab, kde díky svým velmi zjednodušeným a rychlým maticovým operacím je velmi jednoduché naprogramovat jakýkoliv algoritmus. Navíc Matlab obsahuje velmi dobře zpracované metody pro učení a použití neuronových sítí, čímž se může stát velmi dobrým pomocníkem při tvorbě algoritmů využívajících AI.[35]

V této práci jsou využity frameworky Watchmaker a Jenetics, které jsou připraveny pro programovací jazyk Java.

6.1.1 Watchmaker

Watchmaker je jednoduchý, výkonný a dobře rozšiřitelný framework pro tvorbu genetických nebo evolučních algoritmů napsaný v jazyce Java. Framework má otevřený zdrojový kód a lze ho volně stáhnout.[34]

Při vývoji pomocí watchmaker je možné využít připravené metody pro selekci, křížení, mutaci a další základní genetické operátory a zároveň je možné velmi jednoduše naprogramovat vlastní operátor nebo jakékoliv další rozšíření bez omezení. Zároveň watchmaker využívá výhod paralelismu, takže na víceprocesorových strojích je průběh algoritmu rychlý. Pro znázornění výstupu a statistických informací je možné využít několik implementovaných metod, které umožní lehce vygenerovat grafy, či různé statistické hodnoty z průběhu algoritmu.[34]

Jádrem celého frameworku je třída *EvolutionEngine*, pomocí které je spuštěn algoritmus. Před spuštěním se jako parametry třídy nastaví všechny potřebné genetické konstanty, operátory, ukončující podmínky a také fitness funkce a vstupní data. Jediné, co je tedy potřeba definovat je fitness funkce, zvolit podobu jedince, vhodné genetické operátory a vhodně nastavit genetické konstanty. Následně je možné algoritmus spustit. Za běhu je možné odchytávat různá statistická data pomocí *EvolutionMonitor* nebo *EvolutionObserver*. Výsledek je vrácen pomocí třídy *EvolutionResult*, která obsahuje nalezené řešení.[34]

Výhodou tohoto frameworku je bezpochyby jeho jednoduchost a možnost rozšíření. Bohužel zde není implementována podpora GP, kterou lze díky jednoduchosti rozšíření poměrně snadno doplnit.[34]

6.1.2 Jenetics

Jenetics je poměrně novější a modernější knihovna pro tvorbu nejen genetických algoritmů, ale obsahuje i třídy a metody pro GP. Obsahuje mnohem více operací než Watchmaker, ale bohužel na úkor jednoduché rozšiřitelnosti. Knihovnu Jenetics lze rozšířit jen velmi omezeně, a to hlavně o druhy genů a chromozomů a bohužel ani v tomto směru není zaručena stoprocentní volnost.[26]

Na druhou stranu je knihovna Jenetics připravena na řešení prakticky všech možných problémů týkajících se GA nebo GP. Pro vývoj metod hlubokého genetického programování je bohužel nutná volnost v rozšiřitelnosti knihovny, takže Jenetics bohužel není vhodný pro vývoj většiny takových vylepšení. U některých metod je uvedeno ekvivalentní řešení pomocí Jenetics pro ilustraci a porovnání výsledků.[26]

Knihovna Jenetics využívá pro jednotlivé kroky EA třídu *EvolutionStream*, která využívá *Java Stream*. Knihovny Jenetics obsahují velké množství různých genetických operátorů, které je možné využívat k různým druhům genů a chromozomů. Dále je možné využít několik druhů selekce, atd. I tato knihovna umožňuje nastavit velké množství genetických konstant a dále sledovat statistiky průběhu algoritmu. Ze statistik a nalezeného řešení je velmi jednoduché vytvořit několik grafů, které shrnou průběh algoritmu.[26]

6.2 Testovací data

Aby bylo vše jasně vidět, je pro testování jednotlivých metod zvoleno několik stejných sad dat, které jsou generovány podle určitých funkcí. Je tedy možné sledovat, jak se kterému algoritmu daří na určitých druzích funkcí. V následující tabulce jsou uvedeny všechny použité funkce a k nim pravidla pro vygenerování datové sady, jako je interval a celkový počet záznamů v datové sadě. V dalších tabulkách jsou tyto funkce označeny pouze pomocí pořadového čísla, které jednoznačně identifikuje funkci a datovou sadu v této tabulce.

Tabulka 5 - Seznam použitých testovacích datových sad

Pořadové číslo	Funkce	Generovaná datová sada	Počet záznamů v dat. sadě
1	$\sum_{i=1}^{x} \frac{1}{i}$	x ∈ {1, 2, 3, 4, …, 50}	50
2	$cos(2x)$	x ∈ <-5; 0,25; 5>	40
3	$\frac{1}{1+\frac{1}{x^4}} + \frac{1}{1+\frac{1}{y^4}}$	x, y ∈ (-5; 0,4; 5)	625
4	$3x + 18y - 1$	x∈(-20; 3; 20) y∈(-20; 4; 20)	140
5	$(1 + x^{0,5} + y^{-1} + z^{-1,5})^2$	x, y, z ∈ (1; 1; 6)	216
6	$2\sin(x)\cos(y)$	x, y ∈ (1; 0,5; 6)	121
7	$\frac{(x-3)^4 + (y-3)^3 - (y-3)}{(y-2)^4 + 10}$	x, y ∈ (0; 0,6; 6)	121

Za zmínku stojí funkce *cos(2x)*, která může vypadat jednoduše k vyřešení pomocí GP. Opak je pravdou, jelikož v neterminálech všech testovacích sad se nevyskytuje funkce cosinus, tudíž je GP nucen hledat alternativní cestu pro nalezení optimálního řešení této datové sady. Každá datová sada disponuje funkcí sinus, proto by nejlepším výsledkem bylo $cos(x) = 1-sin^2(x)$.

Dále jsou také pro všechny datové sady, a tím i pro všechny experimenty použité stejné genetické konstanty a stejné množiny neterminálů. Množiny terminálů jsou rozšířeny pouze o potřebné hodnoty představující proměnnou hodnotu (x, y, z) pro jednotlivé funkce.

Vyhodnocení experimentů s jednotlivými testovacími sadami je realizováno nejprve souborem se statistikami, který je vygenerován a uložen do příslušné složky s názvem testovacích dat a pořadovým číslem experimentu. Následně je vygenerován graf průběhu jednoho pokusu, který znázorňuje vývoj fitness hodnoty napříč průběhem algoritmu.

Pro každou datovou sadu je spuštěno 25 pokusů algoritmu. Nejlepší jedinci těchto pokusů jsou následně zobrazeni v celkovém grafu s názvem *EvolutionExperiment*, který je uložen v každé složce pro každou datovou sadu zvlášť. Jako poslední souhrnný graf je vygenerován boxplot, který zobrazuje nejlepší jedince u všech testovaných datových sad.

6.3 Jednoduché GP Watchmaker

Jelikož framework Watchmaker neobsahuje podporu pro GP, je nutné doplnit několik tříd tak, aby bylo možné takový algoritmus spustit.[34] Základním rozdílem mezi GA a GP je v podobě jedinců. Jedinec v GP je znázorněn pomocí syntaktického stromu, což je první věc, kterou je nutné naprogramovat. Jelikož každý uzel stromu může mít až *n* potomků, je nutné implementovat třídu *Tree* jako k-cestný strom, kde každý uzel obsahuje seznam potomků.

Dalším rozdílem je, že jednotlivé uzly stromu mohou představovat buď terminály, tedy číselné hodnoty, nebo proměnné, anebo neterminály, tedy funkce. Proto je nutné vytvořit společného předka obou druhů uzlů, v tomto případě *EquationMember*, který obsahuje nejdůležitější metodu pro následné vyhodnocení stromu a to *compute*, kterou dědí všechny druhy uzlů stromu. Tato metoda je volána při dosazení trénovacích dat do syntaktického stromu v případech, kdy se vyhodnocuje kvalita jedince pomocí fitness funkce. Dále se tedy uzly stromu dělí do dvou větví, a to *TerminalMember* a *FunctionMember*. Následně jsou tyto třídy specifikovány díky vytvoření jejich potomků, například *VariableMember* nebo *OneParameterFunction*, podle potřeby využití.[34]

S vytvořením nového druhu jedinců souvisí i úprava genetických operátorů. V tomto případě je možné zanechat stávající logiku určenou pro GA, jelikož je stejná jako v GP. Jediné, co je potřeba, je přetížit volání důležitých metod pro nový druh jedince. Pro křížení je tedy vytvořena třída *TreeCrossover*, která dědí z *AbstractCrossover* metodu *mate*. Rodičovská třída se tedy parametrizuje pro jedince typu *Tree* a v přetížené metodě *mate* se implementuje logika křížení. V tomto případě je zvolena výměna náhodných podstromů u obou rodičů. Stejným způsobem jsou implementovány i další genetické operátory jako je mutace, jen s rozdílem, že pro tyto operátory je využita rodičovská třída *EvolutionOperator* a přetížená metoda *apply*. V tomto GP je použita mutace, a to formou křížení s novým náhodně vygenerovaným stromem.[34]

Po implementaci genetických operátorů zbývá vyrobit fitness funkci, díky které se dosadí do syntaktického stromu testovací data, a provede se jeho výpočet. Tato logika je implementována ve třídě *TreeEvaluator,* která je potomkem třídy *FitnessEvaluator*.[34] Při spuštění výpočtu se pro každého jedince v dané generaci spustí metoda *getFitness*. Tato metoda postupně spočítá chybu pro každou sadu testovacích dat a následně vrátí celkovou chybu pro daného jedince.

Proces výběru jedinců, kteří postupují ke křížení, tedy selekci, je možné využít již připravenou. Framework watchmaker nabízí celou řadu selekčních strategií, které se ukrývají pod rozhraním *SelectionStrategy*. Je možné využít základní, jako ruletové kolo, nebo turnajové schéma, ale i další ne tak známé strategie (*TruncationSelection, SigmaScalling*). V úpravě pro GP je vytvořena nová selekční strategie *TreeTournamentSelection*, která je implementována z důvodu zachycení statistických výsledků ze selekce, a to hlavně časových statistik. Tato třída se odkazuje na původní třídu *TournamentSelection*, která je implementována v rámci frameworku watchmaker.[34]

Poslední třídou, kterou je nutné implementovat je mechanismus, který vygeneruje počáteční generaci jedinců. Tato část je implementována ve třídě *TreeFactory*, která dědí vlastnosti z třídy *AbstractCandidateFactory*. Jediným úkolem této třídy je vygenerovat náhodného, ale smysluplného jedince, který je umístěn do počáteční generace.[34] Smysluplný jedinec je myšlen z pohledu pravidel v syntaktickém stromu, kdy v listech jsou vždy terminály a v ostatních uzlech neterminály. Dále je vhodné dbát pravidel, kdy stromy v počáteční generaci by měly mít malou hloubku a počet uzlů. Tyto hodnoty jsou určeny jako genetické

konstanty definované ve třídě *GeneticParameters* a při generování stromů je nutné dbát na maximální hloubku stromu.

Díky nově vytvořeným třídám je možné sestavit *EvolutionEngine* pro GP řešící symbolickou regresi. V první řadě je ovšem nutné připravit vstupní trénovací data a sestavit množiny možných terminálů a neterminálů. Následně je možné vytvořit a spustit instanci vytvořeného GP. Pro ilustraci je sestavení a spuštění znázorněno v Přílohy, která obsahuje kód, který je nutný pro spuštění GP.

Po získání výsledku z *EvolutionEngine* jsou vyhodnoceny statistiky z průběhu experimentu, tak, jak je vidět v Přílohy. V první řadě jsou vidět genetické konstanty, se kterými je algoritmus spouštěn, tedy nastavení všech pravděpodobností, počtu jedinců v populaci a parametry ukončujících podmínek. Dále jsou vypsány všechny terminály a neterminály, pomocí kterých je výsledek vygenerován. Následuje část s genetickými výsledky, kde je vypsán vítězný, tedy nejkvalitnější jedinec, který byl v průběhu algoritmu vygenerován a následně statistiky z průběhu algoritmu. V části časové statistiky se může zdát matoucí údaj *fitnessTime*, který udává, kolik času trvalo ohodnotit jedince v průběhu algoritmu. Většinou je časový údaj mnohonásobně větší než celková doba trvání algoritmu, tedy *algoTime*. To je dáno tím, že je to pro některé jedince časově náročná operace a watchmaker pro výpočet hodnoty fitness používá více vláken. Tudíž je možné hned zjistit, kolik se tímto vylepšením ušetřilo času při průběhu algoritmu.

6.4 Jednoduché GP Jenetics

Při použití knihovny Jenetics pro řešení GP není třeba rozšiřovat funkčnost, jako tomu je u watchmakeru. Je tedy potřeba vytvořit objekt *Engine*, pomocí kterého se spouští celý algoritmus. Jedná se o generický objekt a právě pomocí genericity se určuje podoba chromozomu a jeho genů. V případě sestavení pro použití v GA je možné vybrat ze široké škály chromozomů a jejich genů, například *BitGene* nebo *DoubleGene*. Pro GP je nutné použít *ProgramGene*. Pomocí těchto genů je možné složit syntaktický strom složený z terminálů a neterminálů, který je potřebný pro použití GP.[26]

Pomocí objeku *Engine* se následně sestavuje *EvolutionStream*, který určuje parametry a podmínky běhu algoritmu. V první řadě je nutné určit *builder*, který obsahuje testovací data, možné hodnoty genů a fitness funkci. V případě GP se jedná o objekt *Regression*, který obsahuje množiny neterminálů, terminálů, omezení velikosti syntaktických stromů při prvotním generování i při křížení, fitness funkci a metodu výpočtu chyby, nejčastěji

metodu nejmenších čtverců. Následně musí být do streamu zapojeny strategie výběru, křížení a následně další genetické operátory, kterých jenetics nabízí velké množství. Sestavením objektu *Engine* je vytvořen proud událostí, který se opakuje, dokud není splněna některá z ukončujících podmínek algoritmu a není vrácen nejlepší jedinec.[26]

Nejlepší jedinec je vrácen do objektu typu *EvolutionResult* po spuštění příkazu *engine.stream().limit().collect()*. Ve funkci *limit* se uvádí ukončující podmínky algoritmu a díky *collect* je vrácen nejlepší jedinec.[26]

Sestavení jednoduchého GA, nebo GP pomocí knihoven jenetics je dobře popsáno v manuálu, který je volně dostupný na oficiálních stránkách jenetics.io. Dále zde jsou popsány další možnosti využití nebo vylepšení algoritmů a jsou zde i popsány jednotlivé balíčky, třídy a metody, které je možné využít. Příklad sestavení celého algoritmu je vidět v Přílohy.[26]

Nejdříve je vytvořen objekt *Regression*, který na vstupech přijímá terminály, neterminály, omezující podmínky velikostí stromu při generování a křížení, ztrátovou funkci a pole *Sample*, které obsahuje datové body s očekávaným výsledkem. Toto pole zde nahrazuje fitness funkci, pomocí vstupních hodnot se vypočítá výsledek modelu a následně se odečte od očekávaného výsledku z pole. Následně je vytvořen *Engine*, kde je definována velikost populace, elitismus, metody křížení, selekce a mutace a popřípadě další genetické operátory. Důležitý je parametr *minimizing*, který určuje, že chceme hodnotu fitness minimalizovat co nejblíže nule. V poslední řadě se vytváří *EvolutionResult*, kde jsou definovány ukončující podmínky, dále je možné připojit i statistiky, jak je vidět i v Přílohy.[26]

6.5 MultiEngine metoda

Jak již bylo řečeno, metoda MultiEngine v první fázi vytváří nové terminály, které následně používá při hledání optimálního řešení daného problému. Je tedy nutné vytvořit mechanismus, který dokáže převést syntaktický strom na vstupní terminál. Po provedení této konverze je spuštěn klasický GP s novou množinou terminálů. Metodu MultiEngine tak, jak je navržena v předchozí kapitole, je vhodné implementovat pomocí frameworku watchmaker, kde je možné více experimentovat s průběhem algoritmu a s podobou vstupních množin. Lze však využít i framework jenetics, kde jsou již připravené vhodné objekty pro MultiEngine metodu. Pro vyhodnocení je tato metoda spuštěna několikrát a je sledován vývoj fitness u jednotlivých experimentů. Tato kapitola tedy obsahuje popis implementace pomocí jednotlivých frameworků a následně vyhodnocení experimentů.

6.5.1 Implementace pomocí watchmaker

Jelikož Framework watchmaker neobsahuje řešení GP, je nutné využít a rozšířit implementovanou logiku z kapitoly 6.2. Dále je nutné vytvořit nový terminál *TreeMember*, který bude obsahovat nově vytvořený terminál typu syntaktický strom. Je nutné zajistit, aby bylo možné vypočítat odezvu syntaktického stromu. Každý terminál musí obsahovat metodu *compute*, která vrátí patřičnou hodnotu (náhodná hodnota *int*, konstanta, proměnná x, …). V případě terminálu, který obsahuje syntaktický strom, je nutné implementovat celou logiku výpočtu stromu tak, jak je implementována při získávání odezvy jedince. Z implementace je tedy jasné, že se tento terminál obsahující syntaktický strom tváří jako jeden prvek jedince, který v průběhu algoritmu nelze rozdělit genetickými operátory. Jelikož se jedná o terminál, bude vždy umístěn v listu stromu jedince.

Dále je nutné implementovat algoritmus, který vygeneruje tyto nové terminály. K tomu je využitý neúplný GP, kde není použita žádná fitness funkce. Prvky se tedy vybírají a kříží na základě náhody a v žádné části algoritmu není prováděno žádné ohodnocování. Pomocí genetických parametrů je nastaven počet generací a počet jedinců obsažených v populaci. V prvním kroku algoritmus vygeneruje náhodné stromy s omezenou hloubkou a následně probíhá cyklus, kde se střídají operace selekce, křížení a mutace. Selekce je prováděna pomocí ruletového kola, kdy se náhodně vyberou dva jedinci. Takto jsou vybrány dvojice, které se v následujícím kroku pokříží a ze vzniklé nové dvojice se vybere jeden náhodný jedinec, který postoupí do další generace. Operace křížení je realizována pomocí jednobodového křížení, kdy se vymění dva náhodné podstromy u vybraných jedinců. Následně je s velmi malou pravděpodobností aplikována na nového jedince operace mutace, kdy je vygenerován nový náhodný strom.

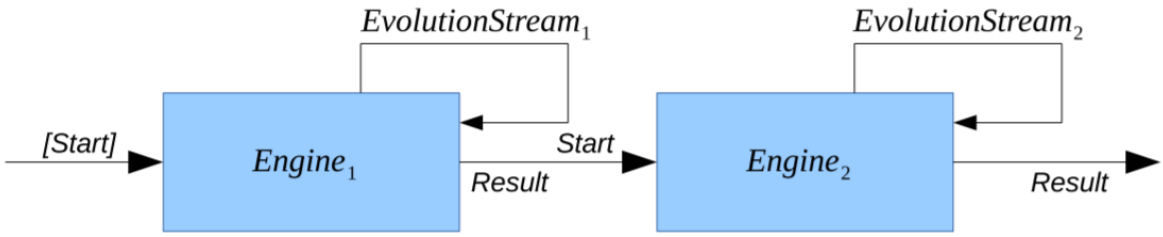
Pomocí tohoto cyklu je vygenerován zadaný počet nových terminálů, které jsou transformovány do objektu typu *TreeMember* a vloženy do množiny terminálů. Následně je spuštěn klasický GP, který hledá optimální řešení daného problému.

6.5.2 Implementace pomocí Jenetics

Jak již bylo několikrát řečeno, jenetics neumožňuje tak rozsáhlá rozšíření, jaká jsou nutná k implementaci této metody. Jelikož tato knihovna nabízí opravdu velké možnosti, obsahuje i alternativy metody MultiEngine, které jsou ji velmi podobné. Jedná se o alternativy k vytvoření objektu *Engine,* a to *ConcatEngine* a *CyclicEngine*. Jejich použití je velmi

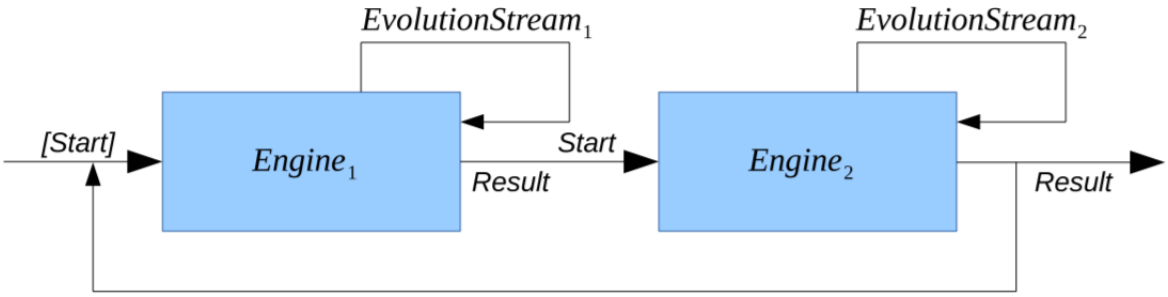
jednoduché, a to že vytvoříme více klasických *Engine*, které následně vstupují jako parametry do rozšířeného *ConcatEngine* nebo *CyclicEngine*.[26]

ConcatEngine složí dohromady dva *Engine* s různými parametry. V první řadě se spustí první *Engine* a po jeho ukončení se jeho výsledku použijí na vstup druhého. Oproti klasickému GP je zde nutné přidat limity, tedy ukončující podmínky do obou *Engine*. Tyto podmínky by měly být u každého *Engine*, jinak by algoritmus nikdy neskončil, ovšem zde je to podmínkou. Využití tohoto modelu je vhodné na problémy, kde je dobré udělat v první řadě širší průzkum a v druhém kroku jej specifikovat.[26]



Obrázek 21 - Model *ConcatEngine[26]*

CyclicEngine je skoro stejný jako předcházející *ConcatEngine*. Jediným rozdílem je, že *ConcatEngine* se ukončí po proběhnutí obou *Engine*, kdežto *CyclicEngine*, pokud zatím nenašel optimální řešení, pokračuje znovu od začátku tak, jak je vidět na obrázku 23. Využití tohoto modelu je stejné, jako u předchozího *ConcatEngine*. Zde je ale možné proces opakovat dokud není nalezen vhodný výsledek.[26]



Obrázek 22 - Model *CyclicEngine[26]*

6.6 Metodika vyhodnocení výsledků

Vyhodnocení výsledků je každé z uvedených metod stejné, tudíž každá metoda implementuje určité operace, které generují výsledky algoritmu, statistické výsledky každého pokusu a následně souhrnné statistické výsledky pro účely porovnání. Každá metoda se spouští

několikrát pro každou uvedenou datovou sadu. Pro každý takový pokus je vygenerovaný spojnicový graf, který zobrazuje vývoj hodnoty fitness v daném pokusu. V každé generaci je tedy uložena hodnota fitness nejlepšího jedince a po ukončení pokusu je vygenerován graf těchto hodnot. Spolu s tímto grafem je vygenerován textový soubor, který obsahuje statistické údaje. Tento soubor obsahuje počáteční nastavení algoritmu, výsledného nejkvalitnějšího jedince a jeho fitness hodnotu a následně statistické údaje z průběhu algoritmu. Příklad takového souboru lze vidět v Přílohy.

Pro každou datovou sadu je spuštěno několik pokusů tak, aby byla možnost nalézt co nejkvalitnější řešení. Pro každý takový pokus je uchován nejlepší jedinec. Tento jedinec je jeden z potencionálních nejlepších jedinců pro danou datovou sadu. Po ukončení všech pokusů je vygenerován boxplot graf, který zobrazuje variabilitu jedinců pro všechny pokusy na dané datové sadě. Tyto hodnoty jsou po ukončení všech pokusů na všech datových sadách seskupeny do společného boxplot grafu, kde je datová sada určena pořadovým číslem.

Poslední fází získání výsledků je zobrazení souhrnných výsledků pro všechny testované metody, tedy jednoduchou SR ve watchmaker a jenetics a metodu MultiEngine. Pro každou metodu jsou vygenerovány textové soubory, kde jsou uvedeny celkové časové statistiky pro každou datovou sadu, nejlepší jedinci a jejich fitness hodnoty. Následně jsou fitness hodnoty shrnuty v boxplot grafu, kde jsou srovnány všechny metody.

7 EXPERIMENTY

Kapitola obsahuje popis a vyhodnocení jednotlivých experimentů. Experimenty jsou navržené tak, aby otestovaly kvalitu a schopnost nalézt optimální řešení při různém nastavení genetických konstant nebo při použití, případně vynechání, různých neterminálů. Ke každému experimentu je tedy uveden důvod jeho použití, počáteční nastavení pomocí genetických konstant, dále jsou uvedeny použité množiny terminálů a neterminálů a nakonec jeho vyhodnocení.

Pro všechny experimenty jsou použity stejné datové sady uvedené v tabulce 5, a to z důvodu porovnání kvality jednotlivých výsledků. Výsledky jsou zobrazeny v tabulkách, kde je uveden nejlepší jedinec a navíc časový údaj, který udává, jak dlouho trvalo nalezení řešení. Dále je zobrazen jeden průběh vývoje hodnoty fitness při hledání jednoho řešení a v poslední řadě boxplot graf, který udává variabilitu při řešení jednotlivých datových sad.

Z hlediska průkaznosti statistických údajů je každý experiment spuštěn na pevný počet generací 1000, a to i v případě, že je optimální řešení nalezeno dříve.

7.1 Experiment č. 1

Jedná se o základní experiment, se kterým můžeme následující porovnávat. Množina neterminálů obsahuje všechny potřebné elementy, které jsou obsaženy i v předpisech funkcí, podle kterých jsou vygenerovány datové sady. Množina terminálů obsahuje náhodné celé číslo a proměnné. Počet proměnných je dán dle potřeby dané datové sady, tudíž pokud je datová sada utvořena podle dvourozměrné funkce, tak množina terminálů obsahuje taktéž dvě proměnné.

7.1.1 Počáteční nastavení

Následující tabulka zobrazuje počáteční nastavení experimentu. Nastavení všech spouštěných metod probíhá právě s uvedenými parametry, ukončující podmínkou 1000 generací a s uvedenými množinami terminálů a neterminálů. Množina neterminálů obsahuje všechny potřebné funkce, které jsou použity při generování testovacích datových sad.

Tabulka 6 - Experiment 1 - Počáteční nastavení

Genetická konstanta	Hodnota
Počet generací	1000
Velikost populace	100
Max. hloubka stromu při inicializaci	3

Maximální velikost stromu při křížení	30
Pravděpodobnost křížení	0,9
Pravděpodobnost mutace	0,1
Pravděpodobnost turnajového schématu	0,9
Počet elitních jedinců	1
Počet generací preevoluce	5
Počet jedinců v preevoluci	10
Maximální velikost stromu preevoluce	5
Maximální hloubka stromu při inicializaci preevoluce	2
Terminály	Náhodné celé číslo z intervalu <0-10>, proměnná x, y, z dle potřeby datové sady
Neterminály	+, -, *, /, ^, sin, cos, sqrt

Běh tohoto experimentu obsahuje spuštění třech uváděných metod řešení GP a to jednoduchý GP pomocí watchmaker a jenetics a následně metoda multiengine pomocí watchmaker. Všechny metody posupně spustí 25 pokusů pro každou datovou sadu uvedenou v tabulce 5. Pro každý takový pokus je v kořenovém adresáři uložen graf, který zobrazuje vývoj fitness ohodnocení nejkvalitnějších jedinců ve všech generacích. Následně je pro každý tento pokus vygenerován textový soubor, který obsahuje vypsané počáteční nastavení, výsledky a statistické hodnoty průběhu. Po dokončení všech 25 pokusů na jedné datové sadě je uložen graf, který zobrazuje nejkvalitnější jedince z jednotlivých pokusů.

Pro každou datovou sadu jsou nadále uchovány statistické údaje, které jsou následně použity pro nalezení nejlepších jedinců, průměrných časových údajů, které se týkají trvání jednotlivých pokusů a na závěr k vygenerování boxplot grafu. Boxplot srovnává všechny metody pro všechny datové sady.

V následujících podkapitolách jsou popsány výsledky uvedených metod. Grafy a výsledky týkající se konkrétní datové sady jsou vždy uvedeny podle datové sady číslo 7 uvedené v tabulce 5.

7.1.2 Vyhodnocení jednoduché GP watchmaker

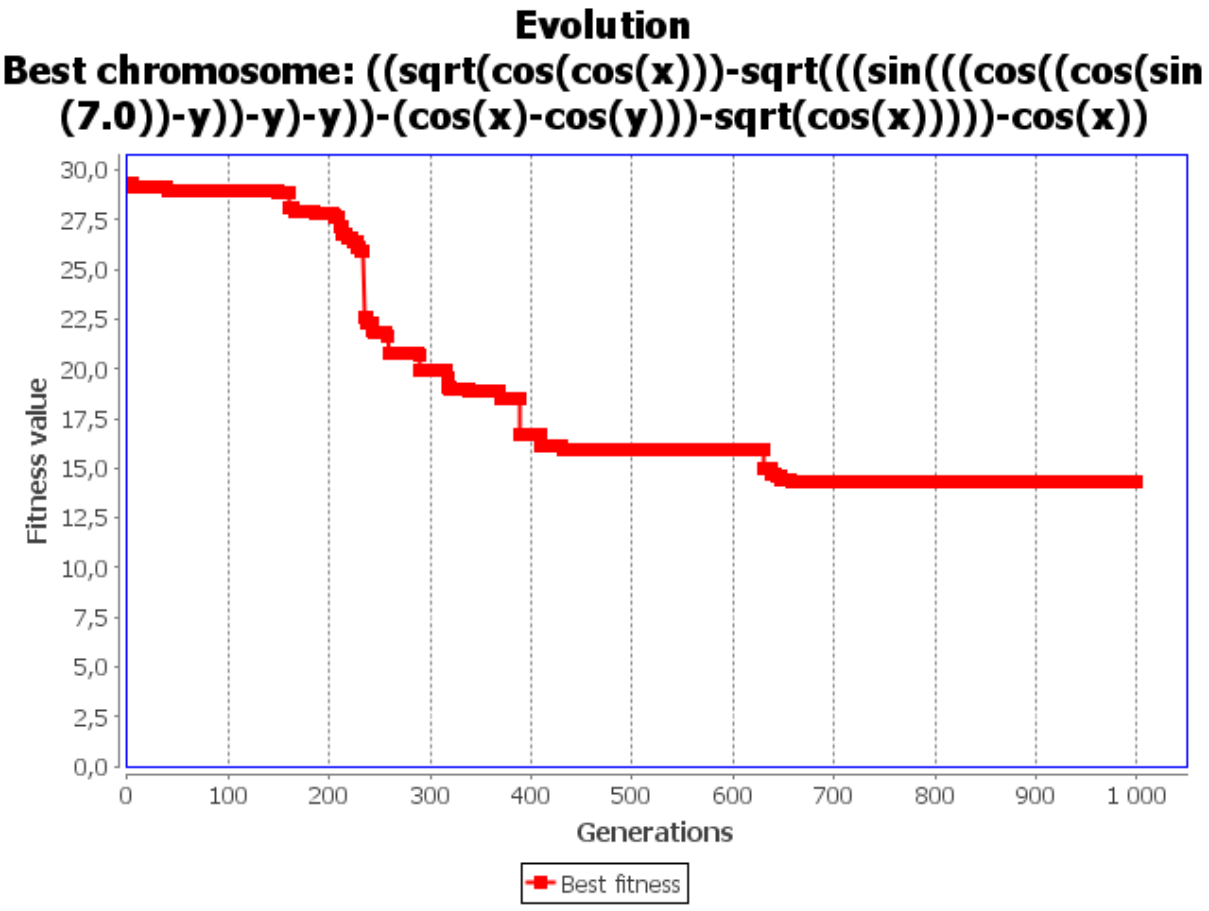
Vyhodnocení jednoduchého GP naprogramovaného pomocí frameworku watchmaker probíhá díky spuštění algoritmu s počátečním nastavením pro daný experiment. Algoritmus hledá optimální řešení pro datové sady uvedené v kapitole 6.2. Nejlepší nalezené řešení jsou

uvedeny v následující tabulce. Každá datová sada je identifikována pomocí pořadového čísla. Následné hodnoty udávají nejlepšího nalezeného jedince, jeho fitness hodnotu a průměrný čas dokončení hledání na konkrétní datové sadě. Podobu jedinců je možné zjednodušit pomocí matematických operací na jednodušší rovnici, ovšem zde jsou jedinci z ilustračního důvodu popsáni tak, jak je vygeneroval algoritmus GP.

Tabulka 7 – Experiment č. 1 - Vyhodnocení jednoduchého GP ve watchmaker

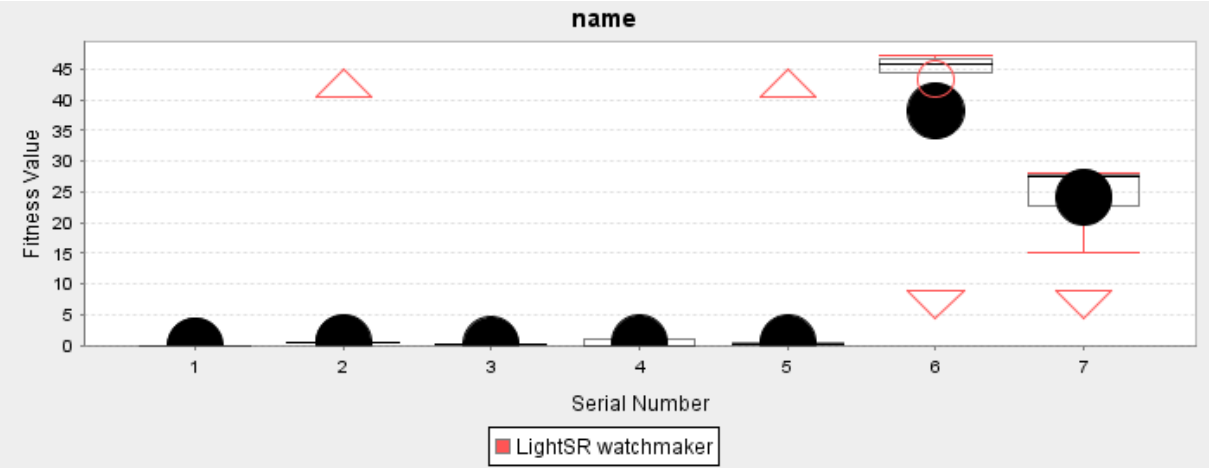
Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	cos(sqrt(sqrt(sqrt(sin(sqrt(sqrt(sqrt(sqrt(sin(4.0))))))))))	0	2,37
2	cos(cos(((1.0+4.0)-7.0)))/((cos(cos((2.0-0.0)))-(6.0+cos(0.0)))-3.0)	0,52	12,2
3	cos((sin((y/cos(cos(0.0)))))/y)) + cos((cos(sqrt(sin(cos(cos(sin((cos(2.0) / cos(cos(4.0)))))))))/x))	0,003	25,09
4	(((((y+(((y+y)+x) + ((6.0*y)+y)))+x)+x)+y) + (y+y))+(y+(y-5.0))	0	7,08
5	(((9.0+((((9.0+x)+sin((x+(sin(y)+4.0)))) + 6.0)/y))/z)+x)+((z/y)+x)	0,26	9.01
6	(sin(x)/cos((x-x)))/(cos(y)/3.0)	0	2,22
7	((cos(cos(cos(x)))-sqrt(((cos(cos(sqrt((cos(y)-6.0))))-sqrt((cos(y)-cos(x))))-cos(x))))+(cos(0.0)-cos(x)))	15,53	5,09

Následující graf popisuje vývoj ohodnocení nejlepšího jedince v každé generaci při hledání optimálního řešení podle datové sady číslo 7. Z grafu je možné vyčíst, že hodnota fitness a tedy i nejlepší jedinec byly nalezeni přibližně po 650 generacích algoritmu. Hodnota se ustálila přibližně na čísle 15. Jak je možné vidět z předešlé tabulky výsledků, jedná se o nejlepšího nalezeného jedince pro tuto datovou sadu. Ve zbylých generacích již tento jedinec nebyl překonán a jeho nezjednodušenou podobu je možné vidět v názvu grafu.



Obrázek 23 – Experiment č. 1 - Vývoj fitness jednoduchá SR watchmaker

Potvrzením, že se jedná o nejlepšího jedince, může být následující graf, který zobrazuje nejlepší jedince nalezené v každém z 25 pokusů provedených na všech datových sadách. Pro tyto účely je využitý boxplot, který znázorní variabilitu výsledků u jednotlivých datových sad. Z grafu je patrné, že u prvních datových sad je většina výsledků velmi blízká nule. Naopak poslední dvě datové sady jsou pro řešení složitější a tomu odpovídají i průměrné výsledky. Ovšem i u těchto metod se nejlepší výsledky dají považovat za kvalitní.



Obrázek 24 – Experiment č. 1 - Výsledky datových sad jednoduchá SR watchmaker

7.1.3 Vyhodnocení jednoduché GP jenetics

Tento pokus vyhodnocuje výsledky získané z hledání optimálního řešení při použití datových sad uvedených v tabulce 5 a při počátečním nastavení algoritmu, které je uvedené v tabulce 9. Výsledky získané při tomto pokusu jsou vidět v následující tabulce, která ke každé datové sadě označené pořadovým číslem uvádí nejlepšího nalezeného jedince ve zjednodušeném tvaru, ohodnocení nejlepšího jedince pomocí fitness funkce a průměrný čas trvání jednoho pokusu na dané datové sadě.

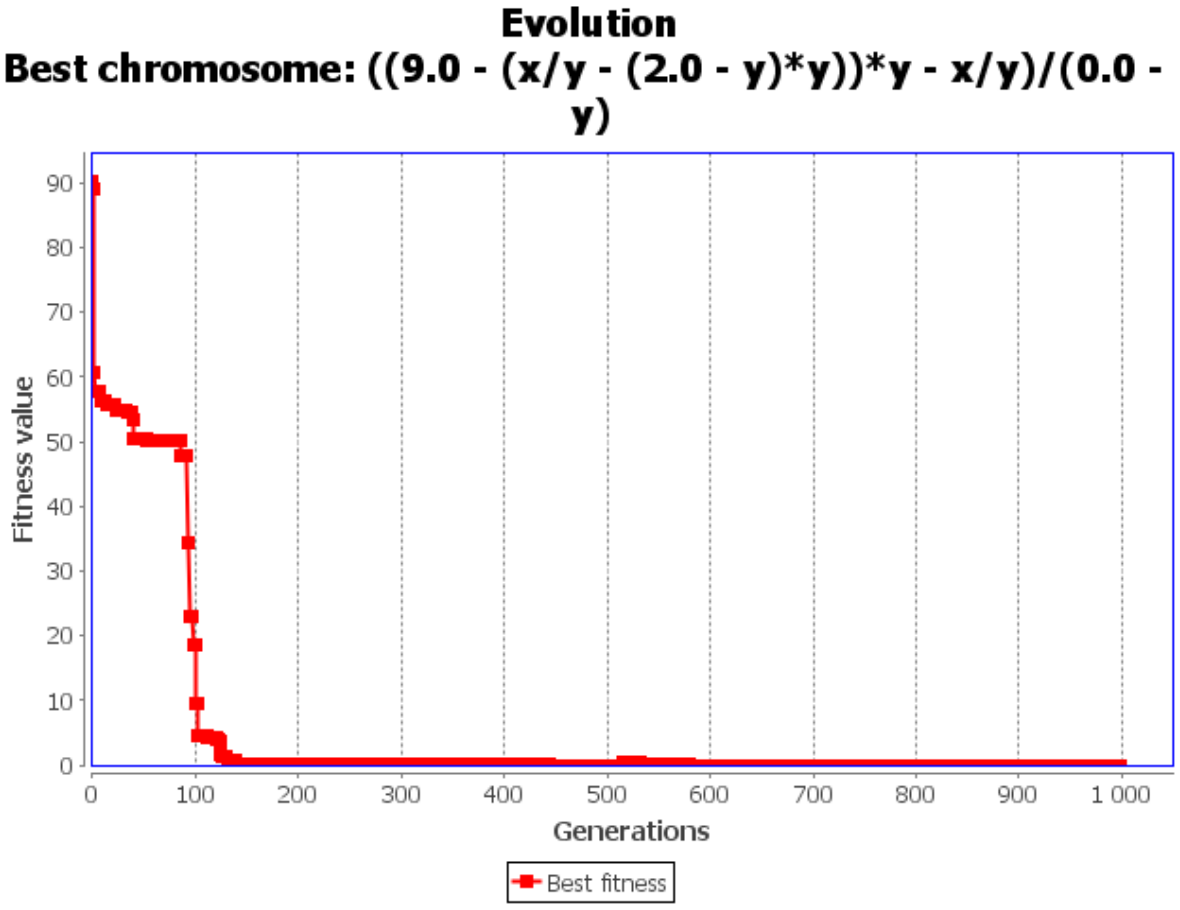
Tabulka 8 - Experiment č. 1 - Vyhodnocení jednoduché SR jenetics

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	sqrt(x/x^6.0)	0	0,62
2	cos(1.0/(2.0*x))	0	0,95
3	cos(exp(cos(exp(cos(6.0^y))))/y) + cos(exp(y))^(y*(sqrt(cos(4.0^(6.0^y)))*cos(y)))	$5{,}96*10^{-11}$	1,04
4	(x + ((x - 1.0) + x)) - ((x*y + y) + y)	0	1,3
5	((5.0 + z)/z + x)/z + 9.0	$3{,}14 * 10^{-4}$	1,42
6	y/cos(y) + ((1.0 + (y - cos(y - 0.5403023058681398))) + (y - sin(sqrt(x))))	$2{,}43 * 10^{-6}$	1,24
7	sqrt((x^(x^x))^(y^2.0 - x^(y^x)))/y - (4.0 + ((6.0 + x) + y))	$1{,}92 * 10^{-5}$	1,37

Z výsledků je patrné, že se jedinci poměrně liší od původních předpisů, podle kterých byly vygenerovány datové sady. Ovšem jejich ohodnocení ukazuje na poměrně kvalitní řešení.

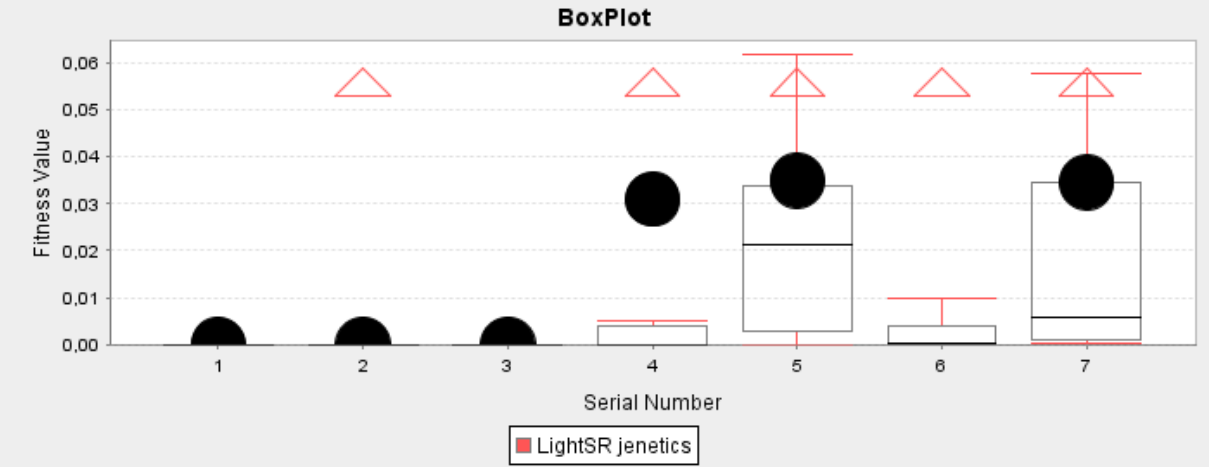
Jenetics umí zjednodušit nalezeného jedince, tak, aby byl lépe čitelný, ovšem po použití dalších matematických úprav, by se tyto jedince dalo ještě zjednodušit. Tento fakt by mohl vést k přiblížení výsledků k původním předpisům datových sad. Pokud by ovšem ani v tomto případě neseděla předloha a výsledek, bylo by jasné, že množina trénovacích dat nebyla dostatečně obsáhlá k dosažení potřebného výsledku. Přesnější předpisy by tedy mohly být nalezeny v případě, že budou datové sady obsahovat více trénovacích dat, což se bohužel projeví záporně na výsledném času trvání algoritmu.

Jelikož je v pokusech implementován elitismus, vývoj fitness ohodnocení nejlepšího jedince v každé generaci by měl mít sestupný charakter, což potvrzuje následující graf. Je použit graf vygenerovaný při hledání optimálního řešení pomocí datové sady číslo 7. Je vidět, že velmi dobré řešení je nalezeno velmi brzo a to přibližně kolem sté generace. Následné zlepšení je již mírné, ovšem výsledná hodnota je velmi blízká nule, což je dobré. V porovnání s jednoduchou SR ve watchmaker, dosahuje jenetics lepších výsledků. Tento fakt je dán tím, jak již bylo několikrát zmíněno, že jenetics je výkonnější a lépe uzpůsobený nástroj pro řešení GP.



Obrázek 25 - Experiment č. 1 -Průběh fitness jednoduchá SR jenetics

Dle následujícího grafu lze porovnat výsledky jednoduchého GP v jenetics u všech datových sadách. Každý sloupec obsahuje 25 hodnot, které představují nejlepší ohodnocení jedinců v jednotlivých pokusech. Z grafu vyplívá, že většina výsledných jedinců skončilo s ohodnocením velmi blízko nule, čímž je dokázána kvalita frameworku jenetics.



Obrázek 26 - Experiment č. 1 - Výsledky datových sad jednoduchá SR jenetics

7.1.4 Vyhodnocení metoda MultiEngine watchmaker

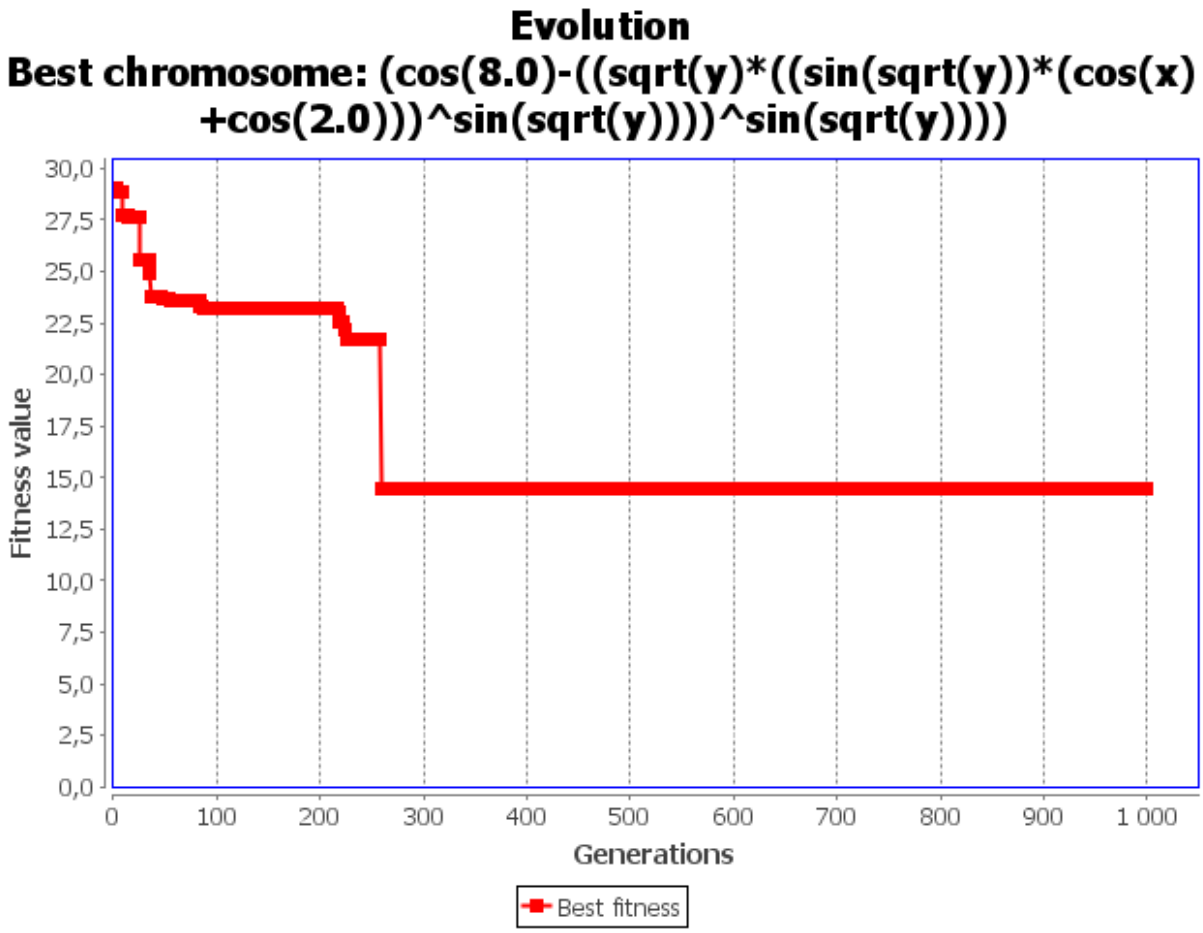
Tato kapitola popisuje výsledky nashromážděné po spuštění metody MultiEngine na všech testovacích sadách. Nashromážděné výsledky jsou prezentovány v následující tabulce a grafech. Tabulka popisuje nejlepšího jedince a jeho ohodnocení a následně průměrnou dobu trvání jednoho pokusu pro každou datovou sadu.

Tabulka 9 - Experiment č. 1 - Výsledky pro metodu MultiEngine

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	(5.0-sqrt(6.0))^(cos(x)-cos(5.0))	0	2,28
2	(x+x)*sqrt((sin(sin((cos(4.0)*cos(4.0)))) / cos(sin(sin(sin((cos(2.0) * sin(sin(sin(sqrt(cos(2.0)))))))))))	0.52	19,09
3	(sin(6.0)/y)+cos(((cos(((sin(6.0) * (sin(7.0)/y))/cos(9.0)))*sin(1.0))/x))	0.007	37,46
4	(((y*1.0)+(((x+(9.0+y)) * 4.0)+(y*9.0)))+5.0)-1.0	0	7,26
5	(x+(x+(7.0/y)))+((5.0+((x/y)/(y+y)))/z)	0.289	10,58
6	((((y+y)+(sin(x)+((x-x) + sin(x))))+((2.0/x)+((y/y)/y)))+sin(x))/cos(y)	0	5,27
7	(cos(y)+sin(y))-sqrt((x-sqrt((sqrt(cos((sin((sin(y)-y))-(cos(y)+sin(y)))))+cos(x)))))	9,46	7,21

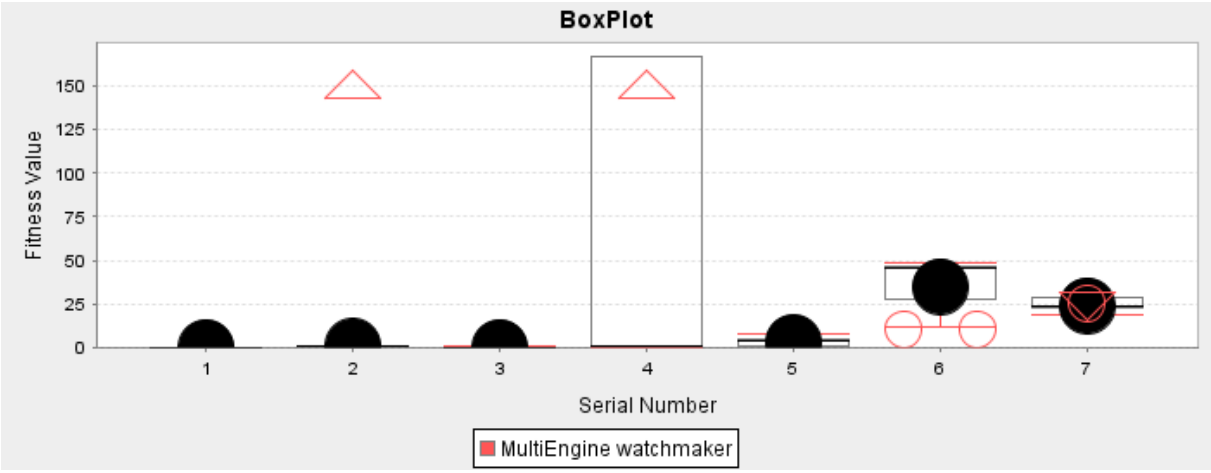
Z předešlé tabulky je vidět, že se oproti použití ostatních metod velmi liší hlavně časové údaje. To je z důvodu, že metoda MultiEngine využívá ještě preevoluci, která ovšem není tolik časově náročná jako následné vyhodnocování pomocí fitness funkce. Pro tuto metodu je to stěžejní část, která zabírá nejvíce času. Výsledná podoba jedinců nalezených díky metodě MultiEngine je velmi špatně čitelná. Po použití jednoduchých matematických úprav by se dalo tyto rovnice zjednodušit a v některých případech by potom mohli připomínat hledané vzory.

Průběh jednoho pokusu zobrazuje následující graf. Jedná se o pokus hledání optimální řešení za použití datové sady číslo 7. Je vidět, že nalezení nejlepšího řešení bylo přibližně v generaci číslo 250 a následně už lepší jedinec nebyl nalezen. Tento údaj ukazuje, že pomocí metody MultiEngine dochází k nalezení řešení dříve, než při použití jednoduchého GP ve watchmaker, bohužel také dokazuje, že jenetics i v tomto případě dosahuje lepších výsledků.



Obrázek 27 - Experiment č. 1 - Průběh fitness MultiEngine

Výsledky získané pomocí metody MultiEngine jsou zobrazeny v následujícím boxplot grafu. Každý box obsahuje nejlepší výsledky pokusů získaných při hledání optimálního řešení podle konkrétní datové sady.



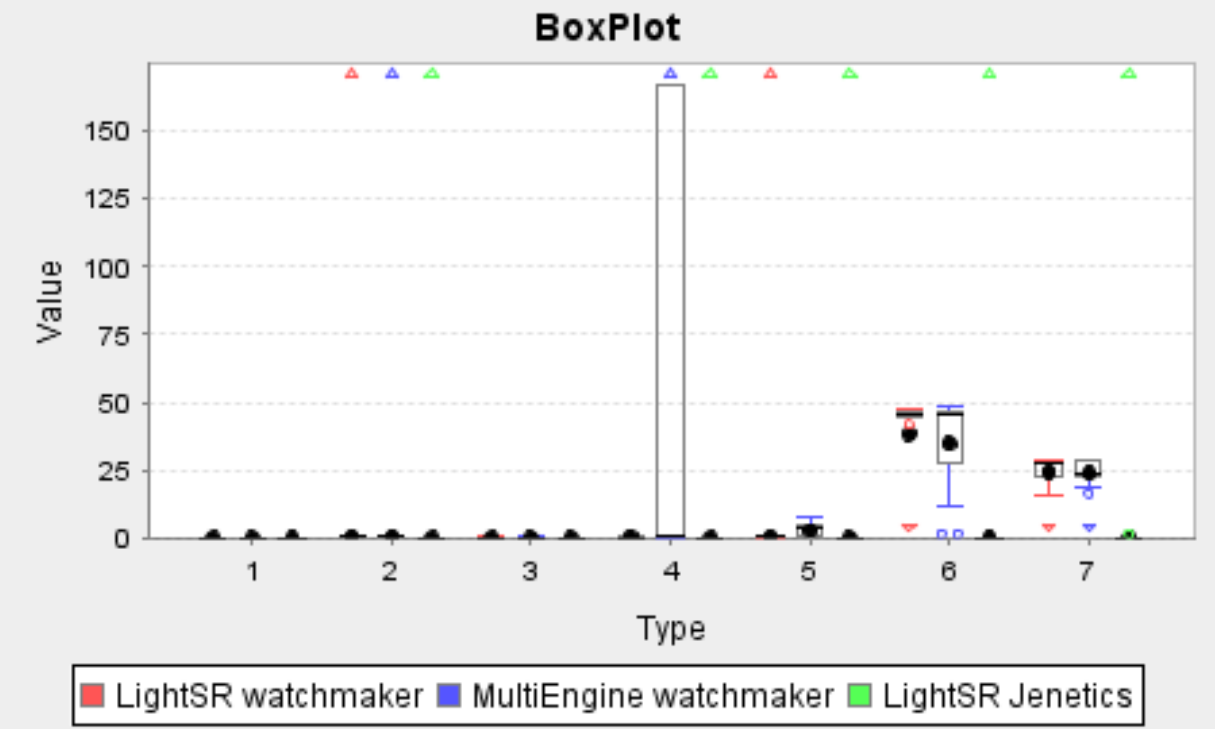
Obrázek 28 - Experiment č. 1 - Vyhodnocení datových sad metoda MultiEngine

7.1.5 Vyhodnocení experimentu číslo 1

Tato kapitola obsahuje porovnání všech tří metod mezi sebou při použití stejného počátečního nastavení všech algoritmů. Výsledky jsou prezentovány pomocí boxplot grafu, který obsahuje výsledky ze všech pokusů a zároveň použitých metod.

Na první pohled je zřejmé, že na prvních 5 datových sadách dosahovali všechny metody výsledků, které jsou blízké nule. Metody implementované pomocí watchmaker v některých případech zkonvergovaly do lokálních extrémů, ze kterých již nenašli lepší řešení, což prezentuje graf pomocí extrémních hodnot. Výsledky získané pomocí jenetics jsou ve většině případech velmi blízko nule, což značí o kvalitě tohoto frameworku. I metody naprogramované pomocí frameworku watchmaker dosahovaly ve většině případů velmi slušných výsledků.

Při sledování metod, které využívají Framework watchmaker, jsou průměrné výsledky velmi podobné. Při pohledu na nejlepší výsledky lze říct, že metoda MultiEngine ve většině případů dosáhla stejných, nebo nepatrně lepších výsledků, než jednoduché GP.



Obrázek 29 - Experiment č 1 - Celkový boxplot

7.2 Experiment č. 2

Experiment číslo 2 používá stejné datové sady, pro vyhodnocení popisovaných metod hledání optimálního řešení při využití GP. Experiment je spuštěn s obměněnými genetickými parametry. Hlavním rozdílem je, že v množině neterminálů chybí důležité funkce *sin, cos* a *sqrt*, kde zejména *sin* a *cos* jsou důležité pro najití řešení u datových sad číslo 2 a 6. Stejně jako v předešlém experimentu je pro každou metodu využito stejných 7 datových sad. Pro každou datovou sadu je dále vytvořeno 25 pokusů a výsledky jsou následně graficky znázorněny a popsány.

7.2.1 Počáteční nastavení

Následující tabulka popisuje počáteční nastavení všech pokusů. Genetické konstanty obsahují oproti předešlému experimentu jen drobné změny. Nejrazantnější změna je v množině neterminálů, která nyní neobsahuje funkce *sin, cos, sqrt*.

Tabulka 10 - Experiment č. 2 - Počáteční nastavení

Genetická konstanta	Hodnota
Počet generací	1000
Velikost populace	100
Max. hloubka stromu při inicializaci	4
Maximální velikost stromu při křížení	25
Pravděpodobnost křížení	1
Pravděpodobnost mutace	0,05
Pravděpodobnost turnajového schéma	1
Počet elitních jedinců	1
Počet generací preevoluce	10
Počet jedinců v preevoluci	10
Maximální velikost stromu preevoluce	5
Maximální hloubka stromu při inicializaci preevoluce	2
Terminály	Náhodné celé číslo z intervalu <0-10>, proměnná x, y, z dle potřeby datové sady
Neterminály	+, -, *, /, ^

7.2.2 Vyhodnocení jednoduché GP watchmaker

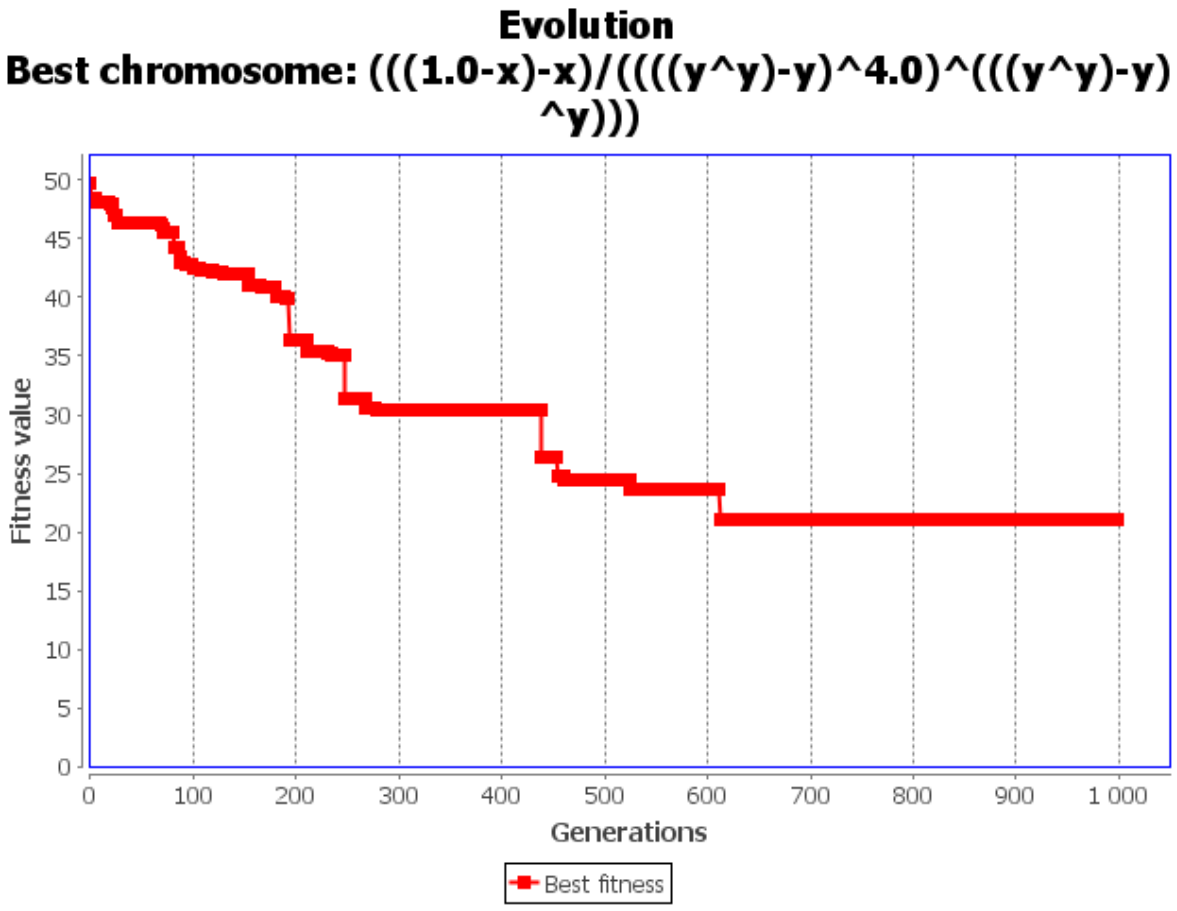
Kapitola obsahuje zhodnocení výsledků získaných spuštěním pokusů pomocí jednoduchého GP ve frameworku watchmaker. Následující tabulka obsahuje nejlepšího nalezeného jedince a jeho fitness ohodnocení pro každou datovou sadu, vybraného ze všech 25 pokusů pro danou sadu. Následně tabulka obsahuje průměrný čas trvání jednoho pokusu pro každou datovou sadu.

Tabulka 11 - Experiment č. 2 – Výsledky jednoduchá SR watchmaker

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	(x-x)^(x-x)	0	0,85
2	(3.0/((6.0/(5.0-(8.0-(((7.0+8.0)+3.0)+2.0))))-9.0))/(1.0+9.0)	0,52	10,8
3	(x+x)/((9.0/(x+(y*(y/(0.0/(x+(x+((5.0/x)+x)))))))+x)	0,044	32,32
4	((x+y)+y)+((x-((8.0-y)-(0.0*(y+y))))-(((y+y)-y)-x))	0	6,46
5	x+(((((z+(((x+(y/z))/z)/z))/z)+9.0)/y)+(x+(4.0/z)))	0,214	9.31
6	((y+y)-x)/(y^(((y^(0.0+((y^x)-y)))^((y^y)-y))-1.0))	16,14	5,65
7	y/((((((y-y)-y)/(y-((y-x)-(y+x))))-y)/(y-1.0))-y)	19,55	5,35

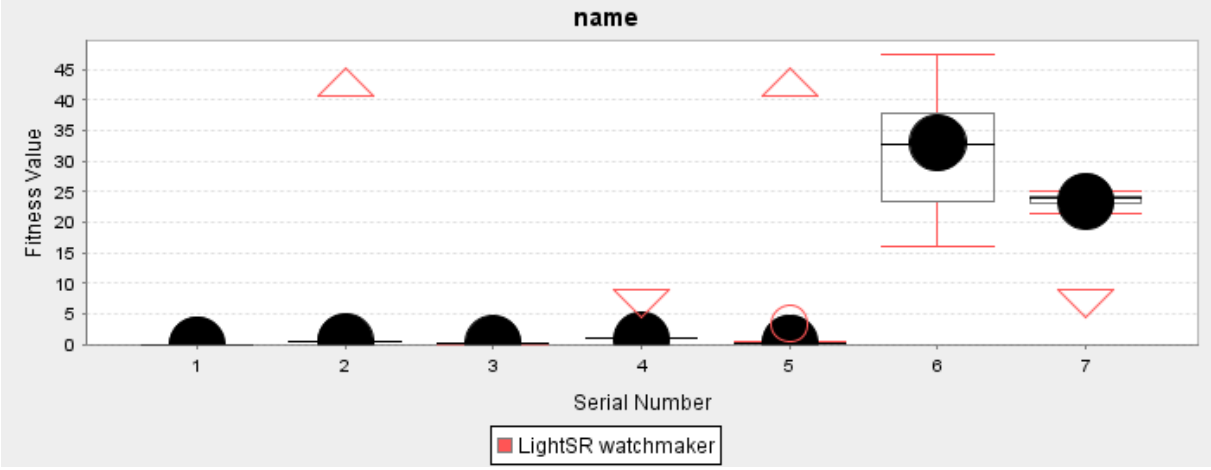
Při porovnání této tabulky s tabulkou 10, která obsahuje výsledky jednoduché SR v základním experimentu č. 1, lze pozorovat poměrně velké změny u datové sady 6. To je dáno tím, že správné řešení této sady obsahuje jak *sin*, tak i funkci *cos*. Absence těchto funkcí se velmi podepsala na výsledné fitness hodnotě nejlepšího jedince. I u dalších sad došlo ke změnám v hodnotách fitness. K těmto změnám mohlo přispět změna vstupních parametrů, nebo pouze náhoda algoritmu, který našel buď méně kvalitní řešení, nebo naopak kvalitnější řešení. Při pohledu na časové statistiky, tak u jednodušších funkcí z důvodů menší množiny neterminálů, došlo ke zlepšení. U funkcí složitějších nedošlo k výrazné změně, ovšem v případě sady, kde absentují chybějící neterminály, došlo k výraznému zhoršení.

Následující graf zobrazuje vývoj hodnoty fitness při pokusu na znevýhodněné sadě číslo 6. Z průběhu je patrné, že hledání probíhalo intenzivně přibližně 600 generací. Následně se hodnota ustálila přibližně na hodnotě 20.



Obrázek 30 - Experiment č. 2 - Vývoj fitness jednoduchá SR watchmaker

Pro ilustraci je možné sledovat nejlepší výsledky, kterých dosáhla daná metoda u všech datových sad. Jedná se o 25 výsledků pro každou datovou sadu, kde je z každého pokusu uložena fitness hodnota nejlepšího nalezeného jedince. Datová sada je v grafu určena pomocí pořadového čísla.



Obrázek 31 - Experiment č. 2 - Výsledky datových sad jednoduchá SR watchmaker

7.2.3 Vyhodnocení jednoduché GP jenetics

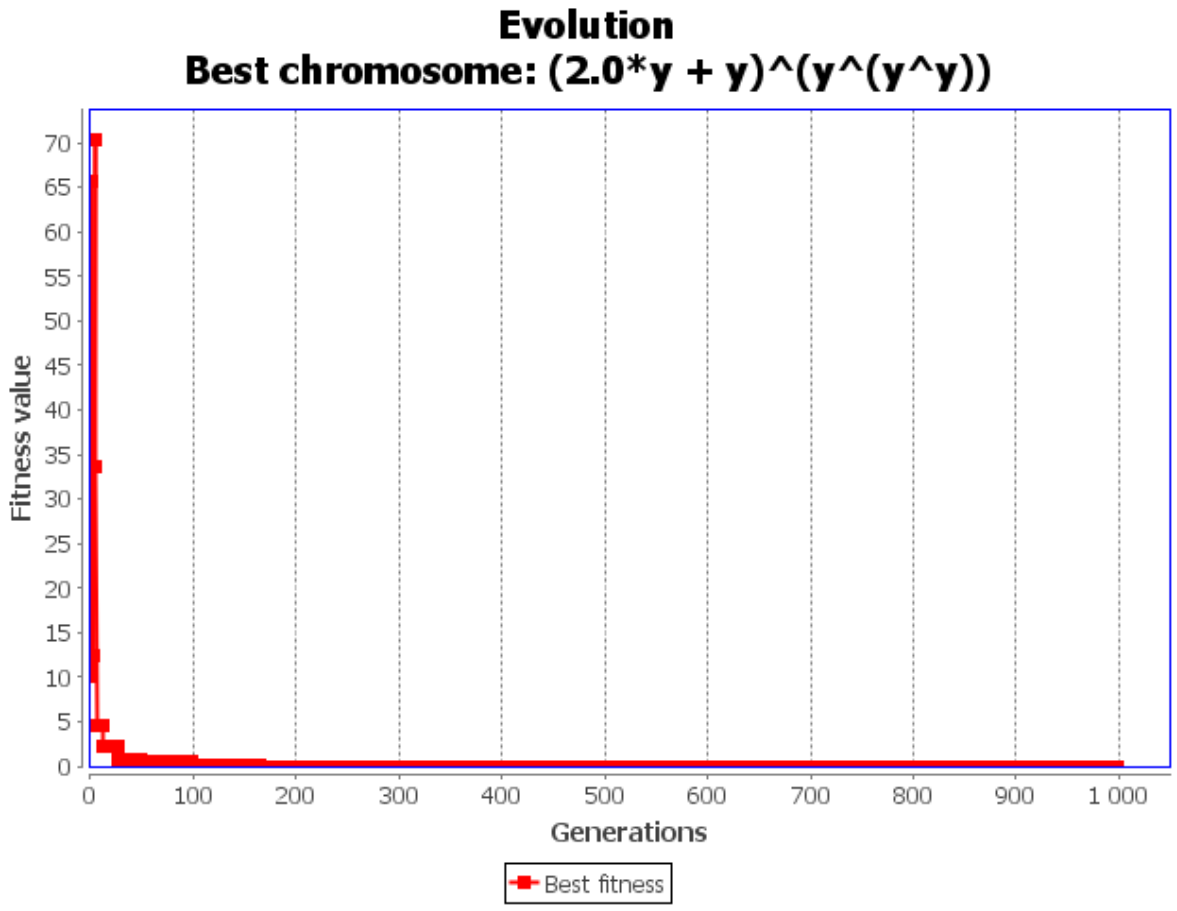
Následující tabulky a grafy zobrazují výsledky získané pomocí hledání optimálního řešení na datových sadách uvedených v tabulce 5 za použití knihoven jenetics a jednoduchého GP. Následující tabulka popisuje získané nejlepší jedince pro všechny datové sady a jejich ohodnocení. Poslední sloupec udává průměrnou dobu trvání jednoho pokusu pro každou datovou sadu.

Tabulka 12 - Experiment č. 2 - Výsledky jednoduchá SR jenetics

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	(2.0 - (2.0*x)^(9.0 + x))^(((0.0 - x) + x^5.0)/(7.0 - 2.0*x))	0	0,44
2	(x^2.0 + x) - ((0.2857142857142857 + x^2.0)*x^2.0 + 2.0*x)^x	$7{,}5 * 10^{-14}$	1,31
3	1.0 + (((y/x)^(x^x) - x^x) - y^x)^y	$9{,}3 * 10^{-11}$	1,35
4	(2.0*x + x) - (x*y + (1.0 + 2.0*y))	0	1,36
5	9.0 + (2.0 + 5.0/(z*x))/z	$3{,}14 * 10^{-4}$	1,32
6	(((2.0*x)/y + y^5.0) - (y^2.0 - (y + (x/7.0)/y)))^y	$8{,}2 * 10^{-5}$	1,26
7	(((y*x - x) - y) - 8.0) + y/(x + (x^2.0 - y))	$3{,}7 * 10^{-5}$	1,38

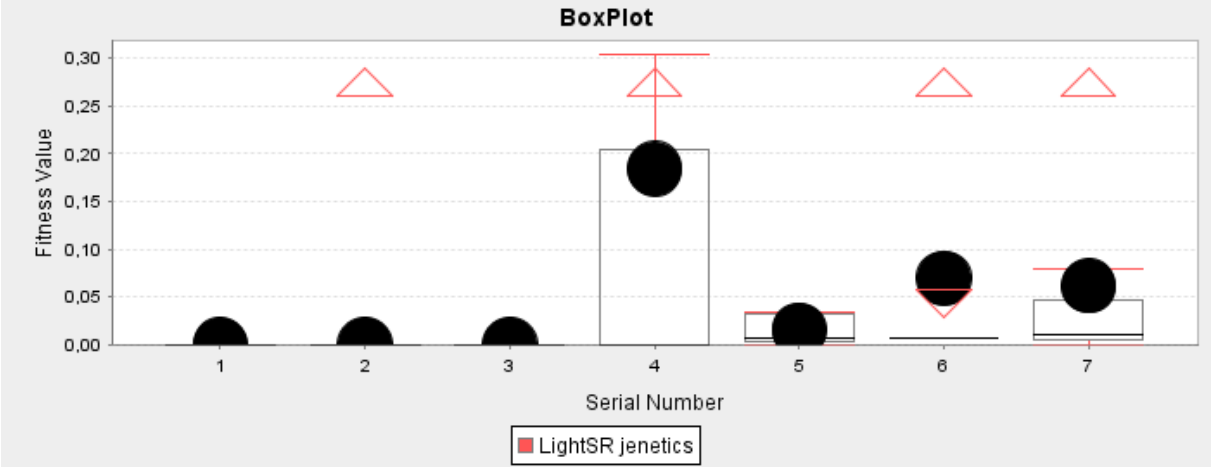
Výsledky získané pomocí jenetics jsou velmi blízké nule, tudíž je jisté, že algoritmus nalezl kvalitní řešení pro datové sady vygenerované pomocí funkcí *sin* a *cos*, i když tyto funkce nebyly zařazeny do množiny neterminálů. Kvalita nalezených jedinců odpovídá velikosti trénovací množiny. Pokud by byla tato množina obsáhlejší, je možné, že by tito jedinci nebyli natolik kvalitními řešeními. Ověřit by se tento fakt dal při použití validační množiny dat, která by byla odlišná od trénovací množiny.

Následující graf zobrazuje vývoj hodnoty fitness v určitém pokusu na datové sadě číslo 6. Z grafu je patrné, že optimální řešení bylo nalezeno již okolo sté generace a následně již lepší jedinec nebyl objeven.



Obrázek 32 - Experiment č. 2 -Průběh fitness jednoduchá SR jenetics

Pro srovnání všech výsledků je zde použit boxplot graf. Tento graf obsahuje fitness hodnoty nejlepších nalezených jedinců pro jednotlivé datové sady. Každý box tedy obsahuje 25 fitness hodnot. Všechny výsledky, až na několik extrémních hodnot, jsou velmi blízké nule, což ukazuje na kvalitu algoritmu.



Obrázek 33 - Experiment č. 2 - Výsledky datových sad jednoduchá SR jenetics

7.2.4 Vyhodnocení metoda MultiEngine watchmaker

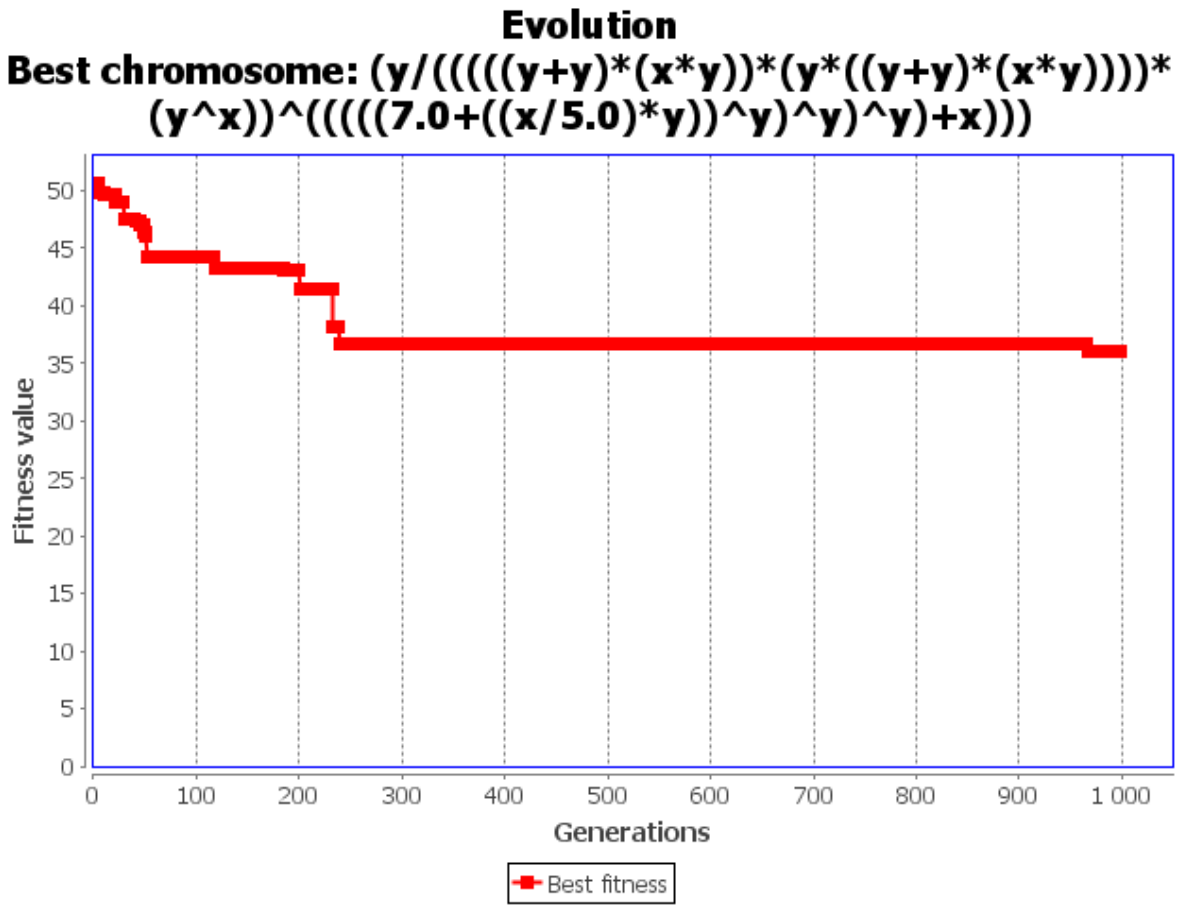
Následující tabulky a grafy zobrazují výsledky získané pomocí hledání optimálního řešení na datových sadách uvedených v tabulce 5 za použití frameworku watchmaker a metody MultiEngine. Následující tabulka popisuje získané nejlepší jedince pro všechny datové sady a jejich ohodnocení. Poslední sloupec udává průměrnou dobu trvání jednoho pokusu pro každou datovou sadu.

Tabulka 13 - Experiment č. 2 - Výsledky metody MultiEngine

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	((0.0-x)/3.0)/(8.0+(x^8.0))	0	2,7
2	(2.0*6.0)/((((6.0*9.0)*(7.0*1.0))/((9.0-x)*x))-(((5.0*7.0)^(0.0/6.0))*(8.0*6.0)))	0,52	24,58
3	y+((((x^3.0)+x)/((x^4.0)+x)) / ((0.0/9.0)/(1.0*(0.0^5.0))))	0,11	46,3
4	((x-x)+(7.0*y))+(x+(9.0+((((6.0*y)+(x-8.0))+x)-8.0)))	0	8,29
5	(x+((7.0+((x/y)/z))/z))+((8.0/y)+x)	0.288	9,65
6	(((x/(x/y))^(5.0^(0.0^(x^0.0))))/(5.0^(4.0^(x^9.0))))/((y-(((x/(x/y))-x)-(1.0^(x/(x/y)))))-(8.0^(x/(x/y))))	27,68	10,1
7	x-(((y-5.0)-(((y-1.0)/((y-4.0)*(y-4.0)))/(y-5.0)))/(2.0/((x+(y+x))-6.0)))	23,46	12,88

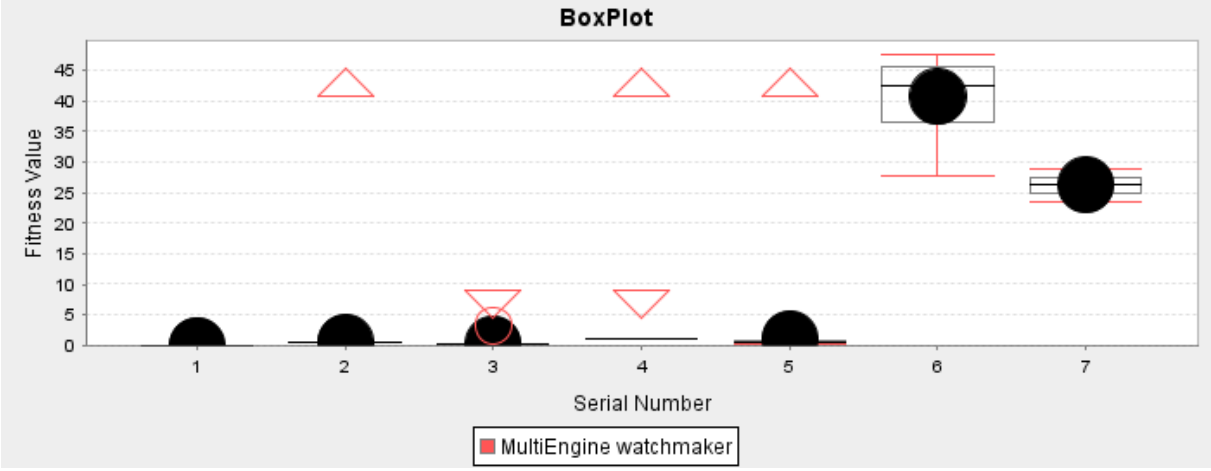
Z tabulky vyplívá, že metoda MultiEngine si i při vyřazení některých důležitých neterminálů dokázala poměrně dobře poradit s řešením hledání optimálního řešení. Výsledky bohužel nejsou tak kvalitní, jako v případě použití frameworku jenetics, ale na druhou stranu metoda MultiEngine dosahuje podobných výsledků, než řešení pomocí klasického GP s využitím watchmaker. Výslední jedinci se mohou na první pohled zdát poněkud složití, to je dáno hlavně tím, že watchmaker neumí zjednodušit nalezené vzorce, tak jako jenetics. Po aplikování matematických úprav by výsledný předpis funkce byl jednodušší.

Následující graf zobrazuje vývoj hodnoty fitness v průběhu jednoho pokusu hledání optimálního řešení při použití trénovací datové sady číslo 6. Jeden z nejlepších jedinců byl nalezen již kolem generace 250. Ke konci algoritmu, přibližně v generaci 975 byl ovšem tento jedinec překonán. Nejlepší jedinec v tomto pokusu dosáhl hodnoty fitness přibližně 35.



Obrázek 34 - Experiment č. 2 - Vývoj fitness MultiEngine

Následující boxplot graf shrnuje všechny výsledky získané použitím metody MultiEngine. Každý box je vytvořen z hodnot, které představují fitness hodnoty jedinců, kteří byli prohlášeni za nejlepší výsledky jednotlivých pokusů. Z grafu je patrné, že výsledky metody MultiEngine na prvních datových Sadách dosahovaly většinou hodnot blízkých nule. U posledních dvou datových sad jsou průměrné hodnoty vyšší, což je dáno nejspíše větší složitostí hledaných funkcí a také velikostí trénovacích množin.

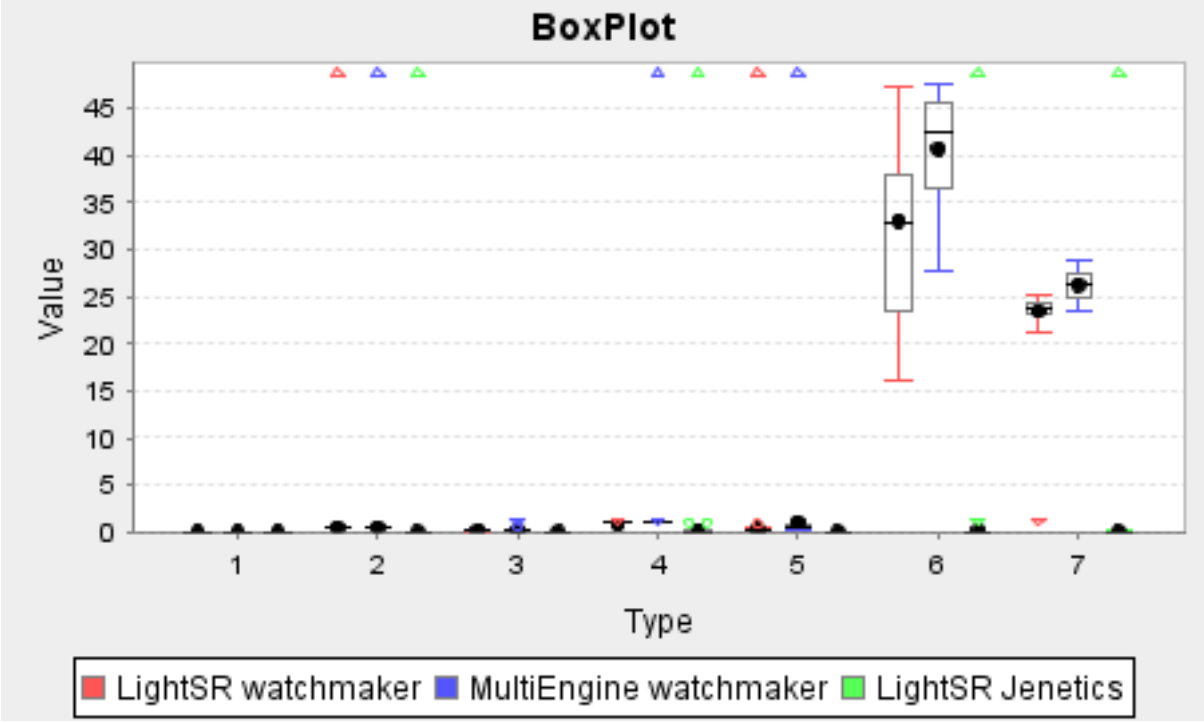


Obrázek 35 - Experiment č. 2 - Výsledky datových sad metoda MultiEngine

7.2.5 Vyhodnocení experimentu číslo 2

Vzhledem k absenci potřebných funkcí *cos* a *sin* došlo ke zhoršení u datové sady číslo 6. Tato sada je vygenerována podle předpisu obsahujícího obě zmíněné funkce. Datová sada číslo 2, také obsahuje jednu ze zmiňovaných funkcí, ale není natolik složitá, tudíž si s absencí této funkce metody poradí a nahradí ji jiným způsobem.

Při porovnání zkoumaných metod je vidět, že v tomto případě došlo při řešení datové sady číslo 6 k výraznému zhoršení oproti jednoduchému GP. To může být způsobeno i tím, že se tato metoda hůře chová při absenci některých důležitých neterminálů. Další možností je, že počet pokusů je nedostačující a při zvýšení například na 50 by bylo dosaženo lepších výsledků. Další možnosti pro zlepšení spočívají v odladění genetických parametrů.



Obrázek 36 - Experiment č 2 - Celkový boxplot

7.3 Experiment č. 3

Tento experiment vychází z prvního základního experimentu. Kapitola popisuje výsledky získané pomocí tří uvedených metod na daných datových sadách. Oproti předešlým experimentům jsou přidány nové složitější neterminály, jako je *min*, *max* a *avg*. Bohužel Framework jenetics neposkytuje možnost využít neterminál *avg* a rozšíření o tuto funkčnost bohužel není možné. Pro zachování přibližně stejné složitosti je proto místo *avg* použit

v nastavení jednoduchého GP v jenetics použit neterminál *exp*. Další nepatrné změny jsou v genetických parametrech, kde byly upraveny některé konstanty.

7.3.1 Počáteční nastavení

Tabulka níže uvádí počáteční nastavení všech pokusů, které jsou v rámci experimentu spuštěny. Největší rozdíly jsou v množině neterminálů, kde jsou přidány nové funkce (*min*, *max*, *avg*). Bohužel funkce *avg* není dostupná ve výčtu funkcí frameworku jenetics a nebyl nalezen způsob, jak ldosáhnout rozšíření o zmíněnou funkci. Namísto funkce průměr je zde pro zachování obtížnosti použita funkce *exp*, která značí Eulerovo číslo s mocninou (e^x).

Dále jsou pro účely tohoto experimentu pozměněny některé genetické parametry pro evoluci i preevoluci, kterou využívá pouze metoda MultiEngine. Takovými parametry jsou například maximální hloubka stromu při inicializaci, pravděpodobnosti pro křížení, mutaci a selekci, nebo maximální velikosti stromů pro preevoluci.

Tabulka 14 - Experiment č. 3 - Počáteční nastavení

Genetická konstanta	Hodnota
Počet generací	1000
Velikost populace	120
Max. hloubka stromu při inicializaci	2
Maximální velikost stromu při křížení	30
Pravděpodobnost křížení	0,95
Pravděpodobnost mutace	0,09
Pravděpodobnost turnajového schéma	0,95
Počet elitních jedinců	1
Počet generací preevoluce	15
Počet jedinců v preevoluci	15
Maximální velikost stromu preevoluce	4
Maximální hloubka stromu při inicializaci preevoluce	2
Terminály	Náhodné celé číslo z intervalu <0-10>, proměnná x, y, z dle potřeby datové sady
Neterminály	+, -, *, /, ^, min, max, avg/exp

7.3.2 Vyhodnocení jednoduché GP watchmaker

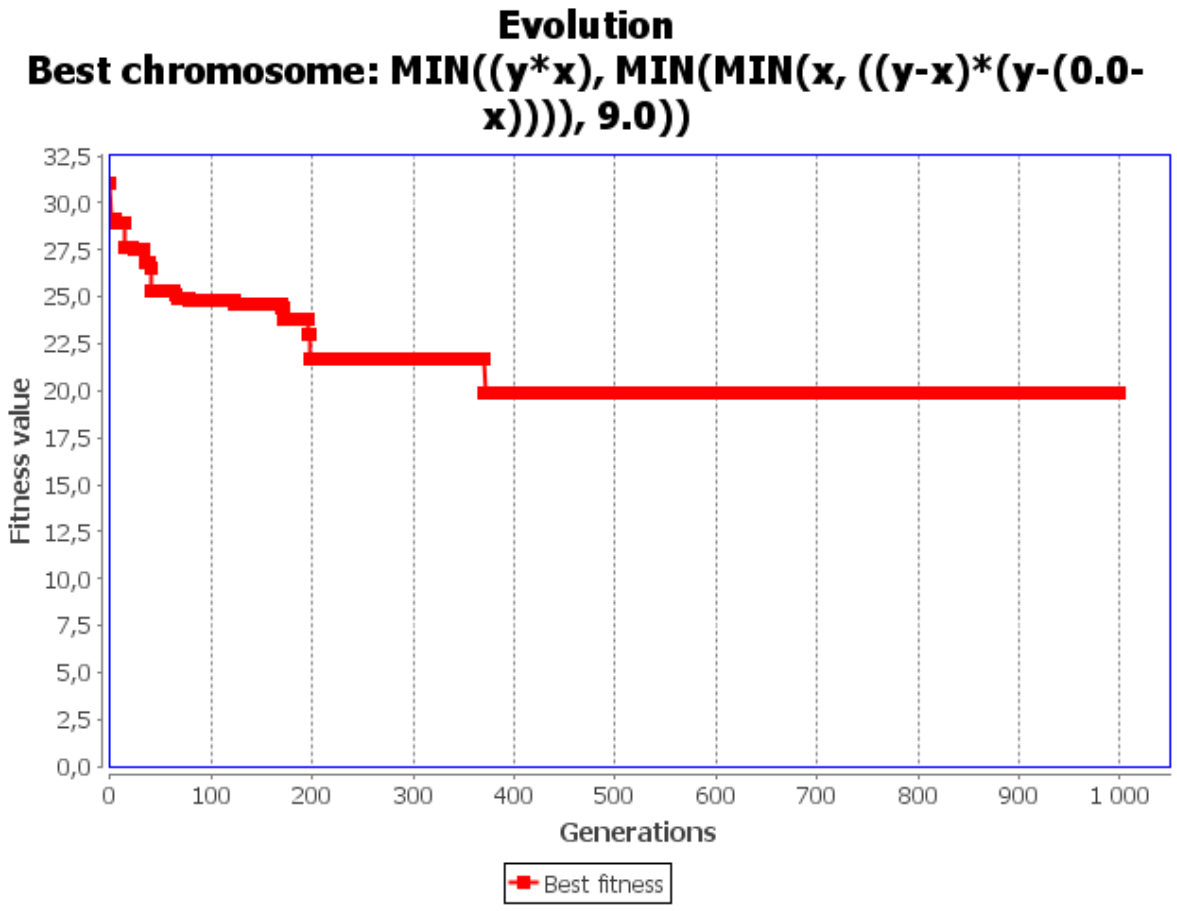
Následující tabulka obsahuje výsledky získané spuštěním jednoduchého GP ve frameworku watchmaker pro všechny datové sady. Pro každou datovou sadu je spuštěno 25 pokusů se stejným počátečním nastavením uvedeným v tabulce 14. Pro každou datovou sadu je uveden nejlepší nalezený jedinec a jeho fitness ohodnocení. Poslední sloupec obsahuje průměrnou dobu trvání jednoho pokusu.

Tabulka 15 - Experiment č. 3 - Výsledky jednoduché SR watchmaker

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	x/x	0	1,16
2	MIN((x/((x/AVG((8.0/3.0), 1.0))-(((2.0+MAX(8.0, (x/AVG(x, (3.0+MAX(x, x))))))+8.0)*x))), x)	0,52	21,29
3	((y*MIN(MIN((y*y), 7.0), (x*x)))+(y+(y+y)))/((y+y)+y)	0,036	44,36
4	x+((((x+(4.0*y))+(x+y))+y)+((((0.0*y)+y)+((y+y)-4.0))+y))	0	10,1
5	(AVG(AVG((x+x), z), ((AVG(x, (x+x))+4.0)/y))+x)+(AVG((x+4.0), 2.0)/z)	0.14	16,17
6	((4.0-x)-x)/(y^((y^((MIN(y, MIN(y, y))^MIN(y, 5.0))^y))-8.0))	20,96	9,95
7	MIN(MIN((y-y), MIN(y, y)), ((((MIN(MIN(y, (x-MIN((0.0-y), y))), y)*(y-x))+y)*y)+y))	19,19	8,83

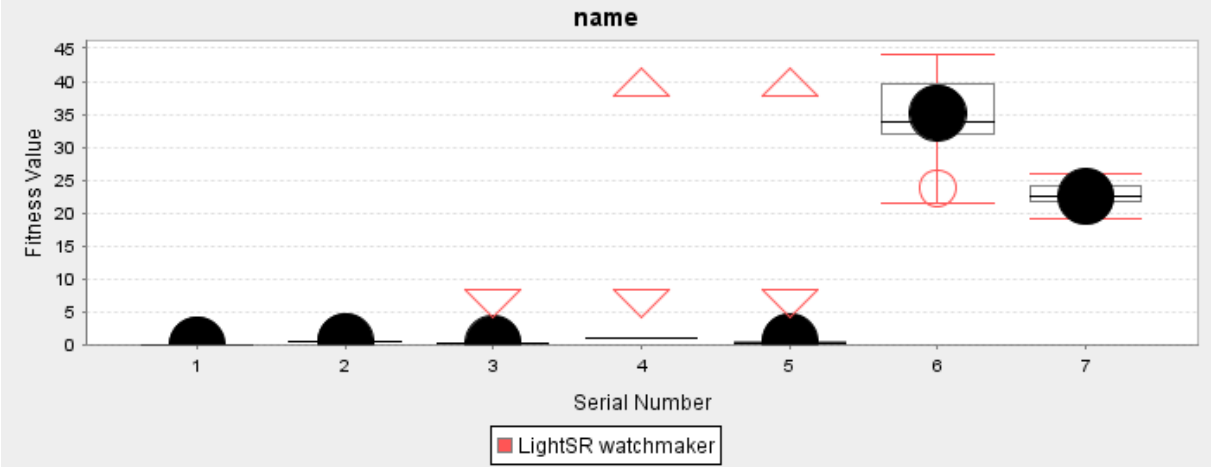
V tabulce je vidět, že nově použité neterminály jsou využity i ve výsledných jedincích. I navzdory absenci některých důležitých neterminálů byli nalezeni jedinci, kteří dosáhli kvalitních výsledků na jednotlivých datových sadách.

Následující graf popisuje vývoj fitness hodnoty během jednoho pokusu na datové sadě číslo 7. Vývoj fitness hodnoty pozvolně klesá a hodnota se ustaluje přibližně kolem 350 generace. V následujících generacích nedošlo již k žádnému pokroku.



Obrázek 37 - Experiment č. 3 - Vývoj fitness jednoduchá SR watchmaker

Konečné výsledky pro všechny pokusy a datové sady jsou shrnuty v následujícím boxplot grafu. Každý box obsahuje fitness hodnoty nejlepších nalezených jedinců v jednotlivých pokusech na dané datové sadě. Hodnoty dokazují, že jednotlivé pokusy na datových sadách dosahovaly podobných výsledků a právě proto jsou boxy poměrně malé.



Obrázek 38 - Experiment č. 3 - Výsledky datových sad jednoduchá SR watchmaker

7.3.3 Vyhodnocení jednoduché GP jenetics

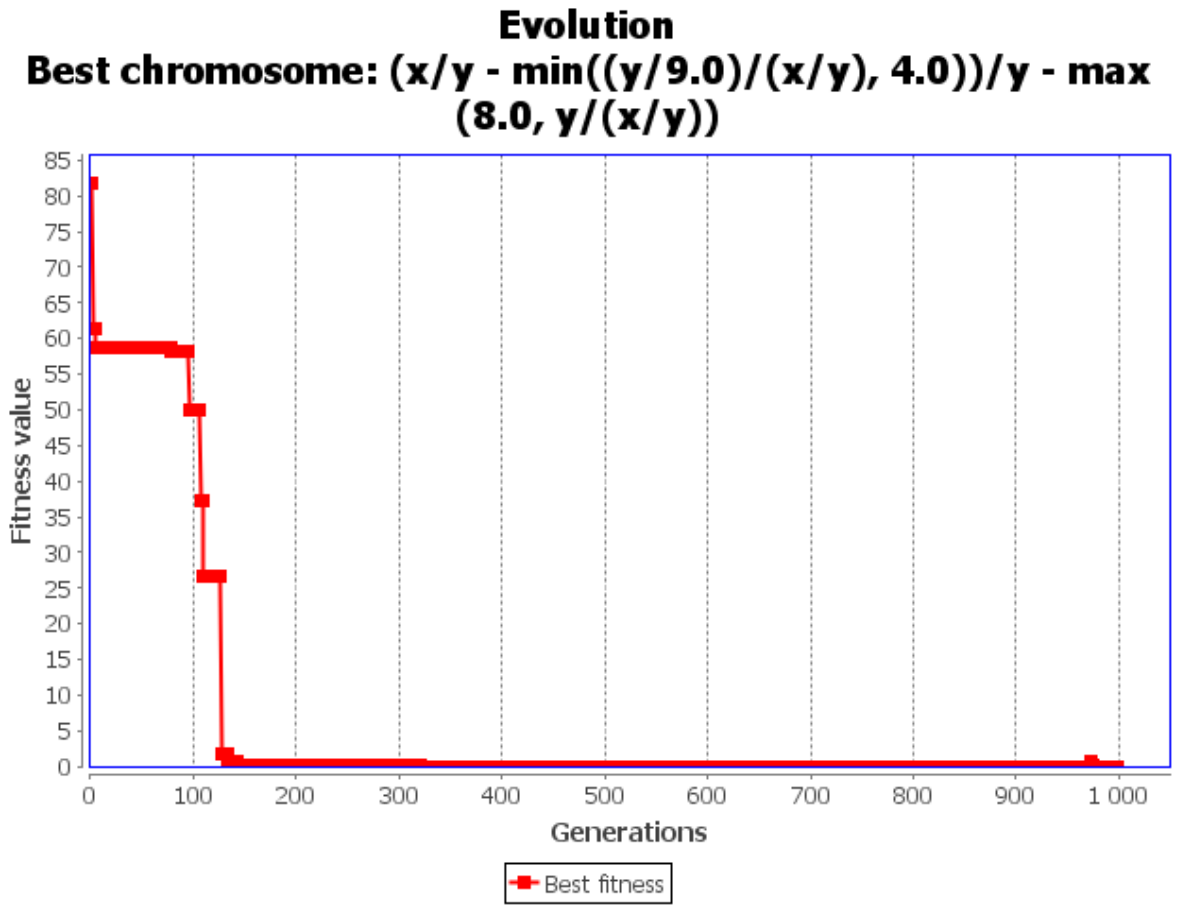
Tato kapitola obsahuje výsledky a shrnutí použití frameworku jenetics s počátečním nastavením určeným pro tento experiment. Jediným rozdílem v počátečním nastavení je, že Framework jenetics neumožňuje využít neterminál *avg*. Z důvodu zachování podobné složitosti je tedy nahrazen neterminálem *exp*. Následující tabulka uvádí získané výsledky pro každou použitou datovou sadu, která je označena pořadovým číslem. Následně je ke každé datové sadě uveden nejlepší nalezený jedinec, jeho fitness ohodnocení a průměrná doba trvání jednoho pokusu.

Tabulka 16 - Experiment č. 3 - Výsledky jednoduché SR jenetics

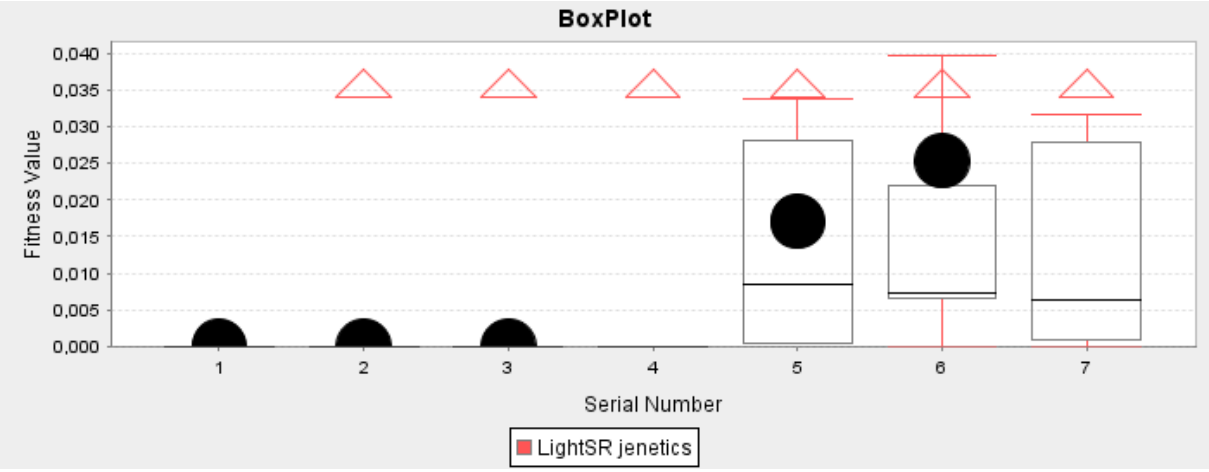
Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	(x^3.0)^4.0	0	0,76
2	x*x^(x^(x^x)) - (x^((x^x)^(((x^(x^x))^x)^x)))^x	$1{,}71 * 10^{-14}$	1,71
3	max(exp(exp(x)/(x + exp(exp((x + y/x) + exp(exp(exp(x))))))) + exp(exp(y)/(x + exp(x))), x)	$1{,}01 * 10^{-10}$	1,48
4	min(x - (y - min(min(min(x, x), x) - (max(1.0 + y, x) - min(y, x)), x)), y) - x*y	0	1,78
5	max(y + max(max((z + (x + 4.0)) / z, x), 7.0/(x - 9.0)), z)/z + 9.0	$3{,}14 * 10^{-4}$	1,62
6	((y*exp(x))^y)^y/x)*exp(y/(y*x)^y)^(y /exp((2.0*y)^y)	$2{,}08 * 10^{-5}$	1,56
7	((y^2.0*((y + ((2.0*x)*x) / y)*x + x))^y)^((y^y)^y) - y)/x	$9{,}72 * 10^{-6}$	1,79

Framework jenetics dosáhl i v tomto případě velmi solidních výsledků, které jsou velmi blízké hodnotě nula. O kvalitě nalezených řešení by rozhodlo ještě testování na validační množině dat.

Následující graf popisuje vývoj hodnoty fitness při hledání optimální řešení pro datovou sadu číslo 7 za využití frameworku jenetics. K razantním změnám docházelo přibližně do generace číslo 130 a následně se již hodnota ustalovala. Konečný výsledek byl velmi blízký hodnotě nula.



Obrázek 39 - Experiment č. 3 -Průběh fitness jednoduchá SR jenetics



Obrázek 40 - Experiment č. 3 - Výsledky datových sad jednoduchá SR jenetics

7.3.4 Vyhodnocení metody MultiEngine

Tato kapitola obsahuje nashromážděné výsledky provedení hledání optimálního řešení pro jednotlivé datové sady za použití metody MultiEngine, která je připravena s využitím frameworku watchmaker. Následující tabulka popisuje nejlepší nalezené jedince a jejich

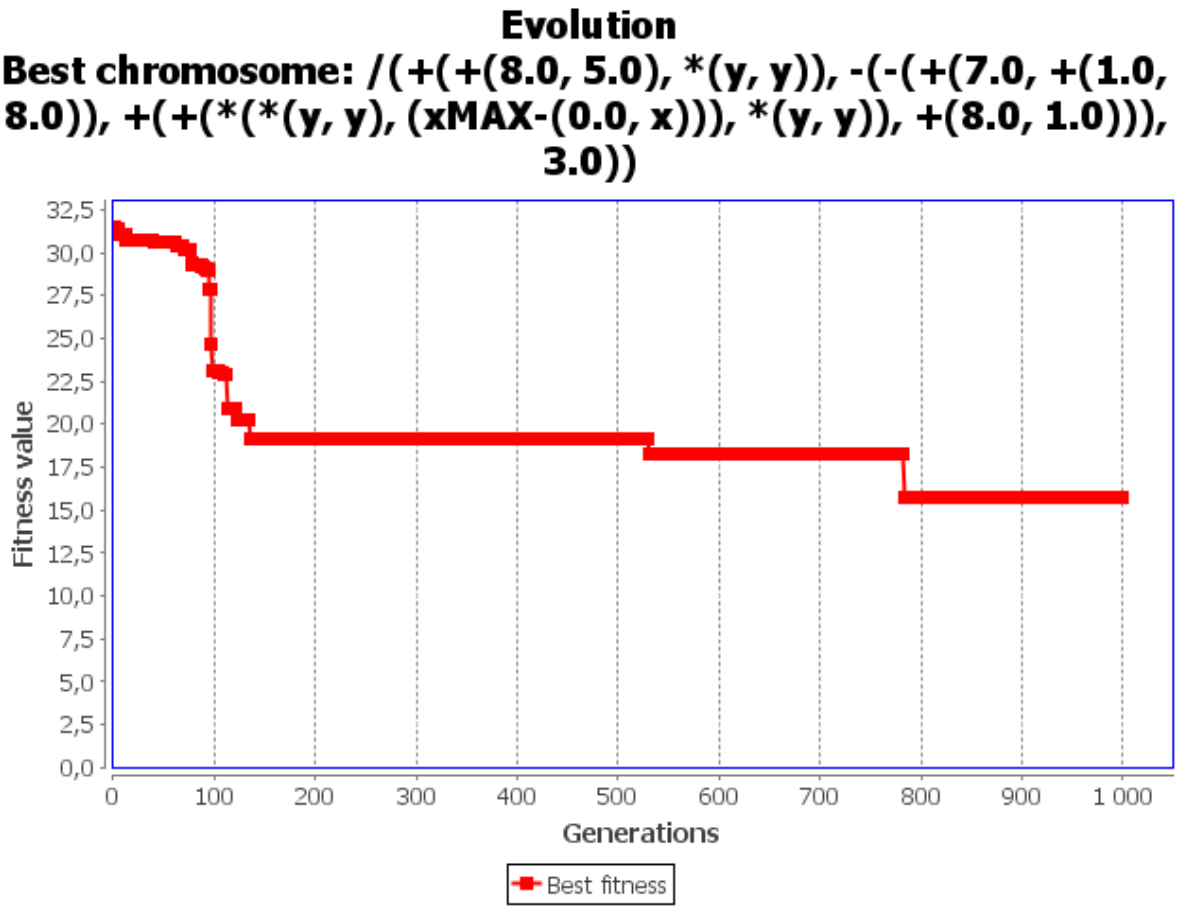
fitness ohodnocední pro jednotlivé datové sady. V posledním sloupci je uveden průměrný čas trvání jednoho pokusu pro určitou datovou sadu.

Tabulka 17 - Experiment č. 3 - Výsledky metody MultiEngine

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	(MAX(MAX(MAX(MAX(x, x), 0.0), x), x)/MIN((MIN(6.0, 4.0)^(x+x)), (6.0^2.0)))	0	2,98
2	AVG(7.0, 0.0)/(((AVG(1.0, 1.0)*AVG(0.0, 2.0))*AVG(3.0, 3.0))-(AVG(9.0, 7.0)*((AVG(0.0, 4.0)*6.0)*(AVG(AVG(3.0, 9.0), MAX(1.0, x))*AVG(4.0, 5.0)))))	0,52	28,9
3	MAX(y, 1.0)+((y/(y*MIN(3.0, x)))/((y/y)+((y/y)/(((y/y)*MAX(MAX(7.0, 0.0), y))*(y*MIN(3.0, x))))))	0,041	60
4	(8.0+AVG((y+y), (y-MAX(7.0, 9.0))))+(((6.0+x)+((AVG((y+y), (y-MAX(4.0, 8.0)))*4.0)+(x+x)))+(y*3.0))	0	8,71
5	z+(((((z+(6.0/z))+(2.0/y))/z)+(7.0/y))+MAX((y-y), x))	0.29	10,87
6	(y^y)/((y^y)-(((MIN(y, (x+x))-((x-((x^y)+MIN(y, (x+x))))+(y^y)))*(y^y))+(y^y)))	22,99	9,42
7	(MAX(MIN(AVG(MIN(MIN(y, 9.0), x), x), (y-(x-1.0))), ((x+AVG(y, AVG(AVG(AVG(4.0, 0.0), 2.0), 1.0)))/MAX((x/MAX(MIN(AVG(MIN(MIN(y, 5.0), x), x), (y-(x-5.0))), MAX(MIN(AVG(MIN(MIN(y, 8.0), x), x), (y-(x-0.0))), y))), MIN(AVG(MIN(MIN(y, 0.0), x), x), (y-(x-6.0)))))))-MAX(x, 7.0))	22,6	11,66

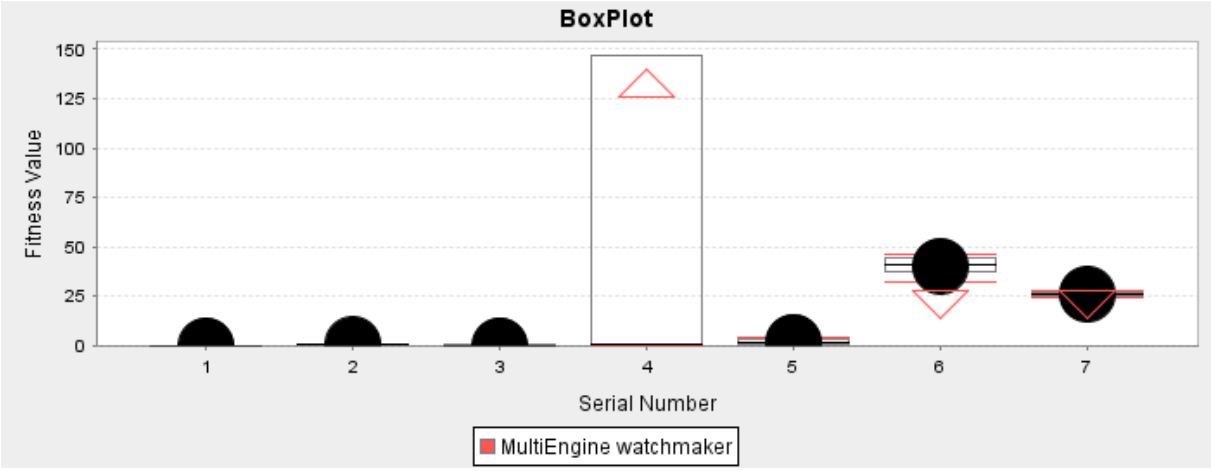
Při pohledu na výsledné jedince se nabízí, že se v žádném případě nemůže jednat o hledané funkce. Jedinci jsou uvedeni ve tvaru, který vygenerovala metoda MultiEngine, tedy nejsou zjednodušeni, tak jako tomu je u jenetics. Po aplikaci některých matematických úprav, by se dalo vzorec velmi snadno zjednodušit do čitelnější podoby. Výsledné fitness hodnoty a časové údaje odpovídají většině testovaným metodám naprogramovaných pomocí frameworku watchmaker.

Následující graf zobrazuje fitness ohodnocení nejlepšího jedince v každé generaci jednoho pokusu při řešení nad datovou sadou číslo 7. Z grafu je možné vyčíst, že k největšímu pokroku docházelo do generace číslo 150 a nadále se hodnota držela na přibližně stejném čísle. Kolem generace číslo 780 došlo k dalšímu výraznému pokroku a nadále se již hodnota nezlepšila.



Obrázek 41 - Experiment č. 3 - Vývoj fitness metoda MultiEngine

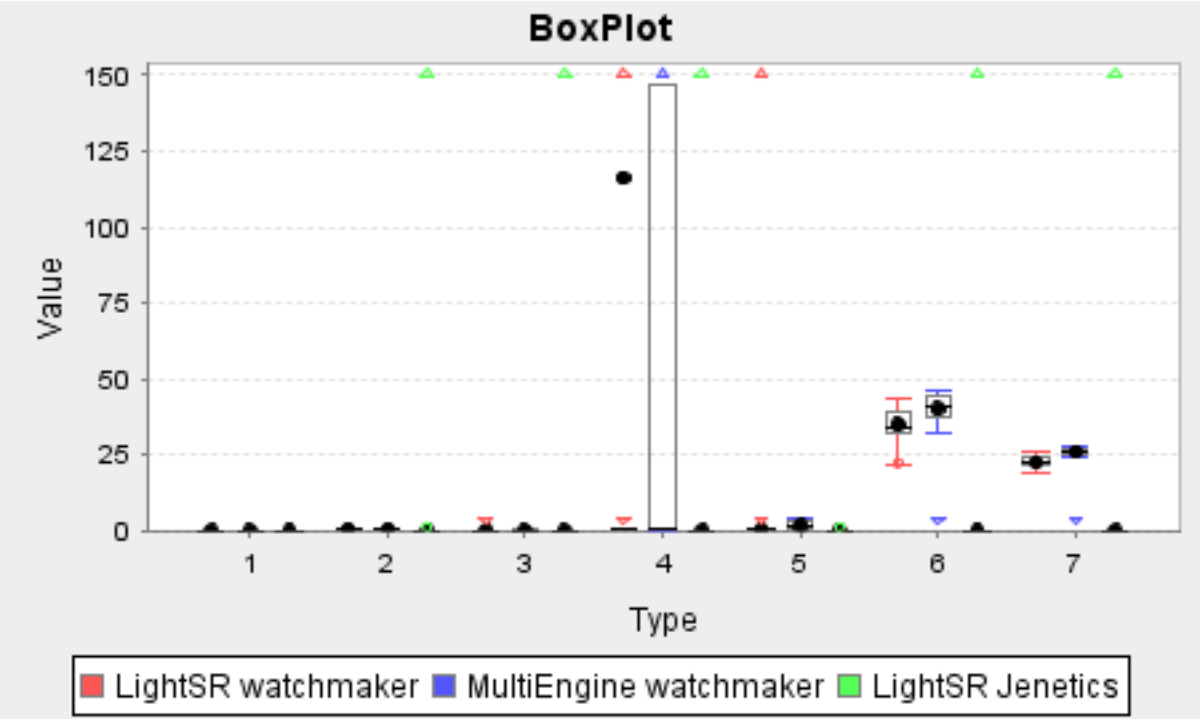
Následující graf obsahuje všechny výsledky jednotlivých pokusů pro metodu MultiEngine. Každý box obsahuje 25 hodnot, které představují fitness ohodnocení nejlepších nalezených jedinců ve všech pokusech pro jednotlivé datové sady. Z výsledků je patrné, že i při absenci důležitých neterminálů *cos* a *sin*, dokáže metoda MultiEngine nalézt kvalitní řešení a do výsledků zahrnout i nově přidané neterminály *avg*, *min* a *max*.



Obrázek 42 - Experiment č. 3 - Výsledky datových sad metoda MultiEngine

7.3.5 Vyhodnocení experimentu číslo 3

Souhrnný boxplot porovnává všechny výsledky experimentu 3 získané z průběhů zkoumaných metod. Z grafu vyplívá, že nejlepších výsledků na datových sadách dosáhla metoda založená na frameworku jenetics. Metody naprogramované pomocí frameworku watchmaker dosáhly v některých případech horších průměrných výsledků, ale obě dosáhly podobných kvalit. Průměrné hodnoty sice ukazují, že řešení pomocí metody MultiEngine dosahuje horších výsledků, na druhou stranu odlehlé hodnoty grafu ukazují, že metoda MultiEngine dosáhla nepatrně lepších výsledků.



Obrázek 43 - Experiment č. 3 - Celkový boxplot

7.4 Experiment č. 4

Poslední zmíněný experiment používá všechny dosud zmíněné neterminály. Ke všem potřebným, které jsou použity v experimentu číslo 1 přidány i nové, uvedené v experimentu číslo 3.

7.4.1 Počáteční nastavení

Počáteční nastavení tohoto experimentu obsahuje drobné změny v genetických konstantách. Nejmarkantnější změna je ovšem v množině použitých neterminálů, která obsahuje všechny dosud zmiňované funkce. V případě jednoduché SR v jenetics je zaměněna funkce *avg*, která není v jenetics k dispozici, za funkci *exp*.

Genetická konstanta	Hodnota
Počet generací	1000
Velikost populace	100
Max. hloubka stromu při inicializaci	2
Maximální velikost stromu při křížení	30
Pravděpodobnost křížení	1
Pravděpodobnost mutace	0,14
Pravděpodobnost turnajového schéma	1
Počet elitních jedinců	1
Počet generací preevoluce	8
Počet jedinců v preevoluci	10
Maximální velikost stromu preevoluce	5
Maximální hloubka stromu při inicializaci preevoluce	2
Terminály	Náhodné celé číslo z intervalu <0-10>, proměnná x, y, z dle potřeby datové sady
Neterminály	+, -, *, /, ^, sin, cos, sqrt, min, max, avg/exp

7.4.2 Vyhodnocení jednoduché GP watchmaker

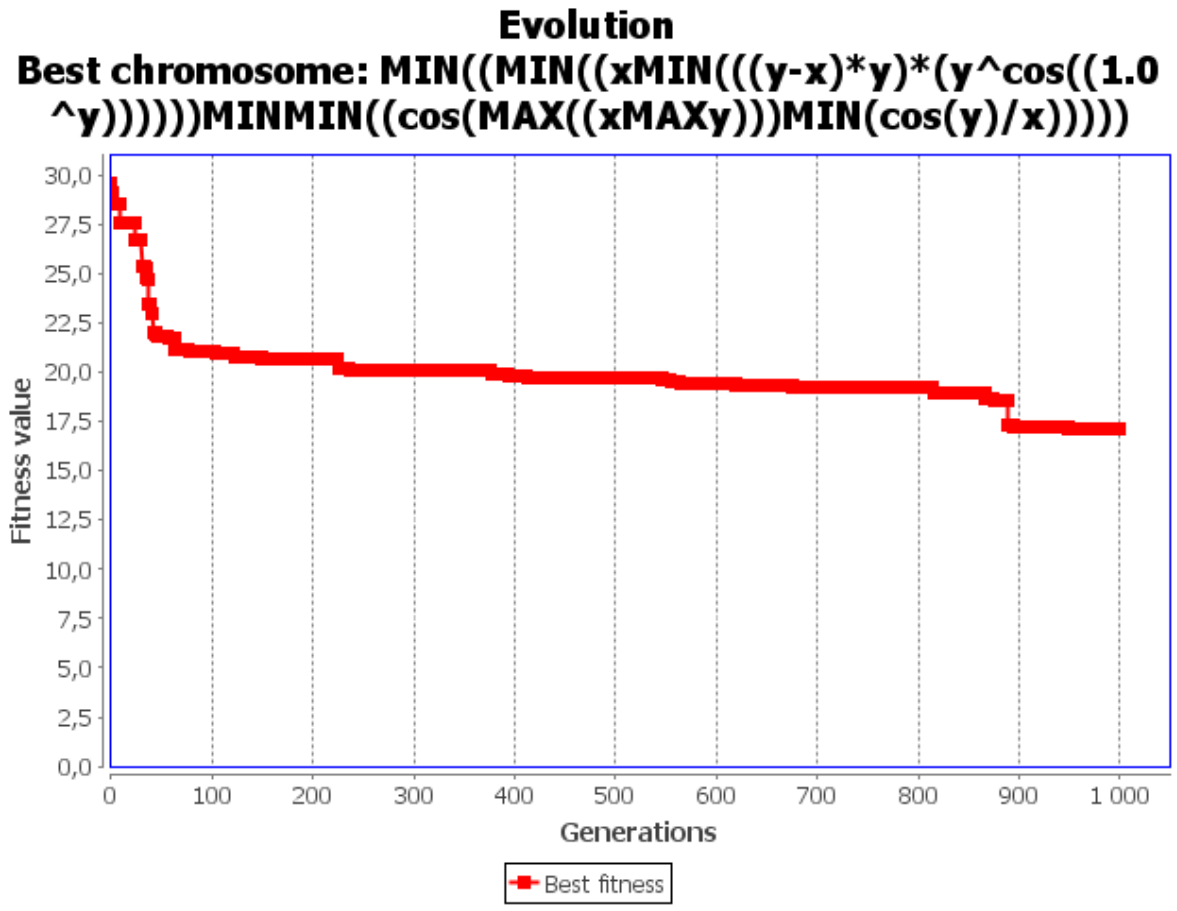
Tato kapitola obsahuje získané výsledky z pokusů, kde byly použity datové sady jako vstupy pro jednoduché GP naprogramované pomocí frameworku watchmaker. Následující tabulka obsahuje nejlepší nalezené jedince a jejich fitness ohodnocení pro každou datovou sadu. Tabulka také obsahuje průměrný čas trvání jednoho pokusu algoritmu.

Tabulka 18 - Experiment č. 4 - Výsledky jednoduché SR watchmaker

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	cos((x-x))	0	2,01
2	cos(4.0)/(cos(cos(5.0))-8.0)	0,52	14,57
3	AVG((cos(cos(2.0))+cos((cos(cos(cos(x)))+cos(x)))), (cos((cos(2.0)+cos(y)))+cos((cos(cos(cos(y)))+cos(y))))	0,025	32,47
4	((((5.0-5.0)+y)+(x+x))+((0.0*y)+y))+(y+(x+(8.0*y)))	0	8,68
5	(AVG(2.0, x)+(((y+AVG((x+MAX(y, 7.0)), 4.0))/y)+(AVG(((4.0+2.0)/z), MAX(sin(y), x))/z)))+x	0.134	12,54
6	sin(x)/MIN(MIN(sin(cos(sin(sin(sin(cos(MIN(x, cos(sin(sin(x)))))))))), y), MIN(y, AVG(MIN(y, MIN(y, cos(y))), (0.0^y))))	0	6,54
7	MIN((((cos(x)*MIN(MIN(MIN(cos((2.0/y)), y), cos((y-sqrt(y)))), (x-x)))*AVG(sqrt(y), x))*y), cos(sqrt(y)))	10,06	6,92

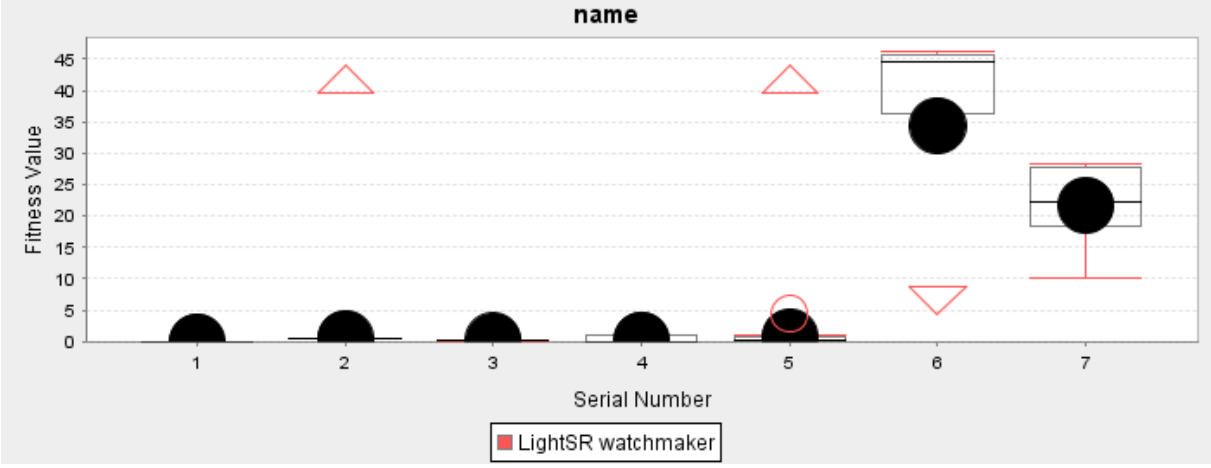
Nalezení jedinci jsou uvedeni v nezjednodušeném stavu tak, jak byli vygenerováni díky GP ve watchmaker. Jak již bylo řečeno, součástí watchmaker ani jeho úpravy není algoritmus pro zjednodušení matematického vzorce. Získané výsledky jsou díky nastavení experimentu poměrně kvalitní a hodnocení jednotlivých jedinců ukazuje na kvalitu řešení.

Následující graf představuje vývoj hodnoty fitness v rámci jednoho pokusu na datové sadě číslo 7. Pro každou generaci je do grafu vloženo fitness ohodnocení nejlepšího jedince. Pro tento pokus se ohodnocení pozvolna zlepšovalo až do generace přibližně 950. Následně se ohodnocení nejlepšího jedince ustálilo přibližně na hodnotě 17,5.



Obrázek 44 - Experiment č. 4 - Vývoj fitness jednoduchá SR watchmaker

Následující boxplot graf zobrazuje v každém boxu 25 ohodnocení nejlepších jedinců, nalezených v pokusech pro každou datovou sadu. Jako ve většině předchozích případů, první datové sady dosahují výsledků velmi blízkých nule. Průměrné hodnoty na datových sadách 6 a 7 dosahují, i po přidání nových neterminálů, velmi dobrých výsledků. Díky vzdáleným hodnotám se i v těchto případech nejlepší výsledky přiblížily fitness ohodnocení nula.



Obrázek 45 - Experiment č. 4 - Výsledky jednoduchá SR watchmaker

7.4.3 Vyhodnocení jednoduché GP jenetics

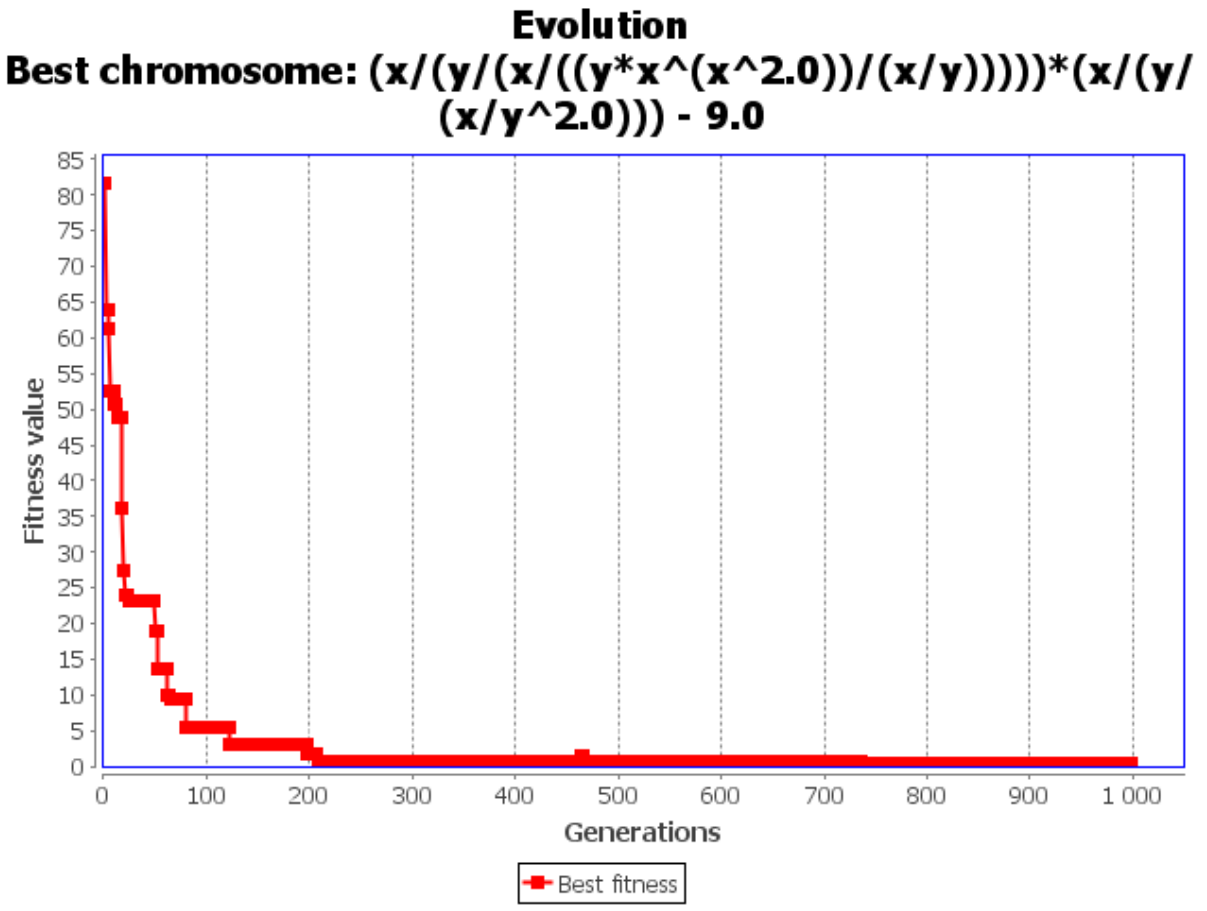
V této kapitole jsou uvedeny výsledky získané při testování jednoduchého GP naprogramovaného ve frameworku jenetics s počátečním nastavením určeným pro tento experiment. Následující tabulka obsahuje nejlepší nalezené jedince spolu s jejich fitness ohodnocením pro každou datovou sadu. Tabulka také obsahuje průměrnou dobu trvání jednoho pokusu u dané datové sady.

Tabulka 19 - Experiment č. 4 - Výsledky jednoduché SR jenetics

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	x^2.0	0	0,52
2	-0,83	0	0,94
3	cos(cos((y/(cos(cos(cos(cos(cos(y))))))/ (cos(cos(cos(exp(cos(x))))) + x) + y))^y)/y) + 1.0	$1{,}79 * 10^{-11}$	0,86
4	(((x - y) - y) + (x + (max(y/(0.0 - y), y) + (y + (x - y))))) - y*x	0	1,24
5	((z + (z + 5.0))/z)/z + 9.0	$3{,}14 * 10^{-4}$	1,06
6	(2.0*sin(x))/cos(y)	0	0,98
7	(x/y)^1.153/y - (exp(y) + (exp(y) + 6.0))	$4{,}33 * 10^{-5}$	1,27

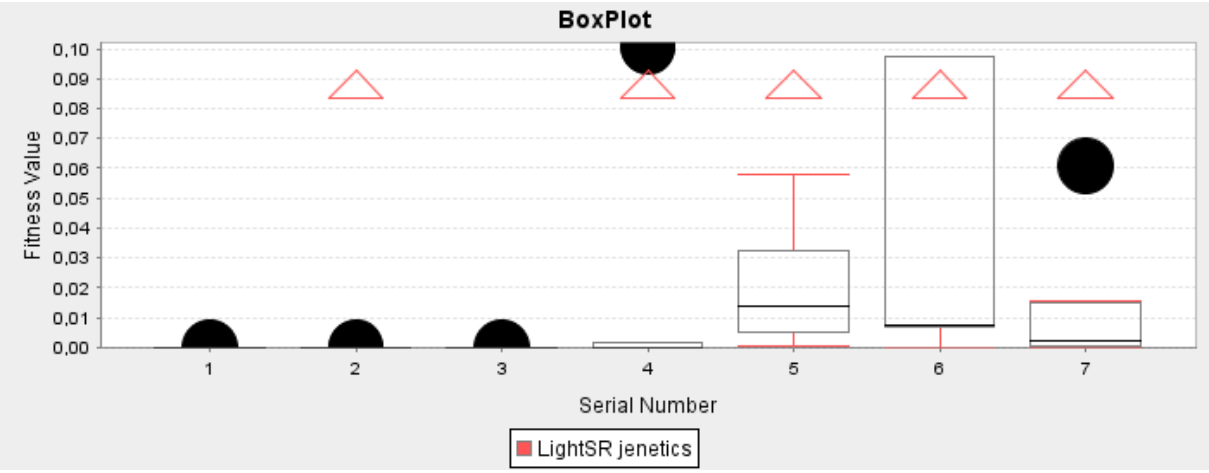
Z výsledků je patrné, že Framework jenetics úplně eliminoval nově přidané neterminály *min* a *max*, což značí kvalitu provedeného algoritmu. Všichni nalezení jedinci mají výslednou fitness hodnotu blízko nule. To znamená, že tyto předpisy kvalitně kopírují trénovací data.

Následující graf představuje průběh vývoje hodnoty fitness vybraného pokusu s využitím frameworku jenetics za použití datové sady číslo 7. Hodnota po 200 uběhnutých generacích dosáhla téměř nuly, dále již docházelo jen k malým zlepšením.



Obrázek 46 - Experiment č. 4 -Průběh fitness jednoduchá SR jenetics

Následující boxplot graf zobrazuje v každém boxu 25 ohodnocení jedinců, kteří byli prohlášeni za nejlepší nalezené řešení v jednotlivých pokusech pro každou datovou sadu.



Obrázek 47 - Experiment č. 4 - Výsledky jednoduchá SR jenetics

7.4.4 Vyhodnocení metoda MultiEngine

Kapitola shrnuje výsledky nashromážděné při spuštění metody MultiEngine naprogramované ve frameworku watchmaker. Hledání optimálního řešení probíhá podle uváděných datových sad a s počátečním nastavením specifickým pro tento experiment.

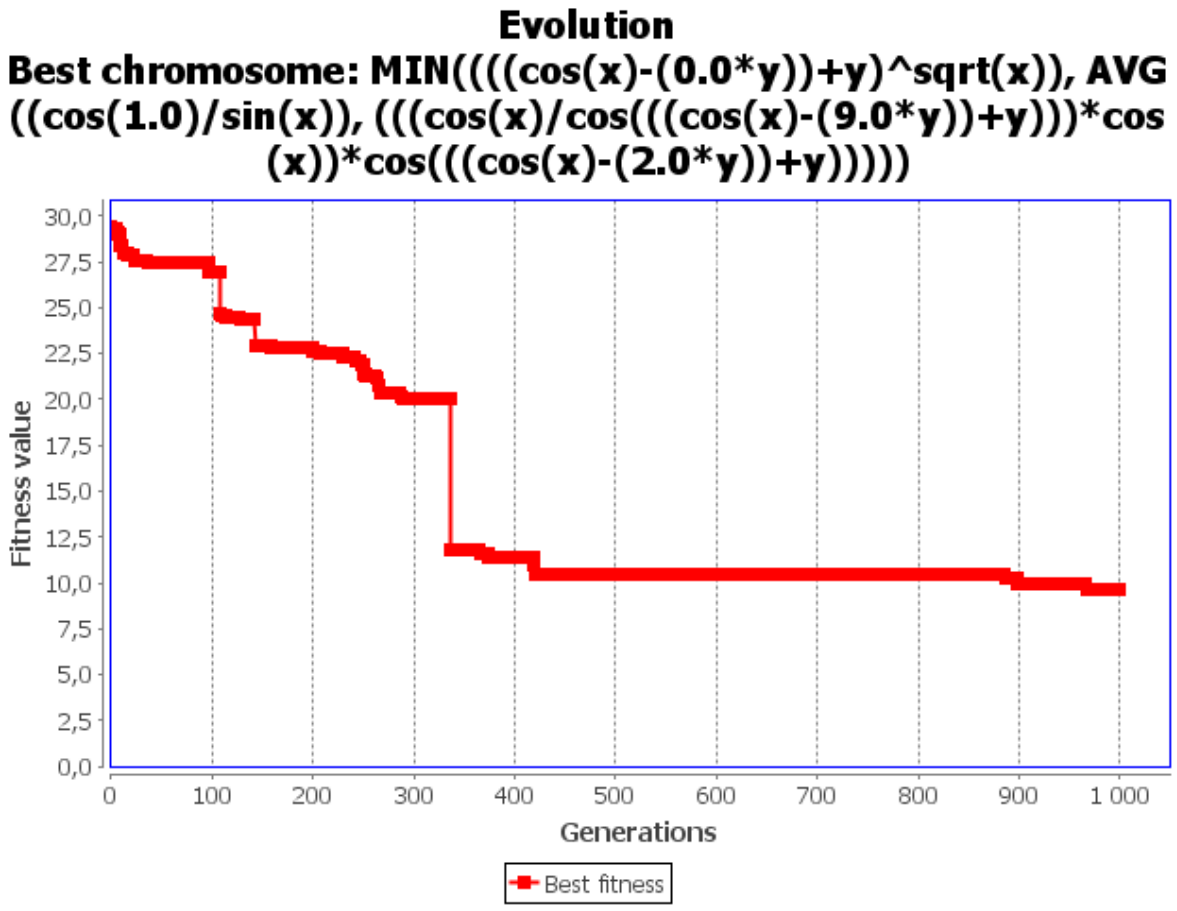
Následující tabulka uvádí nejlepší nalezené jedince a jejich ohodnocení pro každou datovou sadu. Následně je také uveden průměrný čas trvání jednoho pokusu u každé datové sady.

Tabulka 20 - Experiment č. 4 - Výsledky metody MultiEngine

Pořadové číslo	Jedinec	Fitness	Průměrný čas[s]
1	sin(4.0)/cos(2.0)	0	2,5
2	(cos(cos(3.0))^8.0)- ((cos(cos(1.0))^0.0)^((cos(cos(0.0))^2.0)- ((MIN(MIN((9.0*3.0), (x-9.0)), MIN(8.0, AVG(3.0, x)))-cos((((cos(cos(7.0))^2.0) - sin(sin((1.0/3.0)))) +sin(cos((6.0/0.0))))))+(cos(cos(8.0))^2.0))))	0,52	20,73
3	sqrt(sin(0.0)) + sqrt(sqrt(cos(sqrt(((sqrt(sqrt(cos((sin(8.0)^sqrt(x)))))^sin(4.0))^sqrt(y))))))	0,004	53,18
4	(5.0+(((7.0*y)+AVG(((4.0*y)+sqrt((5.0*y))), (((3.0*y)+(6.0*y))+y)))+(MIN(x, x)+x)))-9.0	0	6,17
5	((((((x/y)+5.0)/z)+z)/z)+((0.0/y)+x))+x	0.24	11,11
6	((((((x/y)+5.0)/z)+z)/z)+((0.0/y)+x))+x	$3{,}18 * 10^{-31}$	7,88
7	cos(cos(cos(sqrt(x))))-sqrt(((cos(cos(6.0))- sqrt((cos(y)-cos(x))))-cos(x)))	18,27	9,61

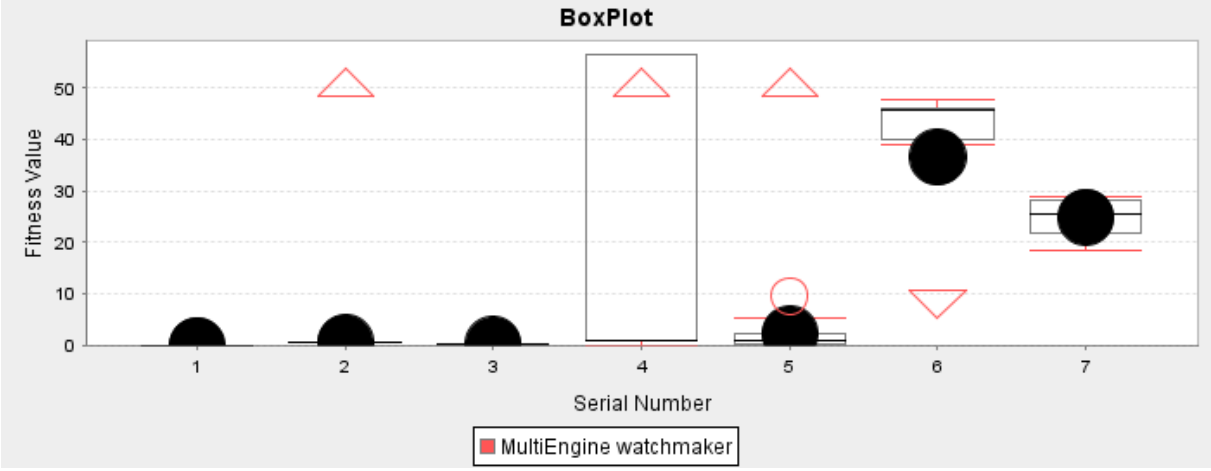
Předcházející tabulka obsahuje výsledné jedince, kteří jsou uvedeni, tak jak byli vygenerováni pomocí metody MultiEngine. Tyto jedince by se ještě dalo zjednodušit pomocí matematických úprav. Po nahlédnutí na výsledné fitness hodnoty lze říct, že metoda nalezla kvalitní jedince, jejichž ohodnocení se ve většině případech velmi blíží nule.

Následující graf popisuje vývoj hodnoty fitness při průběhu jednoho pokusu při hledání optimálního řešení pro datovou sadu číslo 7. K nejvýraznějšímu vývoji fitness docházelo do generace 350. Po zbytek trvání pokusu došlo již jen k mírnému zlepšení.



Obrázek 48 - Experiment č. 4 - Vývoj fitness metoda MultiEngine

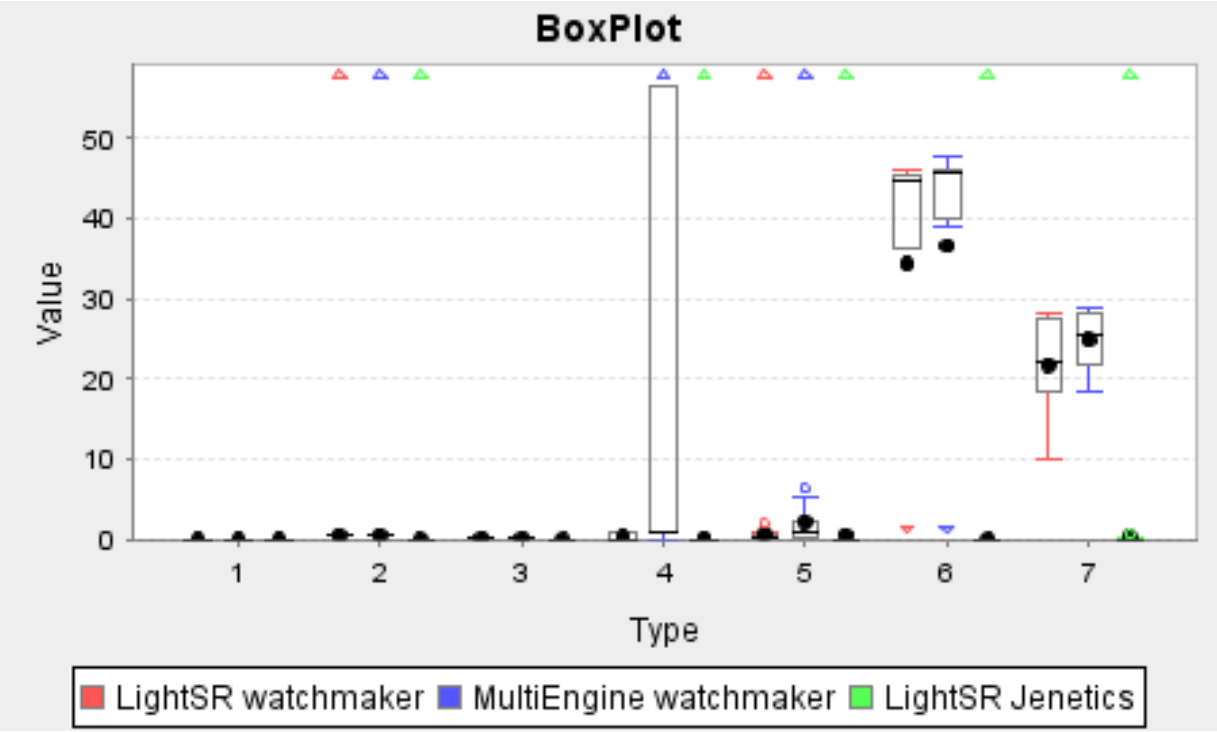
Následující graf zobrazuje výsledky pro všechny uvedené datové sady. Každý box obsahuje 25 hodnot, které představují ohodnocení nejlepších jedinců nalezených v jednotlivých pokusech.



Obrázek 49 - Experiment č. 4 - Výsledky metoda MultiEngine

7.4.5 Vyhodnocení experimentu 4

Následující souhrnný graf porovnává výsledky získané díky zkoumaným metodám u jednotlivých datových sad. Výsledky jsou podobné jako u ostatních experimentů, tedy Framework jenetics dosáhl nejkvalitnějších výsledků. U metody MultiEngine se i zde prokázala přibližně stejná kvalita výsledků jako při řešení pomocí jednoduchého GP.



Obrázek 50 - Experiment č. 4 - Celkový boxplot

7.5 Vyhodnocení

Výsledky nasbírané z jednotlivých experimentů vypovídají o kvalitě řešení jednotlivých metod i přes nepřízeň počátečního nastavení některých experimentů. I v situacích, kde algoritmy neměly k dispozici všechna potřebné údaje a data dokázaly jednotlivé metody nalézt ve většině případů velmi kvalitní řešení problému. V případě řešení pomocí frameworku jenetics se ve většině případů jedna o jedince s fitness hodnotou velmi blízkou nule, což ovšem nezaručuje stoprocentní nalezené řešení. Fitness hodnota je počítána pouze na trénovacích datech. Pro určení skutečného rozdílu od předpisu, pomocí kterého byla vytvořena daná datová sada, by bylo nutné určit odchylku na validační množině.

To samé platí i pro ostatní zkoumané metody, které byly řešeny pomocí frameworku watchmaker. I zde byli nalezeni jedinci, kteří na daných datových sadách dosahovali velmi dobrých výsledků, i když horších než v případě jenetics. Opravdová kvalita nalezených řešení by se projevila až na otestování na validačních datech.

Jelikož jsou trénovací sady vytvořeny jen na menších intervalech a obsahují poměrně málo prvků, je velmi pravděpodobné, že validace mimo tento interval nebude vracet kladné výsledky. K získání kvalitnějších výsledků by tedy bylo nutné do trénovacích množin vložit více záznamů, které budou více rozeseté po definičním oboru. V takovém případě by ovšem velmi vzrostla výpočetní doba algoritmu. Pro testování zkoumaných metod na několika datových sadách a v několika experimentech z různým počátečním nastavením by tedy výpočetní doba vzrostla velmi rychle. Právě proto byly zvoleny poměrně malé trénovací množiny dat.

V porovnání výsledků všech zkoumaných metod nejlepších výsledků vždy dosáhlo jednoduché GP v jenetics. Metody, kde se využívá frameworku watchmaker dosahovaly ve většině případů podobných výsledků. Z jednotlivých boxplot grafů plyne, že v případě experiment číslo 1 byla o trochu lepší metoda MultiEngine a v ostatních případech naopak jednoduché GP. Pro získání prokazatelnějších výsledků by muselo proběhnout více než 25 pokusů tak, aby bylo k dispozici více hodnot pro jednotlivé boxy.

Výsledky tedy v tuto chvílí říkají, že metoda MultiEngine dosahuje přibližně stejných výsledků, jako jednoduché GP. Tato metoda je ovšem připravena na poměrně snadné rozšíření o další vrstvy, nebo úpravu stávajících vrstev. Je možné, že po dalších rozšířeních bude tato metoda dosahovat lepších výsledků, než jednoduché GP.

8 ZÁVĚR

Hlavním cílem práce bylo prozkoumat a popsat oblast umělé inteligence, která se nazývá, hluboké genetické programovaní. Konkrétně se jednalo o zmapování dosud objevených metod a přístupů, a pokus o navrhnutí vlastní metody, která bude splňovat podmínky hlubokého genetického programování. Následným úkolem bylo ověřit funkčnost nové metody v porovnání s klasickými přístupy.

Před uvedením do problematiky hlubokého genetického programování byl stručně popsán vývoj oblastí umělé inteligence, od expertních systémů, přes strojové učení až po hluboké učení. V dalších kapitolách byly popsány nejběžněji používané druhy algoritmů. V prvním případě se jednalo o neuronové sítě. Tato kapitola se věnuje vývoji neuronových sítí od jednoduchého perceptronu až po hluboké neuronové sítě, které úzce souvisí se zmiňovanou problematikou hlubokého genetického programování. Za zmínku zde stojí hlavně konvoluční neuronové sítě, které rozdělují model sítě do vrstev s různými odpovědnostmi (například extrakce a klasifikace).

V dalších teoretických kapitolách jsou popsány algoritmy strojového učení. Prvním z nich je genetický algoritmus, u kterého jsou popsány základní pojmy a principy jednotlivých součástí evoluce. Jedná se zde hlavně o definice jedince, genu a fitness funkce a následných částí, které dohromady tvoří evoluci, tedy selekce, křížení, mutace a další genetické operátory. Dalším z algoritmů strojového učení, který je popsán, je genetické programování. Zde jsou uvedeny rozdíly mezi genetickým algoritmem a genetickým programováním, kde základním rozdílem je typ úlohy, ke kterým se využívá. Genetické programování se využívá k tvorbě počítačových programů, které následně řeší konkrétní úlohu. Kdežto genetické algoritmy se využívají například k numerické optimalizaci nebo rozhodování.

V následující kapitole jsou popsány koncepty hlubokého genetického programování. Ve zkratce se jedná o model, který se snaží vhodně nakombinovat přístupy uvedené v předchozích odstavcích do několikavrstvého modelu. Kapitola také obsahuje popis některých již existujících metod, které jsou považovány za „hluboké“. Důležitou součástí jsou dvě nově navrhnuté metody, a to metoda MultiTree a MultiEngine. Metoda MultiTree bohužel nepřinesla požadované výsledky, jelikož nebylo možné ladit jednotlivé podstromy, které se musely pro účely ohodnocení skládat dohromady, a proto není již dále uváděna. Metoda MultiEngine pracuje s více vrstvami evolucí, které tvoří nové terminály, které

postupují do vyšších vrstev. Tato metoda byla podstoupena implementaci a testování v následujících kapitolách.

Praktická část popisuje implementaci vybraných metod pomocí dvou různých nástrojů a v následné experimentální části jsou jednotlivé metody porovnány. Pro účely testování byl zvolen programovací jazyk Java a v něm připravené frameworky watchmaker a jenetics. Framework watchmaker je určen pro řešení genetických algoritmů, tudíž zde bylo třeba implementovat rozšíření pro genetické programování. Framework jenetics již v základu obsahuje funkcionality i pro tvorbu GP, ovšem omezením je nemožnost rozšíření tohoto frameworku. Je tedy nutné používat pouze funkcionality, které Framework nabízí. V kapitole implementace byly tyto frameworky popsány a byly uvedeny postupy tvorby evoluce. Pro účely testování byly zvoleny jednotné trénovací datové sady, které jsou popsány v tabulce 5. Implementační část následně obsahuje detailnější popis implementace jednoduchého GP a metody MultiEngine v uvedených.

Pro experimentální část byly zvoleny 3 metody, které se porovnávají, a to jednoduché GP v obou frameworcích a následně metoda MultiEngine ve frameworku watchmaker. Framework jenetics dosahuje ve výsledcích značné kvality, tudíž nebylo zvoleno testování na dalších metodách tohoto frameworku. Navíc oproti watchmaker nelze metody v jenetics upravit podle potřeby, tudíž by jejich srovnání s watchmaker nebylo relevantní.

Pro testování byly zvoleny 4 experimenty s odlišným počátečním nastavením algoritmů, a to zejména s různými prvky v množině neterminálů. V některých experimentech nejsou použity některé důležité neterminály, jako *sin* a *cos*, v jiných byly naopak přidány složitější neterminály *min*, *max* a *avg*.

Při porovnávání výsledků experimentů u metod, které jsou naprogramovány pomocí watchmaker si lze všimnout, že u jednotlivých experimentů dosahovaly přibližně stejných kvalit. U Obrázek 29 - Experiment č 1 - Celkový boxplot je vidět, že metoda MultiEngine dosáhla lepších výsledků, ovšem u ostatních experimentů je trochu horší. Tento fakt nemusí znamenat, že metoda nedosáhla hledané zlepšení oproti jednoduchému GP, ale mohl být způsoben hned z několika důvodů. Jedním z nich může být nedostatečná velikost a rozmanitost datových sad. Při rozšíření by se sice velmi prodloužila doba výpočtu, ale nejspíše by se více projevila kvalita jednotlivých řešení. Navíc by tato metoda mohla lépe fungovat na vícerozměrných datových sadách, které obsahují například 8 proměnných. Dalším důvodem může být nedostatečný počet pokusů. Pro každou metodu bylo v rámci

jednoho experimentu spuštěno 25 pokusů. Po zvýšení počtu pokusů by se mohly více projevit rozdíly mezi jednotlivými metodami.

Další možnosti pro zlepšení výsledků metody MultiEngine jsou dle mého zakořeněny v preevoluci. Jelikož se jedná o nově vytvořenou metodu, možnosti zkoumání jsou velmi široké a nabídka úpravy preevoluce jsou takřka nekonečné. Hlavní změnou, která by se mohla projevit kladně na získaných výsledcích, by byla možnost rozložení nově vzniklých terminálů na základní prvky v průběhu evoluce. Algoritmus by tedy například u křížení nebral nový terminál jako jeden nerozdělitelný prvek, ale mohl by ho rozložit na jednotlivé terminály a neterminály a rozdělit. Další možností by mohla být úprava selekčního mechanismu preevoluce. V současné podobě metody se vybírají jedinci zcela náhodně. Změna na turnajové schéma by mohla přinést posun dopředu.

Jelikož se jedná o metodu hlubokého genetického programování, tak určitě základní změnou, která by mohla přinést velké zlepšení pro tuto metodu je přidání dalších vrstev. V tomto případě by bylo možné využít například ladění jedinců podle vah nebo ladění konstant.

Na základě zjištěných výsledků lze konstatovat, že nově nalezená metoda v tuto chvíli nemá přidanou hodnotu oproti jednoduchému GP. Je zde ovšem velmi mnoho možností na úpravu a rozšíření této metody. Po dalším zkoumání rozšíření je možné, že metoda bude dosahovat mnohem lepších výsledků, než jednoduché genetické programování.

9 POUŽITÁ LITERATURA

[1] HYNEK, Josef. Genetické algoritmy a genetické programování. Praha: Grada, 2008. Průvodce (Grada). ISBN 978-80-247-2695-3.

[2] MITCHELL, Melanie. An introduction to genetic algorithms. Cambridge, Mass.: MIT Press, c1996. ISBN 978-0262133166.

[3] POLI, Riccardo, W. B. LANGDON, Nicholas F. MCPHEE a John R. KOZA. A field guide to genetic programming. [S.l.: Lulu Press], 2008. ISBN 978-1409200734.

[4] CHOLLET, François. *Deep learning v jazyku Python: knihovny Keras, Tensorflow*. Přeložil Rudolf PECINOVSKÝ. Praha: Grada Publishing, 2019. Knihovna programátora. ISBN 978-80-247-3100-1.

[5] BLÁHA, Milan. Analýza a hodnocení biologických dat: Umělá inteligence. *Matmatická biologie* [online]. Brno: Masarykova univerzita [cit. 2020-08-12]. Dostupné z: https://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence

[6] *Vymezení umělé inteligence* [online]. Brno: Mendelova univerzita v Brně, 2009 [cit. 2020-08-12]. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=21468

[7] HELIOSADMINISTRATOR. Vše, co jste chtěli vědět o strojovém učení. In: *Blog …podnikání a inovace* [online]. Praha: Asseco Solutions, 2017, 7.12. [cit. 2020-08-12]. Dostupné z: https://blog.helios.eu/cz/clanky/vse-co-jste-chteli-vedet-o-strojovem-uceni/

[8] KURFÜRSTOVÁ, Jana. Strojové učení kouzla zbavené. In: *Edtech Kisk* [online]. Brno: Masarykova univerzita, 16.4.2018 [cit. 2020-08-12]. Dostupné z: https://medium.com/edtech-kisk/strojov%C3%A9-u%C4%8Den%C3%AD-kouzla-zbaven%C3%A9-e066d79ebe51

[9] Strojové učení - Machine learning. In: *Wiki* [online]. 2012 [cit. 2020-08-12]. Dostupné z: https://cs.qwe.wiki/wiki/Machine_learning#Types_of_learning_algorithms

[10] ŠTARHA, Dominik. *Měření podobnosti obrazů s pomocí hlubokého učení* [online]. Brno, 2018 [cit. 2020-08-12]. Dostupné z: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=172305. Diplomová práce. Vysoké učení technické v Brně. Vedoucí práce Ing. Martin Rajnoha.

[11] MATERNA, Jiří. *Deep Learning: budoucnost strojového učení?* [online]. 2013 [cit. 2020-08-12]. Dostupné z: https://blog.seznam.cz/2013/01/deep-learning-budoucnost-strojoveho-uceni/

[12] BAO, Jie. Yoshua Bengio AAAI 2013: Deep Learning of Representations. In: *Speaker Deck* [online]. Fewer and Faster, 2013 [cit. 2020-08-12]. Dostupné z: https://speakerdeck.com/baojie/yoshua-bengio-aaai-2013-deep-learning-of-representations?slide=9

[13] Umělá inteligence. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-08-12]. Dostupné z: https://cs.wikipedia.org/wiki/Um%C4%9Bl%C3%A1_inteligence

[14] KAY, Alexx. CO JSOU TO UMĚLÉ NEURONOVÉ SÍTĚ? In: *Science World* [online]. Praha: fsolutions, 2001 [cit. 2020-08-17]. Dostupné z: https://www.scienceworld.cz/technologie/co-jsou-to-umele-neuronove-site-4077

[15] DURČÁK, Pavel. Neuronové sítě a princip jejich fungování. In: *NaPočítači.cz* [online]. Praha: Verlag Dashöfer, nakladatelství, 1997, 2017 [cit. 2020-08-17]. Dostupné z: https://www.napocitaci.cz/33/neuronove-site-a-princip-jejich-fungovani-uniqueidgOkE4NvrWuNY54vrLeM670eFNQh552VdDDulZX7UDBY/

[16] GLOBEMA.CZ. NAPODOBOVÁNÍ MOZKU: VŠE, CO POTŘEBUJETE VĚDĚT O UMĚLÉ NEURONOVÉ SÍTI. In: *Globema: Inteligentní geoprostorová řešení* [online]. Praha, 2018 [cit. 2020-08-17]. Dostupné z: https://www.globema.cz/umele_neuronove_site/

[17] TRENZ, Oldřich. Neuronové sítě. In: *Mendelu: Univerzitní informační systém* [online]. Brno: Mendelova univerzita v Brně, 2018, 2009 [cit. 2020-08-17]. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=21471

[18] MEDEK, Michal. *Knihovna pro práci s hlubokými konvolučními neuronovými sítěmi v jazyce C#* [online]. Plzeň, 2018 [cit. 2020-08-17]. Dostupné z: https://dspace5.zcu.cz/bitstream/11025/31801/1/mmedek_dp.pdf. Diplomová práce. Západočeská univerzita v Plzni. Vedoucí práce Kamil Ekštein.

[19] POLÁŠKOVÁ, Lenka. *Použití metod hlubokého učení v úlohách zpracování obrazu* [online]. Brno, 2016 [cit. 2020-08-17]. Dostupné z: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=125374. Diplomová práce. Vysoké učení technické v Brně. Vedoucí práce Jan Mikulka.

[20] DOLEŽEL, Petr. *Úvod do umělých neuronových sítí pro studenty technických vysokých škol* [online]. Pardubice: Univerzita Pardubice, 2016 [cit. 2020-03-09]. ISBN ISBN 978-80-7560-022-6. Dostupné z: https://eshop.upce.cz/epub/9004940/

[21] MATERNA, Jiří. Středověk umělé inteligence skončil, seznamte se s neuronovými sítěmi, které umí psát básně. In: *Machine Learning Guru* [online]. 2014, 2015 [cit. 2020-08-17]. Dostupné z: http://www.mlguru.com/cs/basnik/

[22] PILÁT, Martin. Neuronové sítě: RBF sítě a rekurentní sítě. In: *Martin Pilát* [online]. Praha [cit. 2020-08-17]. Dostupné z: https://martinpilat.com/cs/prirodou-inspirovane-algoritmy/neuronove-site-rbf-site-rekurentni-site

[23] SAHA, Sumit. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. In: *Toward Data Science* [online]. 2018 [cit. 2020-08-17]. Dostupné z: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[24] BERKA, Petr. Evoluční algoritmy. In: *Sorry.vse.cz* [online]. Praha [cit. 2020-08-17]. Dostupné z: https://sorry.vse.cz/~berka/docs/izi456/kap_5.5.pdf

[25] Genetický operátor - Genetic operator. In: *Qwe.wiki* [online]. Wikimedia Foundation [cit. 2020-08-18]. Dostupné z: https://cs.qwe.wiki/wiki/Genetic_operator

[26] WILHELMSTÖTTER, Franz. *JENETICS: LIBRARY USER'S MANUAL 5.2* [online]. In: . California, 2020, 2020, s. 146 [cit. 2020-08-17]. Dostupné z: https://jenetics.io/manual/manual-5.2.0.pdf

[27] POLI. *Genetické algoritmy* [online]. , 4 [cit. 2020-08-17]. Dostupné z: http://poli.cs.vsb.cz/edu/isy/down/ga.pdf

[28] MACHÁČEK, Martin. Genetické programování. In: *Posterus: Portál pre odborné publikacie* [online]. 2018 [cit. 2020-08-17]. ISSN ISSN 1338-0087. Dostupné z: http://www.posterus.sk/?p=10198

[29] DANIELBRITTEN a ANDREW1234. Tree Traversals (Inorder, Preorder and Postorder). In: *GeeksforGeeks: A computer science portal for geeks* [online]. 2018 [cit. 2020-08-17]. Dostupné z: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

[30] ZACHA, Jiří. *Konvoluční neuronové sítě pro klasifikaci objektů z LiDARových dat* [online]. Praha, 2019 [cit. 2020-08-19]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/82351/F3-BP-2019-Zacha-Jiri-Konvolucni_neuronove_site_pro_klasifikaci_objektu_z_LiDARovych_dat.pdf. Bakalářská práce. ČVUT. Vedoucí práce Patrik Vacek.

[31] MCCONAGHY, Trent. FFX: Fast, Scalable, Deterministic Symbolic Regression Technology. In: *ResearchGate: Find and share research* [online]. ResearchGate, 2008, 2011 [cit. 2020-08-17]. Dostupné z: https://www.researchgate.net/publication/225965816_FFX_Fast_Scalable_Deterministic_Symbolic_Regression_Technology

[32] DRAHOŠOVÁ, Michaela. *Symbolická regrese a koevoluce* [online]. Brno, 2011 [cit. 2020-05-16]. Dostupné z: http://hdl.handle.net/11012/54137. Diplomová práce. Vysoké učení technické v Brně. Fakulta informačních technologií. Ústav počítačových systémů. Vedoucí práce Lukáš Sekanina.

[33] PERIĆ, Vladimir Peri´. *Symbolic regression as a surrogate model in evolutionary algorithms* [online]. Praha, 2016 [cit. 2020-08-17]. Dostupné z: https://pdfs.semanticscholar.org/8dca/0a1ae5fa50452722b18e71191be06bc8031c.pdf. Diplomová práce. Czech Technical University in Prague. Vedoucí práce Petr Pošík.

[34] *Watchmaker Framework: for Evolutionary Computation* [online]. 2006 [cit. 2020-08-17]. Dostupné z: https://watchmaker.uncommons.org/

[35] SLOUKA, David. 6 jazyků pro programování umělé inteligence. In: *AI world.cz: Denní zpravodajství ze světa umělé inteligence* [online]. 2019 [cit. 2020-08-17]. Dostupné z: https://aiworld.cz/umela-inteligence/6-jazyku-pro-programovani-umele-inteligence-235

[36] SINGH, Aishwarya. A Comprehensive Guide to Ensemble Learning (with Python codes). In: *Analytics Community: Analytics discussions* [online]. 2013, 2018 [cit. 2020-08-17]. Dostupné z: https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models

[37] RAMZAI, Juhi. Holy Grail for Bias-Variance tradeoff, Overfitting & Underfitting. In: *Toward Data Science* [online]. 2019 [cit. 2020-08-18]. Dostupné z: https://towardsdatascience.com/holy-grail-for-bias-variance-tradeoff-overfitting-underfitting-7fad64ab5d76

[38] POLIKAR, Robi. Ensemble learning. In: *Scolarpedia* [online]. [cit. 2020-08-18]. Dostupné z: http://www.scholarpedia.org/article/Ensemble_learning

[39] IBA, Hitoshi. *Bagging, Boosting, and Bloating in Genetic Programming*. Tokyo, 1999. Computer Science. The University of Tokyo. [online]. In: . [cit. 2020-08-18]. Dostupné z: https://pdfs.semanticscholar.org/0591/5cf9e9317da1e2083c8a2514e22d6354cc3e.pdf

10 PŘÍLOHY

Příloha A – Příklad statistického vyhodnocení průběhu jedné iterace experimentu

Experiment number:
Genetic prameters
	Population size:	100
	Max init tree depth:	3
	Max tree size:	20
	Selection propability:	1.0
	Mutation propability:	0.1
	Elite size:	1
Terminal conditions
	Max generations:	1000
	Target fitness:	0.001
Preevolution settings:
	Population size:	5
	Max tree size:	5
	Max init tree depth:	2
	Max generations:	5

Nonterminals
[*,+,/,-,sin,sqrt,^]
Base terminals
[x,y,4.0,x,y,4.0,x,y,2.0]
All terminals
[x,y,4.0,x,y,4.0,x,y,2.0,(y+(x/y)),sin(2.0),(x^x),(((x*3.0)-x)^((x-x)^sin(x))),
((sqrt((y*y))+(y+x))*y)]
--------Evolution result----------
sin((((((sqrt((y*y))+(y+x))*y)+sin(x))+(x+((sqrt((y*y))+(y+x))*y)))+(((sqrt((y*y))+(y+x))*y)
+((x^x)+sin(x)))))
fitness: 19.32835464522232

Evolution Statistics
	generationCount:1000
	crossoverCount:	sum:59042	avg:59
	mutationCount:	sum:9966	avg:9
	invalidsCount:	sum:6278	avg:6
Population Statistics
	populationCount:100
	meanFitness:	NaN
	minFitness:	19.32835464522232
	maxFitness:	Infinity
Time Statistics
	crossoverTime:	sum:0,412790900000 s;	avg:0,000412790900 s
	mutationTime:	sum:0,079297900000 s;	avg:0,000079297900 s
	fitnessTime:	sum:3249350684,563019800000 s;	avg:3249350,684563020000 s

Příloha B – Příklad vytvoření a spuštění GP ve frameworku Watchmaker

```
TreeFactory = new TreeFactory(expData.getTerminals(),
expData.getNonTerminals(), maxInitTreeDepth);

        // operátory pro křížení a mutování stromů
final TreeCrossover = new TreeCrossover(maxTreeSize);

final TreeMutation = new TreeMutation(new
Probability(mutationProp), treeFactory, treeCrossover);

final List<EvolutionaryOperator<Tree>> operators =
Arrays.asList(treeCrossover, treeMutation);

// objekt pro ohodnocení pomocí fitness funkce
FitnessEvaluator<Tree> treeFitnesssEvaluator = new
TreeEvaluator(expData.getInputs(), expData.getOutputs());

// selekční strategie
SelectionStrategy<Object> selectionStrategy = new
TreeTournamentSelection(tourSelProp);

// nastavení enginu algoritmu
GenerationalEvolutionEngine<Tree> engine
                = new GenerationalEvolutionEngine<>(
                        treeFactory,
                        new EvolutionPipeline<>(operators),
                        treeFitnesssEvaluator,
                        selectionStrategy,
                        expData.getRng());

//Ukoncujici podminky
TargetFitness = new TargetFitness(this.targetFitness,
treeFitnesssEvaluator.isNatural());
        Stagnation steadyFitness = new
Stagnation(GeneticParameters.STEADY_FITNESS_VAL, true, true);
        GenerationCount = new GenerationCount(generations);
        TerminationCondition[] terminationConditions =
{targetFitness, steadyFitness, generationCount};

//výsledek
Tree result = engine.evolve(population, eliteSize,
terminationConditions);
```

Příloha C – Sestavení GP pomocí jenetics

```
Sample[] ar = new Sample[expData.getDataset().size()];
Regression reg = Regression.of(
Regression.codecOf(
     expData.getNonTerminals(),
     expData.getTerminals(),
     GeneticParameters.MAX_INIT_TREE_DEPTH, t ->
     t.getGene().size() <
     GeneticParameters.MAX_TREE_SIZE),Error.of(LossFunction::ms
e),expData.getDataset().toArray(ar));
final Engine<ProgramGene<Double>, Double> engine=
Engine.builder(reg)
     .populationSize(GeneticParameters.POPULATION_SIZE)
     .survivorsSize(GeneticParameters.ELITE_SIZE)
     .offspringFraction(1)
     .minimizing()
     .alterers(new SingleNodeCrossover<>(GeneticParameters.
          CROSS_PROB.doubleValue()), new
          Mutator<>(GeneticParameters.MUTATION_PROBABILITY))
     .build();

EvolutionStatistics<Double, ?> stats =
     EvolutionStatistics.ofNumber();

final EvolutionResult<ProgramGene<Double>, Double> result =
     engine.stream()
     .limit(Limits.byFitnessThreshold(GeneticParameters.TARGET_
          FITNESS_VAL))
     .limit(GeneticParameters.MAX_GENERATION)
     .peek(stats)
     .collect(EvolutionResult.toBestEvolutionResult());

ProgramGene<Double> gene =
     result.getBestPhenotype().getGenotype().getGene();

TreeNode<Op<Double>> node = gene.toTreeNode();
```