

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Genetické programování v platformově nezávislém jazyce

Genetic Programming Based on a Platform Independent Language

2016

Zdeněk Gold

**Tuto stránku nahradíte v tištěné verzi práce oficiálním
zadáním Vaší diplomové či bakalářské práce.**

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava*.

Zde vložte text dohodnutého omezení přístupu k Vaší práci, chránící například firemní know-how. Zde vložte text dohodnutého omezení přístupu k Vaší práce, chránící například firemní know-how. A zavazujete se, že

1. o práci nikomu neřeknete,
2. po obhajobě na ni zapomenete a
3. budete popírat její existenci.

A ještě jeden důležitý odstavec. Konec textu dohodnutého omezení přístupu k Vaší práci.

V Ostravě 16. dubna 2009

..... + + +

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 16. dubna 2009

..... + + +

Rád bych poděkoval prof. Ing. Ivan Zelinka, Ph.D. za užitečné rady, zajímavé podměty a věnovaný čas při tvorbě této práce. Rovněž velmi děkuji svým rodičům a dalším rodinným příslušníkům za všeestrannou podporu během studia.

Abstrakt

Klíčová slova: EVT, evoluční algoritmy, genetické programování, syntaktický strom, křížení, mutace, selekce, program, jedinec, symbolická regrese, Java

Abstract

This is English abstract. This is English abstract.

Keywords: EVT, evolutionary algorithm, genetic programming, syntactic tree, cross, mutate, selection, program, individual, symbolic regression, Java

Seznam použitých zkratok a symbolů

EVT	– Evoluční výpočetní techniky
GA	– Genetický algoritmus
GP	– Genetické programování
SR	– Symbolická regrese

Obsah

1	Úvod	5
2	Evoluční algoritmy	6
2.1	Pojmy z oblasti evolučních algoritmů	6
2.2	Životní cyklus evolučních algoritmů	8
2.3	Genetický algoritmus (GA)	9
2.4	Particle Swarm	9
2.5	SOMA	9
3	Genetické programování (GP)	10
3.1	GA vs. GP	11
3.2	Úlohy řešené GP	11
3.3	Reprezentace jedince	12
3.4	Množina funkcí a terminálů	13
3.5	Reprezentace stromové struktury	13
3.6	Generování počáteční populace	13
3.7	Selekce	14
3.8	Křížení	15
3.9	Mutace	18
3.10	Další operátory	18
3.11	Algoritmus GP	20
3.12	Omezení GP	22
4	Implementace	24
4.1	Struktura aplikace	24
4.2	Grafické uživatelské rozhraní	24
4.3	Evoluční procesy	27
4.4	UML model	27
5	Testování na vybraných problémech	28
5.1	Umělý mravenec	28
5.2	Symbolická regrese	30
5.3	Vyhodnocení	32
6	Závěr	36
7	Reference	37

Seznam tabulek

1	Balíčková struktura aplikace EvolTool	26
2	Konfigurace GP pro Sextic problém	32

Seznam obrázků

1	Průběh evolučního algoritmu	8
2	Nějaký graf	12
3	Reprezentace stromu	14
4	Ruletové kolo	15
5	Pořadový výběr	15
6	Příklad operátoru křížení	16
7	Příklad jednobodového operátoru křížení	18
8	Příklad operátoru mutace	19
9	Příklad operátoru editace. Levý strom zjednodušíme a výsledkem je strom pravý	20
10	Průběh algoritmu Genetického programování	21
11	Třídní diagram	25
12	GUI aplikace EvolTool	26
13	Stezka Santa Fe	29
14	Regresní funkce (modrá) s nalezenou approximací (červeně)	31
15	Průběh funkce $x^5 - 2x^3 + x$	33
16	Výsledná approximace funkce $x^5 - 2x^3 + x$	33
17	Průběh funkce $x^6 - 2x^4 + x^2$	34
18	Výsledná approximace funkce $x^6 - 2x^4 + x^2$	34
19	Vývoj fitness nejlepších nalezených řešení Quintic funkce	35
20	Vývoj fitness nejlepších nalezených řešení Sextic funkce	35

Seznam výpisů zdrojového kódu

1 Úvod

Tento text je ukázkou sazby diplomové práce v L^AT_EXu pomocí třídy dokumentů `diploma`. Pochopitelně text není skutečnou diplomovou prací, ale jen ukázkou použití implementovaných maker v praxi. V kapitole ?? jsou ukázky použití různých maker a prostředí. V kapitole 6 bude „jako závěr“. Zároveň tato kapitola slouží jako ukázka generování křížových odkazů v L^AT_EXu.

2 Evoluční algoritmy

Evoluční algoritmy nejsou populární jen proto, že jsou moderní a odlišné od klasických, ale hlavně pro fakt, že v případě vhodného aplikování jsou schopny nahradit člověka.

2.1 Pojmy z oblasti evolučních algoritmů

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

2.1.1 Jedinec

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

2.1.2 Populace

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

Typickým rysem evolučních algoritmů je, že jsou založeny na práci s populací jedinců. Populace může být znázorněna jako matice NxM, kde sloupce představují jednotlivé jedince. Každý jedinec představuje aktuální řešení daného problému. S každým jedincem je navíc spojena hodnota účelové funkce, která říká, jak vhodný je jedinec pro další vývoj populace.

Hlavní činnosti evolučních algoritmů je cyklické vytváření nových populací, tedy nahraď starých populací novými. To vše pomocí přesně definovaných matematických pravidel.

K vytvoření populace je třeba nadefinovat tzv. vzor, podle kterého se generuje celá počáteční populace. Ve vzoru jsou pro každý parametr konkrétního jedince definovány tři konstanty, a to typ proměnné a hranice intervalu, v němž může pohybovat hodnota parametru. Volba hranice je velmi důležitý krok, protože při jejich nevhodném zvolení se může stát, že budou nalezena řešení, která nebudou možné fyzikálně realizovat nebo nebudou mít opodstatnění.

Další neméně důležtý význam hranic souvisí se samotným evolučním procesem. Může se stát, že daný problém bude reprezentován plochou, která bude nabývat lokálních extrémů stále větších hodnot se vzrůstající vzdáleností od počátku. To způsobí, že evoluce bude nacházet stále nová řešení až do nekonečna. Je to způsobeno tím, že evoluční proces směřuje do stále hlubších a vzdálenějších extrémů. Populace je na základě vzorového jedince vygenerována podle vzorce.

$$\Theta(1 + \alpha).$$

Tento vztah zajišťuje, že všechny parametry jedinců budou náhodně vygenerovány uvnitř povolených hranic prostoru možných řešení.

Zobrazení o tom, jak kvalitně proběhla evoluce, se provádí pomocí tzv. historie vývoje hodnoty účelové funkce ve formě jednoduchého grafu. Na něm je vykreslena závislost vývoje účelové funkce na aktuálním počtu jejich ohodnocení. Jde o sekvenci nejhorších a nejlepších řešení z jednotlivých populací. Výhodnější je však zobrazení závislosti hodnoty účelové funkce na aktuálním počtu jejich ohodnocení. To proto, že u evolučních cyklů se provádí u jednotlivých algoritmů různý počet ohodnocení účelové funkce. U prvního případu může být pomalejší konvergence hodnoty účelové funkce zobrazena jako rychlejší a naopak. Skutečná informace o kvalitě evoluce je pak zkreslená. U druhého způsobu můžeme objektivně porovnat různé typy algoritmů bez ohledu na jejich vnitřní strukturu. Kromě vývoje nejlepšího jedince je pak vhodné zobrazovat vývoj i nejhoršího jedince z populace, a to do jednoho grafu.

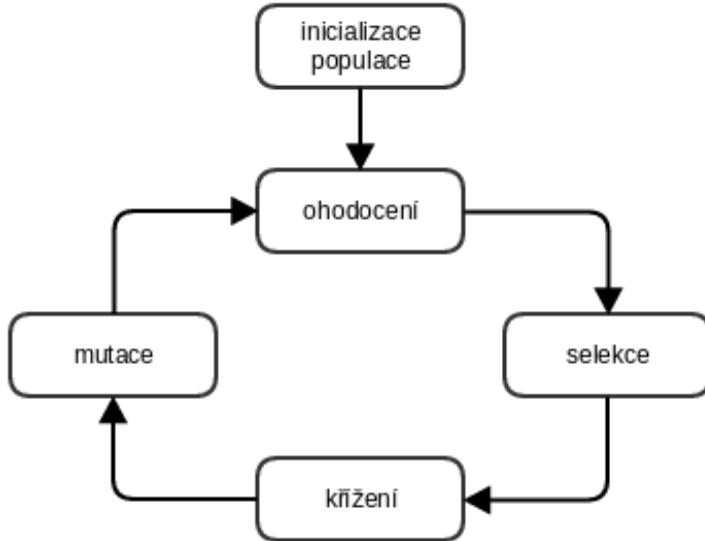
Vývoj populace musí být vždy konvergentní k lepším hodnotám, což znamená, že nemůže nikdy vykazovat divergenci. V daném algoritmu funguje tzv. „elitismus“, který slouží jako jakýsi jednosměrný filtr, jenž propouští do nové populace pouze ta řešení, která jsou lepší či stejně dobrá jako ta ze staré populace.

2.1.3 Účelová funkce

Každého vytvořeného reprezentativního jedince je třeba ohodnotit v rámci celé populace čítající desídky jiných jedinců tak, aby bylo možné některé jedince (řešení) upřednostňovat při výběru pro křížení před jinými (horšími) řešeními. Hodnota takto přidělená jedinci je označována za jeho fitness (neboli vhodnost).

2.1.4 Operátor mutace

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.



Obrázek 1: Průběh evolučního algoritmu

2.1.5 Operátor křížení

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

2.1.6 Elitismus

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potomkům.

2.2 Životní cyklus evolučních algoritmů

Životní cyklus algoritmů z rodiny evolučních algoritmů je v zásadě totožný a lze jej zjednodušeně popsat několika kroky, jak je znázorněno na obr. Na počátku je v rámci námi zkoumané instance problématisky náhodně vytvořena populace, a to z genofondu, který přísluší danému algoritmu.

Nad takto vytvořenou populací je za pomocí fitness funkce (obecné ohodnocení kvality řešení) daného problému každému jedinci přiřazena hodnota, vyjadřující míru vhodnosti

jedince, tzv. fitness. Ohodnocená populace je dále podrobena selekci jedinců vhodných k reprodukci. Selektivní funkce přímo pracuje s přiřazenými fitness hodnotami a na základě těchto hodnot ovlivňuje míru pravděpodobnosti jedinců v populaci.

V reprodukční části algoritmu dochází k výměně genetických informací nově vzniklých jedinců. Typickými operátory je křížení - vzájemná výměna části fenotypů mezi více jedinci a mutace - náhodné pozmenění fenotypů jedince.

2.3 Genetický algoritmus (GA)

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potom

2.4 Particle Swarm

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potom

2.5 SOMA

Podle klasické Darwinové a Mendelovy teorie evoluce, je uznáváno dogma, podle něhož se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří podléhají při svém vzniku mutacím. Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají cyklicky po tzv. generacích, čímž uvolňují místo novým rodičům a jejich potom

3 Genetické programování (GP)

Název "genetické programování"(dále už jen GP) se zrodil již počátkem 80. let, kdy byl představen jako algoritmus pro využití v problémeh jako je predikce, klasifikace, aproximace, tvorba programů. GP je odvětvím evolučních výpočetních technik. Otcem GP je standfordský informatik John Koza [1, 4]. Díky konceptu, který vytvořil, lze využít při tvorbě programů stejných evolučních operátorů, jaké obsahuje GA (křížení, mutace).

V zásadě lze označit GP jako hledání nejlépe ohodnoceného (hodícího se) programu ve skupině přípustných programů, kdy populace požadované velikosti – obvykle několika stovek jedinců – je geneticky křížena a mutována. Jde o program, který umožňuje konstruovat další programy. Může jít o algoritmicky založenou evoluční metodologii, inspirovanou biologickými evolučními procesy (přežití nejschopnějších), ke zjištění, který program nejlépe odpovídá řešení.

Proto, abychom pochopili smysl tohoto algoritmu, položmě si tuto otázku: "Jak se může počítač naučit řešit problémy bez toho, abychom jej k tomu přímo nenaprogramovali? Jinými slovy, jak může počítač sám dělat to co potřebujeme aby dělal, bez toho abychom mu to přesně řekli?" Touto otázkou Arthur Samuel již v roce 1959 otevřel debatu kolem umělé inteligence.

Paradigma genetického programování následuje přístup k řešení problému klasickým genetickým algoritmem s navýšením komplexnosti struktury jedince procházející adaptací. Takové jedince tvoří složitější struktury, které tvoří hierarchicky členěné počítačové programy, vyznačující se různou délkou a tvarem. Jsou to zejména problémy jako umělá inteligence, symbolická regrese, strojové učení, které lze přeformulovat na požadavek nalezení počítačového programu, který bude na výstupu generovat jiné programy jako vstup ke konkrétní úloze. Koza ve své knize tvrdí, že proces řešení těchto problémů lze formulovat jako hledání nejvhodnějšího počítačového programu v prostoru všech možných programů. Prohledávací prostor je tvořen funkemi a terminály odpovídající doméně problému. A právě genetické programování nám poskytuje postup, jak pomocí genetického šlechtění nalézt nejlépe odpovídající program k dané úloze.

Typickým příkladem pro genetické programování je model, který se snaží nalézt logický výraz, jehož výsledkem je buď ano či ne. Můžeme si to představit na modelu banky, která eviduje své zákazníky a jejich úvěry. U každého záznamu zákazníka známe cílový stav (dostal úvěr nebo nedostal)? Nás potom ude zajímat, podle jakých kritérií bychom mohli tento model kategorizovat (čili vytvořit podmínky pro rozhodnutí, kdo úvěr dostane a kdo ne).

Představme si situaci, kdy budeme chtít na základě nashromážděných dat vytvořit systém, který bude tyto data celkem dobře čist a popisovat je. Můžeme si to osvětlit na příkladě s bankou, která vede záznamy o úvěrech spolu s informacemi o věřitelích. Může

se jednat o velikosti rodinného rozpočtu, počtu členů v rodině, počtu pracujících atd. S těmito parametry bychom potom chtěli vytvořit program, který by predikoval, zda je klient vhodným kandidátem na úvěr nebo ne.

3.1 GA vs. GP

Přesto, že je tato technika GP poměrně mladá, patří k nejintenzivněji zkoumaným příspůškům. Jak již bylo dříve zmíněno, vychází z analogie GA. Liší se v účelu použítí a také v reprezentaci jidince.

Genetické algoritmy obecně jsou založeny na myšlence optimalizace hodnot nezávislých proměnných za pomocí adaptivního preparátu, který známe z přírodních zákonitostí [2].

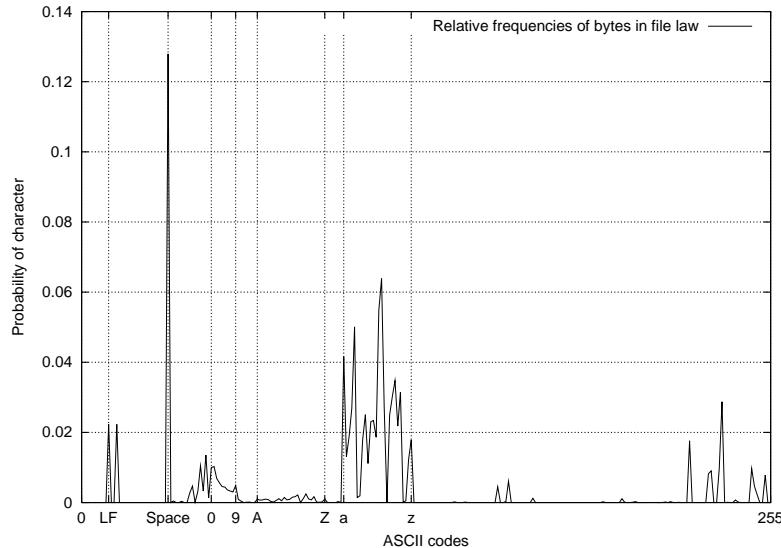
1. GA je založeno na optimalizaci hodnot nezávislých proměnných

Všechny dříve popsané algoritmy vznikly a jsou většinou užitečné v případech, kdy hledáme určitou konfiguraci pro matematický model problému tak, aby bychom dosáhly určitých mezních hodnot. Popusťme však ještě uzdu své fantazii a představme si situaci, kdy budeme chtít řešit problém, u kterého si nejsme jistí, jaký má být správný postup v jeho řešení. V zásadě budeme chtít vytvořit algoritmus, který dokáže generovat jiné algoritmy (či programy), který budou daný problém řešit.

2. Jedinec v GP tvoří řešením pro nějaký model problému, který však už není pouze konfigurace, ale celý algoritmus (program).
3. Populace v GP je tvořena stovkami až tisíci jedinců, kdy každému jedinci odpovídá nějaký program

3.2 Úlohy řešené GP

1. problém umělého mravence
2. nalezení herní strategie
3. generátor náhodných čísel
4. klasifikační problémy
5. nalezení vzoru



Obrázek 2: Nějaký graf

3.3 Reprezentace jedince

První problém, který po zrození myšlenky využít GA pro tvorbu a šlechtění počítačových programů je samotná reprezentace počítačového programu. Tato reprezentace musí být dostatečně obecná pro popsání v různých programovacích jazycích a zároveň zachovávat smysluplnost, syntaktickou správnost a spustitelnost nově vyšlechtěných programů při použití evolučních operátorů mutace a křížení.

Vhodnou reprezentaci programů nalezl Koza v jazyce LISP, který reprezentuje programy jako S-výrazy, což je prakticky syntaktický strom, kterým reprezentují svoje programy překladače. Syntaktický strom je tvořeny dvěma možnými typy uzlů. Jsou to buď neterminály (tvoří funkce) a terminály (proměnné a konstanty). Při definici problémů se specifikuje množina neterminálů $\Pi = \{f_1, f_2, \dots, f_n\}$ a množina terminálů $\Gamma = \{t_1, t_2, \dots, t_n\}$. Příkladem syntaktického stromu, který znázorňuje výraz pro kombinaci k-té třídy z n prvků je na obrázku.

Vlastnostmi se podobá neuronovými sítěmi. Od jiných evolučních algoritmů vyčnívá svou značně velkou populací (čítající tisíce jedinců) a z toho důvodu se jeví, jako značně pomalý algoritmus. Výhodou oproti jiným algoritmům (např. genetický algoritmus), které mají většinou lineární strukturu, zde tvoří jedince ne-lineární chromosomy (např. stromy či grafy).

Funkce f_i z množiny Π tvořící uzly syntaktického stromu jejichž argumenty reprezentují hrany vedoucí do hloubky o jedno větší. Terminály zastupují proměnné či konstanty umístěné v listových uzlech ukončují růst stromu a samotné individuum.

3.4 Množina funkcí a terminálů

Množina funkcí může obsahovat:

1. aritmetické operace (+, -, *, /, atd.)
2. matematické funkce (AND, OR, NOT, NAND, atd.)
3. podmínkové operátory (IF-THEN-ELSE)
4. iterační konstrukce (DO, FOREACH, WHILE)
5. rekurzivní funkce

Aby byl algoritmus GP skutečně úspěšný a efektivní při hledání vhodného řešení, musíme velmi pečlivě definovat množinu použitých funkcí a také terminálních symbolů. Chybí-li v množině funkcí nějaký klíčový krok, můžeme se nám tvorba správného programu zesložitit či v extrémním případě nemusíme dojít k řešení vůbec. Naproti tomu, pokud budeme mít množinu funkcí příliš bohatou, rozšíří se prostor možných řešení a tedy i časová složitost pro prohledání celého prostoru. Může se stát, že i přesto, že je výsledná funkce jednoduchá, bude přesto časově zdlouhavé, abychom toto řešení našli. Je to jako hledat jehlu v kupce sena. Vhodné je omezit funkce na co nejmenší počet.

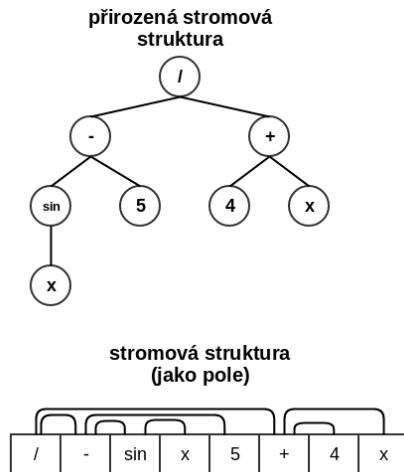
3.5 Reprezentace stromové struktury

Implementací syntaktického stromu je několik. Jednou z možností jak strom v počítacích reprezentovat je s použitím uzlů obsahující seznamy potomků. Jde o přirozené vnímání stromové struktury, která je však při samotné implementaci dosti obtížně manipulovatelná.

Druhou možností je seznam, který představuje strom v prefixovém tvaru zápisu. Pořadí prvků v seznamu odpovídá pořadí navštívení každého uzlu ve stromě, při procházení stromu do hloubky. Zajištění zpětné rekonstrukce stromu nám umožňuje informace o tom, zda prvek patří do množiny terminálních symbolů nebo naopak do množiny funkcí a arita každé funkce. Příkladem takového stromového reprezentace je na obrázku níže.

3.6 Generování počáteční populace

Při prvotním generování populace se budeme zabývat, jakým způsobem lze náhodně vygenerovat několik individuí. Existují dvě metody řídící růst stromu. Úplná (full method), kdy všechny listové uzly stromu mají stejnou hloubku, která je rovna maximální velikosti stromu. Metoda růstová (grow method) naproti tomu dovoluje tvorbu rozmanitějších stromových struktur. Pro generování následného potomka se náhodně rozhoduje, zda je použitý terminál nebo uzel. I zde je limitní maximální hloubka stromu.



Obrázek 3: Reprezentace stromu

Koza ve své knize [1] doporučuje kombinaci obou metod, což v praxi znamená, že pro polovinu populace se použije růstová metoda a pro druhou polovinu úplná. Zároveň rovnoměrným rozložením hloubky stromů mezi všechny jedince, snížíme pravděpodobnost výskytu stejných řešení, která jsou v populaci nevyhovující.

Vše si můžeme ukázat na příkladě populace s $N = 500$ jedinců, kde je stanovena maximální hloubka $h_{max} = 6$. Při generování základní populace budeme postupovat tak, že 100 jedinců bude generováno s $h_{max} = 2$ (50 s růstovou metodou a 50 s úplnou metodou), následně pro dalších 100 řešení generujeme obdobné stromy hloubký o 1 vyšší $h_{max} = 3$ a tak dále, až do hloubky $h_{max} = 6$.

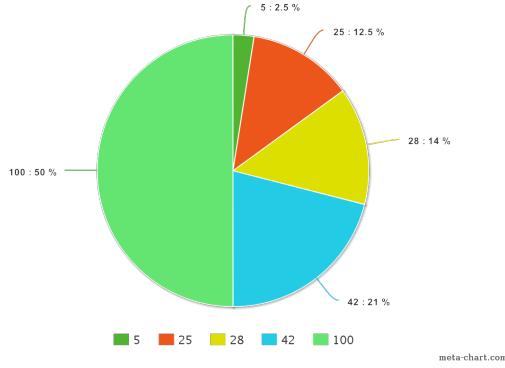
3.7 Selekcce

Cílem operátoru selekce je udžení v nové populaci co možná nejlepší řešení daného problému. Praktikuje se přitom přímá vazba na hodnotu fitness funkce jednotlivých jedinců (řešení). Výběr jedinců z populace je často aplikován algoritmem, založeným na pravděpodobnosti.

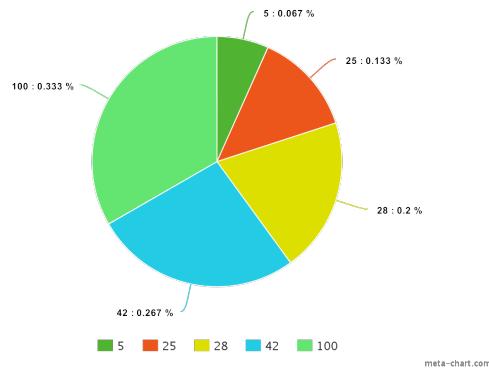
Variant selekce existuje několik. Běžne se v publikacích objevují varianty označované jako *roulette wheel selection*

Roulette wheel selection

V češtině pod názvem výběr ruletovým kolem, je operátor používaný v genetických algoritmů pro výběr potenciálně vhodných řešení pro následnou rekombinaci. K přiřazené fitness hodnotě se úměrně přidružuje pravděpodobnostní hodnota pro výběr jedince k další reprodukci.



Obrázek 4: Ruletové kolo



Obrázek 5: Pořadový výběr

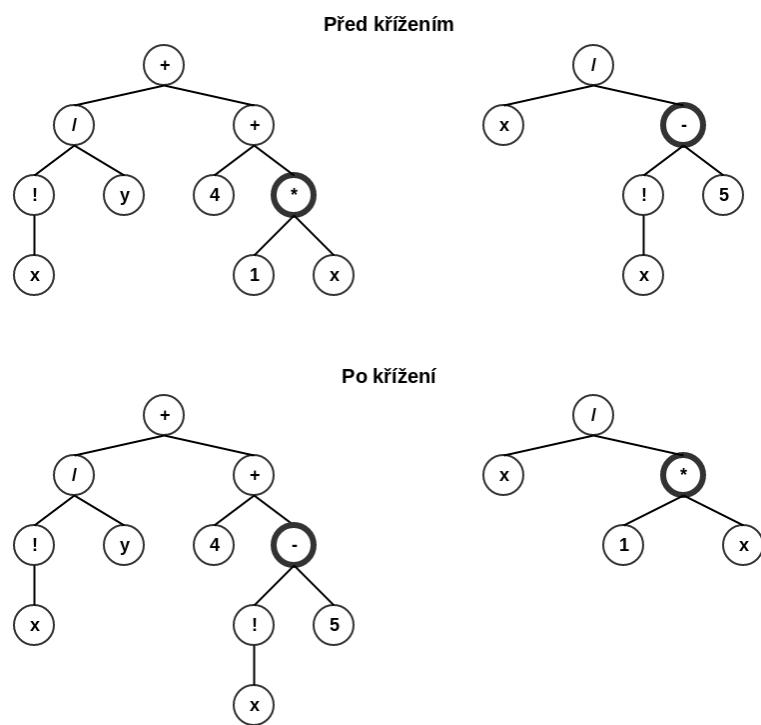
Hodnota fitness je použita k přiřazení pravděpodobnosti pro výběr jednotlivých řešení následujícím způsobem. Pokud je f_i přiřazená fitness hodnota jedince i v populaci, pak pravděpodobnost, že toto řešení bude vybráno rovno $p_i = \frac{f_i}{\sum_{j=1}^n f_j}$, kde n je počet jedinců v populaci.

Rozdělení pravděpodobnosti pro jednotlivé jedince si lze dobře představit na kole s výšeči, kde plocha každé výšeče reprezentuje velikost pravděpodobností každého řešení. Odtud tedy název tohoto algoritmu.

Rank selection

3.8 Křížení

V GP budeme používat stejné operace jako u GA. Operátor křížení pracuje s dvojicemi stromů. U každého stromu z dvojice zvolí náhodně uzel pro křížení a v dalším kroku zamění podstromy pod zvolenými uzly mezi sebou. Více osvětlíme příkladem na obrázku.



Obrázek 6: Příklad operátoru křížení

Oproti GA se zde potkáváme s problémem postupného růstu hloubky reprezentujících stromů (tzv. bloat). V první generaci si obstaráme stromy, který mají definovanou maximální hloubku. Při aplikaci křížení však snad nastává případ, kdy je například listový uzel jednoho stromu nahrazen téměř celým podstromem druhého jedince. V podstatě jen v případech, kdy se budou zaměňovat podstromy z uzlů, které jsou ve stejné hloubce se žádný z obou stromů nerozšíruje. Ve všech ostatních případech bude jeden nebo druhý strom rozšířený. Řešením může být:

- úprava operátorů, aby netvořily dlouhá řešení
- penalizace dlouhých řešení při ohodnocování
- vícekriteriální optimalizace

Koza doporučuje, abychom definovali i maximální hloubku stromů vzniklých křížením. Doporučuje se trojnásobek hloubky počáteční populace. Pokud bychom při křížení překročili tuto maximální povolenou hranici, takto vzniklé potomci budou odmítnuti a nahrazeni jedním z rodičovských jedinců.

3.8.1 Standardní křížení (Tree-based crossover)

Křížení probíhá zcela tím způsobem, že se zcela náhodně vybere z každého stromu jeden uzel a ten se použije k následné výměně souvisejícího podstromu mezi dvěma rodiči. Jde o intuтивně obdobné chování, jaké známe z genetického algoritmu (GA). Názornou ukázkou nalezneme na obrázku 6.

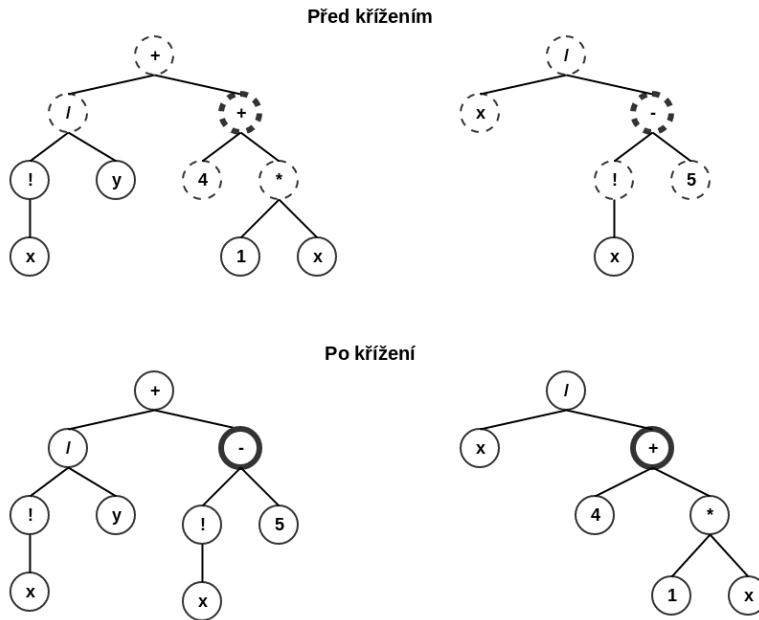
3.8.2 Jednobodové křížení (One-point crossover)

Při biologické reprodukci je genetický materiál obou rodičů vyměňován tak, že výsledné geny dítěte jsou na přibližně stejném místě, jako tomu bylo u rodičovských genů. Což je však zcela odlišné od standardního křížení, kde mohou být podstromy přesunuty na zcela jinou pozici vzhledem k rodičovským stromům [?].

Metody křížení, mající tendenci zachovat polohu genetické informace se nazývají homologní. Tyto homologní metody křížení se běžně používají v lineárním GP.

Fungují tak, že se vybere společný bod křížení v rodičovských reprezentacích, a poté se vymění odpovídající podstromy. Protože oba rodiče mohou mít různý tvar, tak než dojde k výběru bodu křížení, analyzují se oba stromy a pro křížení se vyberou pouze dvojice uzlů, se stejným umístěním. 7.

Speciálním případem při křížení je tzv. "incestrní křížení", které nastane tehdy, pokud do křížení vstoupí dva identiční jedinci. V případě GA je takový případ nežádoucí, protože vznikají identiční potomci. V případě GP k tomu nastává pouze tehdy, když by byl v obou stromech vybrán tentýž uzel pro křížení.



Obrázek 7: Příklad jednobodového operátoru křížení

3.9 Mutace

Operátor mutace také náhodně vybere uzel stromu a celý jeho podstrom odstraní. Na tomto místě se poté náhodně vygeneruje chybějící část stromu do maximální hloubky předepsané parametrem tak, aby vyšledný strom nebyl příliš velký.

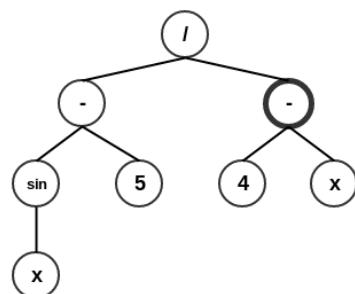
Variant operace mutace je několik. Mezi základní patří tyto:

1. uzlová mutace (point mutation) - neterminální uzel nahradí neterminálem se stejnou aritou a terminální uzel jiným terminálem
2. vyzvedávající mutace (boist mutation) nahrazuje celý syntaktický strom některým z jeho podstromů
3. smršťující mutace (shrink mutation) nahrazuje náhodně zvolený podstrom jediným terminálem

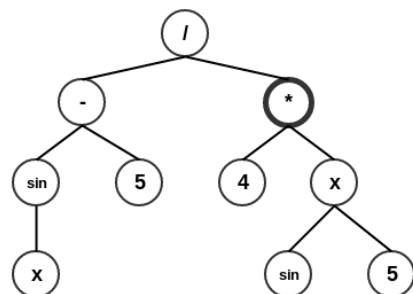
3.10 Další operátory

To však není celý výčet možných operátorů. Existují i operátor, které se v GP nevyužívají zcela běžně, ale stojí za to se o nich zmínit. Mezi méně časté operátory řadíme například operátor permutace, zapouzdření, editace či decimace.

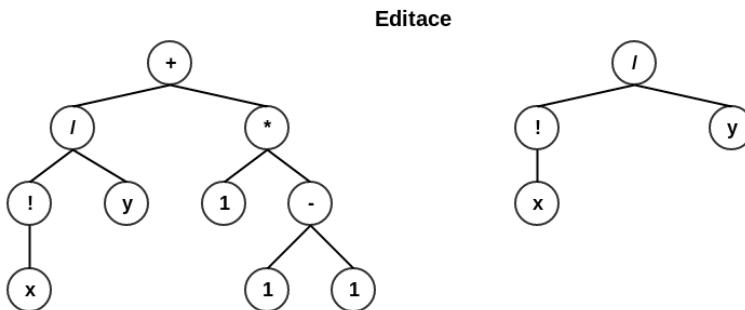
před mutací



po mutaci



Obrázek 8: Příklad operátoru mutace



Obrázek 9: Příklad operátora editace. Levý strom zjednodušíme a výsledkem je strom pravý

Operátor permutace je příkladem asexuálního operátora, který využívá jen jednoho rodiče a náhodně zvolí neterminál s n operátory, jejichž pořadí náhodně prohodí. Jde tedy o náhodné prohození pořadí větví takového podstromu.

Operátor editace se snaží poskytnou nedestruktivní redukci hloubky stromu tak, aby výsledný strom představoval tentýž program, ale bez přebytečných uzlů. Nejlépe, když si takový případ ukážeme na obrázku.

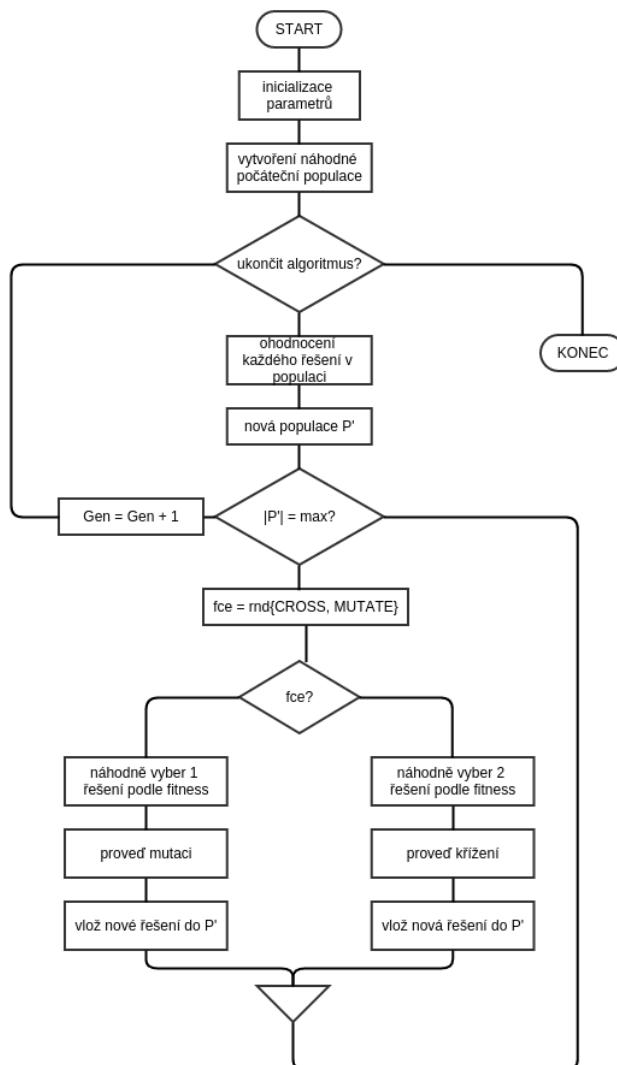
3.11 Algoritmus GP

Jak ilustruje níže uvedený obrázek 10, algoritmus GP lze rozdělit do několika kroků, které budou následně rozebrány v závislosti na způsobu realizace.

Příklad 3.1

Postup evolučního algoritmu

1. Vymezení parametrů evoluce – jako je stanovení kritéria ukončení (počet cyklu, vhodnost řešení, ...), stanovení účelové funkce, případně tzv. vhodnosti. Účelovou funkcí se rozumí obvykle matematický model, jehož minimalizace/maximalizace vede k řešení
2. Generování prvopočáteční populace (obecně matice $M \times N$, kde M je počet parametrů jedince a N je počet jedinců v populaci). Jedincem se rozumí vektor čísel s takovým počtem složek, kolik je optimalizovaných parametrů. Složky jsou nastaveny nahodile a každý jedinec představuje jedno možné řešení
3. Všichni jedinci se ohodnotí přes definovanou účelovou funkci a každému z nich se přiřadí: a) bud' přímá hodnota vrácená účelovou funkcí, nebo b) vhodnost, což je upravená hodnota účelové funkce
4. Nastává výběr rodičů podle jejich kvality



Obrázek 10: Průběh algoritmu Genetického programování

-
5. Křížením rodičů se tvoří potomci. Proces křížení je u každého algoritmu odlišný.
 6. Každý potomek je zmutován
 7. Každý jedinec se ohodnotí stejně jako v kroku 3.
 8. Vyberou se nejlepší jedinci
 9. Vybraní jedinci zaplní novou populaci
 10. Stará populace je zapomenuta a na její místo nastupuje populace nová. Dále se pokračuje krokem 4.
-

Prvním krokem je vytvoření náhodné populace jedinců, která bude sloužit k tvorbě dalších, vhodnějších řešení. Než se dostaneme ke šlechtění nových "lepších" jedinců, je potřeba, ve stávající populaci, nějak rozlišit vhodnější řešení reprezentováno jedincem, od řešení horšího. Proto je iniciační populace ohodnocena fitness funkcí.

Před tím, něž budou řešení v populaci podrobena operátorům křížení nebo mutaci, stanou nejprve před genetickou operací selekce. Program je vybrán na základě dříve zmíněné Darwinovy teorie přírodního výběru. Konkrétně pro nás je to na základě předchozího ohodnocení.

Ohodnocená populace je následně předaná genetickým operacím. Zde bude velmi záviset, s jakými parametry budou operace křížení či mutace operovat. Je třeba parametrem stanovit, jak velká část populace bude podrobena operaci mutace a naopak jak velká část operaci křížení.

3.12 Omezení GP

Koza ve své knize definuje [1] dvojí požadavky na množinu funkcí a terminálů. Je to požadavek uzavřenosti (closure) a postačitelnosti (sufficiency) obou množin. Uzavřenosť množiny je splněna tehdy, pokud může libovolná funkce přijmout jako argument libovoulnou funkci z množiny funkcí či terminál z množiny terminálních symbolů. Uzávřenosť zamezí tvorbu syntakticky nesprávných programů.

Postačitelnost nám naproti tomu umožňuje, abychom byli schopní k danému problému nalézt odpovídající program (funkci), která by jej řešila. Díky tomu jsme schopni říci, že k danému problému můžeme vyjádřit řešení daného problému. Představme si problém nalezení potravy umělým mravencem v mřížkové soustavě kterou můžeme popsat množinou funkcí tvořenou příkazy $\{KROKVPRED, OTOCVLEVO, OTOCVPRAVO\}$ a množinou terminálů s jediným příkazem $\{ZASTAV\}$. I bez dokázání si můžeme jasně říci, že jsme schopní díky této příkazům dostat mravence do jakéhokoliv místa a zastavit.

U jednoduchých příkladů, kde je množina terminálů a neterminálů tvořena pouze Booleovskými funkcemi můžeme i bez důkazů předpokládat, že takové množiny jsou uzavřené a splňují také podmínu postačitelnosti. Ovšem v reálných příkladech budeme často používat programy složené číselných proměnných a funkcích, podmínkách, rekurencích, různých typů v argumentech funkcí. Nebo budeme při práci s číselnými funkcemi muset ošetřit některé zakázané stavy (dělení nulou, logaritmus záporného čísla, atd.). V takových případech už podmínka uzavřenosti nemusí platit.

I v těchto případech se však můžeme odkázat na Kožu, který tvrdí, že lze jednoduchým postupem zajistit, aby byly množiny terminálů a funkcí uzavřené pro obecně jakýkoliv problém. Vezměme si případ programu pracujícího s číselnými funkcemi a terminály, kde může nastat situace, že se bude muset provést dělení nulou. Proto je třeba upravit funkce z těchto množin tak, aby umožňovaly zpracovat také ne definovatelnou hodnotu ("undefined") v podobě nějakého zástupného symbolu a ošetřit stavy funkcí tak, aby funkce mohla vrátit kromě čísla také tuto hodnotu v případech, kdy je operační zakákána nebo v případě, že alespoň jeden z argumentů vrací tuto hodnotu. Takto upravené funkce Koža nazývá chráněné funkce (protected functions).

Vlastnost uzavřenosti množin je vhodná, ale není absolutně nutná. Jako alternativa k chráněným funkcím může sloužit jiný přístup. Pokud striktně nevyžadujeme, aby byl program syntaktického stromu validní, můžeme nesprávně generované stromy v populaci zakázat nebo je jinak penalizovat, aby nepřecházely do nových generací. V této práci se však takovým způsobem nebude zabývat a budeme pracovat pouze se splněnou podmínkou uzavřenosti.

Protože výsledná funkce je tvořena nejen proměnnými, ale také konstantami, je třeba si říci, jak je při tvorbě jedince reprezentovat. Protože můžeme chtít použít GP pro ekonomické modelování, měl by mít algoritmus možnost generovat vhodné konstanty v daném rozsahu a daného typu podle řešené úlohy. Koža navrhuje množinu terminálů o náhodnou konstantou C tak, že se během běhu programu při ohodnocení individuů doplní o náhodnou hodnotu z příslušné množiny. Toto číslo se potom využije jako kterýkoliv jiný terminál.

4 Implementace

4.1 Struktura aplikace

Samotná aplikace je rozdělena do dvou komponent. První část je tvořena knihovnou, která tvoří jádro projektu. Obsahuje samotnou implementaci genetického programování a rozhraní pro komunikaci a řízení běhu evolučního cyklu. Úkolem je umožnit uživateli nebo externímu systému vhodně nastavit běh evoluce a konfigurovat způsob křížení, mutace, reprodukce volbu nastavení parametrů pro testované problémy. V tabulce 4.1.1 je zobrazena balíčková struktura jádra aplikace.

4.1.1 JEvolution

Knihovna JEvolution, která obsahuje implementaci evoluce je tvořena komponentami v podobě charakteristických rysů evolučních algoritmů. Ty jsou tvořeny abstraktními třídami operátorů mutace, křížení, selekce a funkcemi pro ohodnocení a kontejnerovou třídou reprezentující populaci spolu s třídami instance problému. Vztah a vazba mezi komponentami jednotlivých částí cyklu je popsána níže vysvětlujícími obrázky.

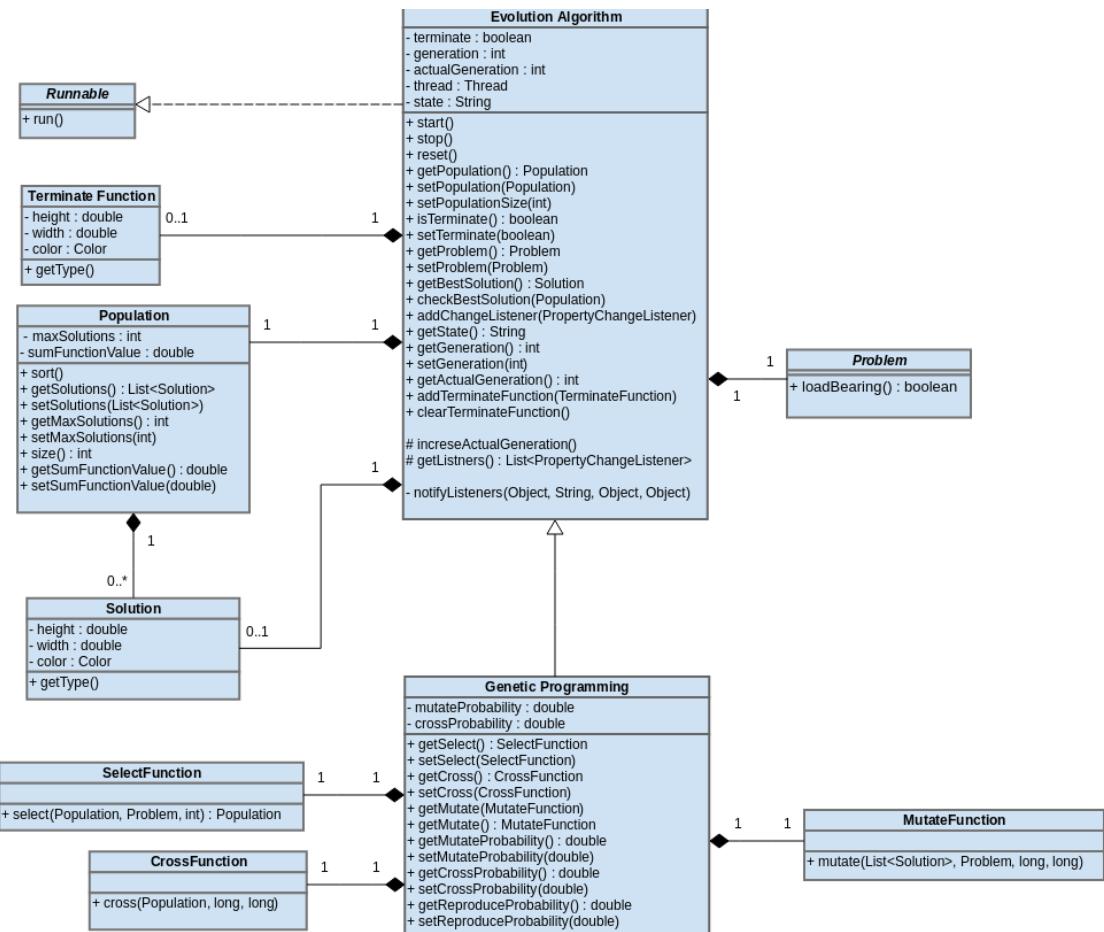
Popis komunikace a vazba mezi komponentami evolučních funkcí

4.2 Grafické uživatelské rozhraní

Tato část je věnována implementaci uživatelského rozhraní v jazyce Java na bázi platormy JavaFX z dílny Sun Microsystems. Platforma využívá pro implementaci kódu návrhový vzor MVC. Modelové třídy jsou obsaženy v balíčku org.evolution.model, struktura vzhledu a sazení ovládacích prvků, panelů a tlačítek je tvořena pomocí XML souborů s příponou .fxml v balíčku resources.gui a nakonec je to samotný řídící kód pro události reagující na uživatelské akce (klik, nastavení hodnoty,...) jednotlivých panelů s ovládacími prvky zase popsané v fxml souborech v balíčku org.evolution.controller.

Evoluční proces volající metody jádra běží v samostatném vlákně s možností parameterizace evoluce. Nějdůležitější části kódu celé aplikace si blíže popíšeme níže.

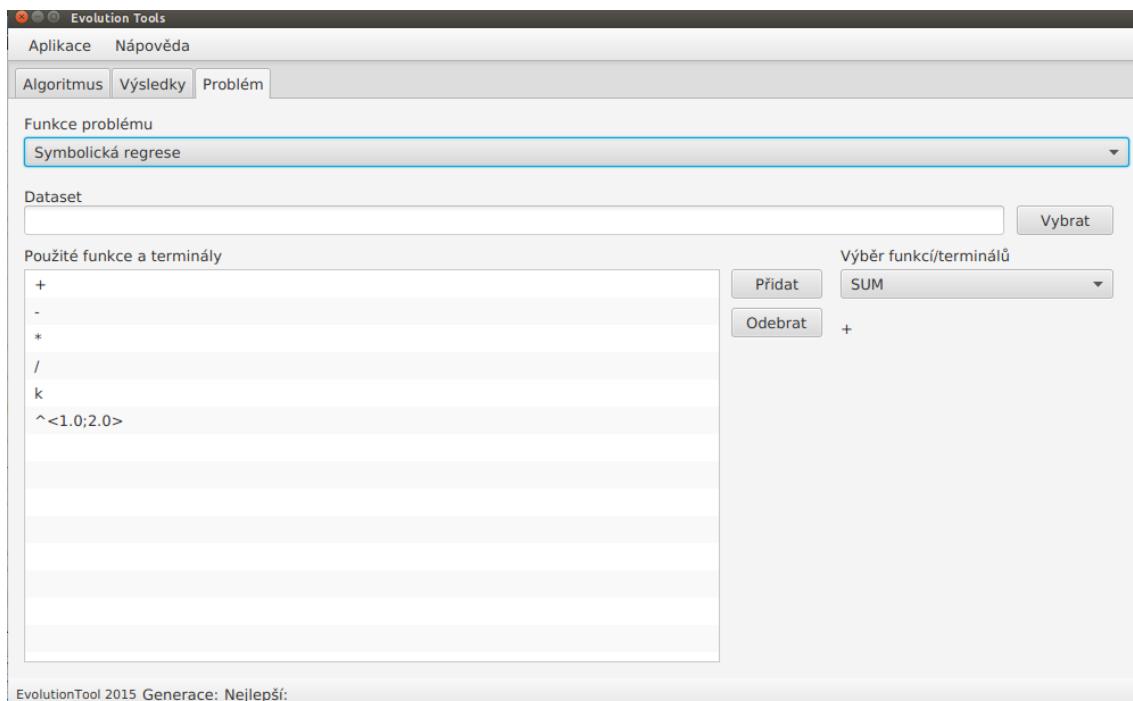
Vstupním bodem aplikace je třída EvolutionTool.java. Je vázána na grafické rozhraní EvolutionTool.fxml. Třída ControlPanel.java řídí inicializaci, spouštění a ukončení evolučního procesu. Položky menu pro uložení a načtení konfigurace testovacích problémů jsou implementovány třídou EvolutionToolController.java. Přidružená grafická podoba je popsána v souboru EvolutionTool.fxml. Je vytvořena jako kontejnerový soubor pro ostatní grafické komponenty. Nastavení konkrétních testovacích problémů je obsaženo v souborech SymbolicRegressionProperty.fxml a ArtificialAntProperty.fxml s řídícím kódem v třídách SymbolicRegressionProblemProperty.java a ArtificialAntProblemProperty.java.



Obrázek 11: Třídní diagram

<i>org.evolution.algorithm</i>	abstraktní evolučních algoritmus se společnými metodami pro evoluci
<i>org.evolution.cross</i>	implementace operátorů křížení
<i>org.evolution.mutate</i>	implementace mutačních operátorů
<i>org.evolution.problem</i>	instance reprezentující jednotlivé problémy k testování genetického programování
<i>org.evolution.problem.regression</i>	pomocné třídy pro problém symbolické regrese
<i>org.evolution.select</i>	implementace operátoru selekce k výběru vhodných jedinců z populace
<i>org.evolution.solution</i>	Třídy pro reprezentaci jedinců v populaci (stromová struktura, chromozomy, ...)
<i>org.evolution.solution.type</i>	jednotlivé typy genomu tvořící syntaktický strom
<i>org.evolution.terminate</i>	implementace operátoru ukončení evolučního procesu
<i>org.evolution.util</i>	pomocné třídy ke zpřehlednění kódu a ulehčující práci implementaci

Tabulka 1: Balíčková struktura aplikace EvolTool



Obrázek 12: GUI aplikace EvolTool

4.3 Evoluční procesy

Evoluční procesy, do kterých řadíme "přirozený výběr", "mutaci", "křížení" a další techniky, které jsou inspirované zákonitostmi v přírodě jsou základním kamenem pro jakýkoliv program napodobující evoluční chování za pomocí počítačů. Proto je třeba dbát o co nejvhodnější a nejpreciznější implementaci tak, aby byl výsledný algoritmus dobře použitelný i pro nejrůznější problémy. Představme si nyní některá úskalí, která je třeba dobře postihnout a zefektivnit výsledný kód.

4.3.1 Výběrová funkce

V evolučních technikách se využívá vždy a ovlivňuje populační vývoj, protože předchází evoluční operátory (křížení, mutaci). Tato funkce se má stát přirozeným výběrem, který v přírodě funguje již od pradána. Tedy právo silnějšího (odolnějšího, přizpůsobivějšího) jedince na tvorbu nové generace.

Několik pravidel pro efektivnější výběrovou funkci:

1. Zamezení výběru dvou totožných jedinců pro křížení
2. Možnost vícenásobného výběru stejného jedince (posílení vlivu lepších řešení do dalších populací)

4.4 UML model

Před tím, než se pustíme do popisu toho, co jsou to evoluční algoritmy a na jakých principech fungují, se seznámíme s tím, co jsou to „evoluční výpočetní techniky“.

5 Testování na vybraných problémech

Po představení pestré, ale zdaleka né kompletní, škály algoritmů je třeba si představit a vyhodnotit jejich sílu na problémech, které se objevují v mnoha publikacích jako vhodní adepti. V následujících pasážích si představíme ty nejběžnější avšak netriviální problémy. U každé definice problému je popsáno, jakými evolučními algoritmy je vhodné tyto problémy řešit. Spolu s definicí problému si popíšeme i výsledky, které byly během testování zaznamenány a konfigurace s jakými byly algoritmy spouštěny. Podtržením toho všeho bude srovnání s výsledky, které byly zveřejněny v publikacích jako optimální, přičemž nemáme za cíl dosáhnout co nejlepšího řešení, ale orientačně vyhodnotit použitelnost na problémech s přihlédnutím k tomu, že výsledným produktem této práce bude aplikace sloužící jako opora při úvodu do problematiky a výuky evolučních algoritmů a hlavně genetického programování.

5.1 Umělý mravenec

Běžně uváděným problémem pro genetické algoritmy je úkol navést umělého mravence tak, aby se pokusil nalézt všechnu potravu ležící podél nepravidelné stezky. Cílem problému je nalezení konečného automatu, který by v konečném čase dosáhl daného problému. [10].

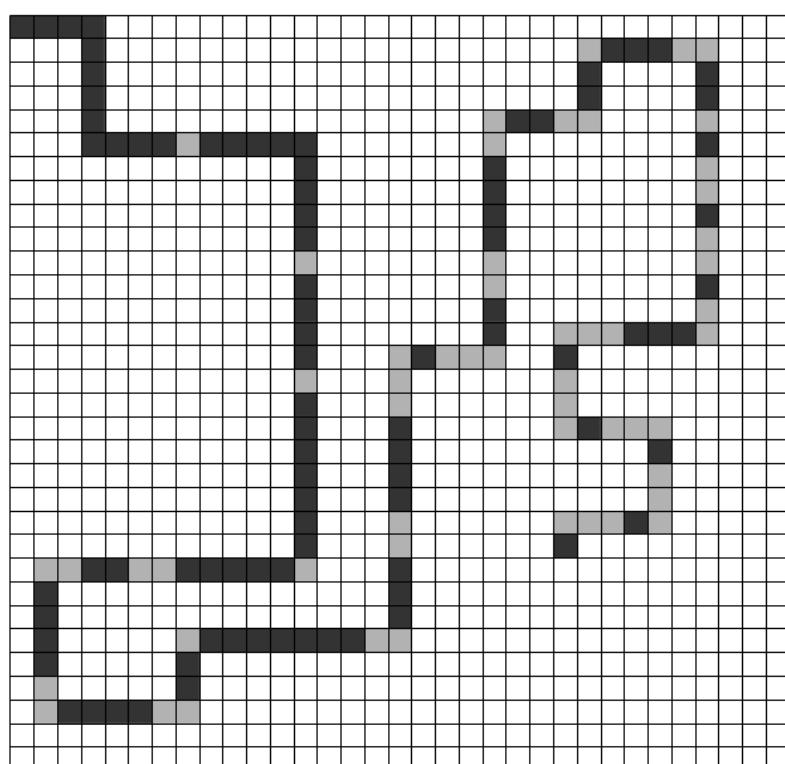
Umělý mravence se pohybuje po dvourozměrné toroidní mřížce 32x32 polí tak, že začíná v levém horním rohu na souřadnicích (0,0) a směruje po "Santa Fe"stezce až na konec. Na této stezce se přitom vyskytuje 89 nepravidelně rozložených polí s potravou. Mimo polí s potravou se na této stezce vyskytuje množství mezer délky až 3 buňky, kde potrava není, a to nejen na rovných úsecích, ale také v rozích.

Celá stezka je zobrazena na obrázku. Jídlo reprezentují tmavá pole. Spolu s jídlem jsou na stezce mezery, které jsou na obrázku v šedé barvě.

Umělý mravenec má velmi omezený pohled na svět kolem. Má senzor, kterým může vidět pouze na sousední buňku bezprostředně následují před mravencem a může provádět pouze některé z následujících primitivních akcí:

- **RIGHT** otočení mravence o 90 doprava
- **LEFT** otočení mravence o 90 doleva
- **MOVE** pohyb mravence o jedno políčko vpřed, ve směru natočení

Pro problém budeme také potřebovat zpracovávat informace z vnějšího světa pomocí malého senzoru mravence. Rozumným přístupem jak toho dosáhnout je použití podmíněného operátoru pro rozvětvení programu instrukcí a také operátoru pro sekvenční provádění instrukcí.



Obrázek 13: Stezka Santa Fe

- **IF-FOOD-AHEAD** přijímá dva argumenty a spustí první v případě, že se bezprostředně před mravencem nachází potrava, jinak se vykoná instrukce z druhého argumentu
- **PRG2** přijímá dva argumenty, které se sekvenčně po sobě vykonají
- **PRG3** přijímá tři argumenty, které se sekvenčně po sobě vykonají

5.2 Symbolická regrese

Symbolická regrese (dále jako SR) se v publikacích popisována jako statická metoda používaná k předpovídání hodnot nějaké proměnné, závislé a jedné nebo více nezávislých proměnných. Snaží se odhalit vnitřní vztahy v datovém souboru. Jde tedy o inverzní postup pro získání symbolického tvaru (původního předpisu funkce) na základě tohoto souboru dat - množina hodnot z definičního oboru funkce.

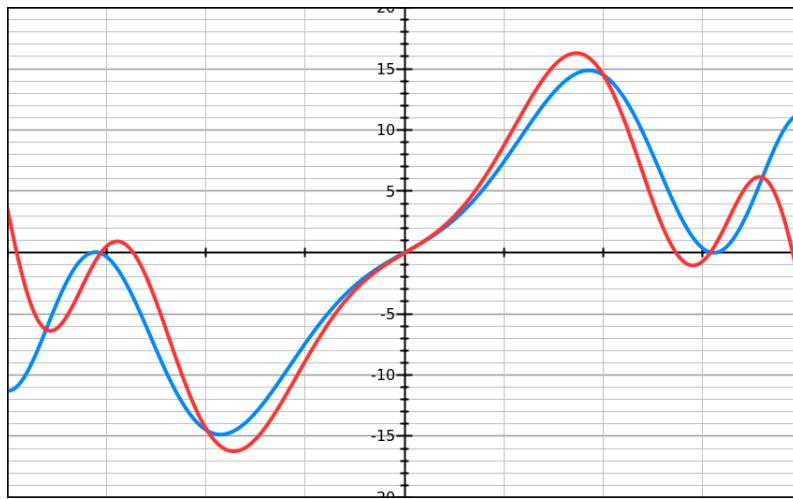
Obecně může jít o jakoukoliv funkci např. $y = \sin(x) + 1/e^x$. Symbolická regrese je jeden z možných způsobů, jak najít vhodnou regresní funkci pro zadaná data.

Regresní funkce může být zkonstruována kombinací elementárních funkcí, jako jsou matematické operátory: sčítání (+), odčítání (-), násobení (*), dělení (/), goniometrické funkce: sinus ($\sin(x)$), cosinus ($\cos(x)$)), proměnné: x, y, z, \dots , a konstant: a, b, c, \dots

V GP je cílová regresní funkce konstruována a upřesňována během evolučního procesu. Na začátku se v závislosti na velikosti populace a matematických výrazech, proměnných, respektive konstantách vytvoří počáteční populace, kde každý jedinec představuje jednu z možných regresních funkcí.

Fitness funkce, která řídí vývoj modelů bere v úvahu nejen chybové metriky (s cílem zajistit modely přesně předpovědět data), ale také ...

Zatímco běžné regresní techniky se snaží optimalizovat parametry pro předem specifikované modelové struktury, symbolická regrese žádné modelové struktury nepředpokladá, a místo toho vyvazuje tento model ze samotných dat. Jinými slovy, se pokouší zjistit, jak modelové struktury, tak parametry. Tento přístup má samozřejmě nevýhodu v mnohem větším prostoru k prohledání - ve skutečnosti, je nekonečný nejen prostor pro modely symbolické regrese, ale existuje nekonečné množství modelů, které se perfektně hodí na konečná množina dat (za předpokladu, že složitost modelu není uměle omezena).



Obrázek 14: Regresní funkce (modrá) s nalezenou approximací (červené)

Velikost prohledávaného prostoru, a tak i časovou náročnost pro nalezení modelu lze omezit tím, že omezíme sadu stavebních bloků poskytnutých algoritmu na základě stávajících znalostí systému, který produkoval data. Nicméně, tento rys symbolické regrese má také výhodu - evoluční algoritmus využívá rozmanitosti generování modelu, a tak účinně prozkoumává prostor možných řešení. V konečném výsledku je proto pravděpodobné, že z výběru navržených modelů, které celkem optimálně approximují datový soubor, můžeme poskytnout lepší vhled do systému, a umožňuje uživateli identifikovat approximaci, která lépe vyhovuje potřebám (co do přesnosti a jednoduchosti).

Symbolická regrese má široké využití. Mezi nejčastější možnosti aplikace bychom mohli zařadit:

1. Syntéza logických obvodů
2. Syntéza neuroových sítí
3. Syntéza trajektorií robotů
4. Aproximace funkcí

5.2.1 Aproximace funkcí

Pro otestování životaschopnosti a správnosti implementovaného algoritmu GP bylo vytvořeno rozhraní také pro approximaci neznámého modelu funkce za pomocí předložených dat. Cílem bylo nalezení modelu funkce k dosažení co možná neoptimálnějšího proložení dat.

K testování byly použity polynomy pátého a šestého řádu, tzv. Sextic a Quintic problémy a dvojice trigonometrických funkcí s značením 3Sine a 4Sine problémy [3].

Datový soubor	tvoře 50 vzorky s krokem 0,4 a rozmezí [-1;1]
Velikost chyby	Suma absolutních hodnot rozdílů mezi původní hodnotou a hodnotou syntetizované funkce
Množina funkcí	$+, -, *, \text{pow}(a, b)$
Množina terminálů	konstanty s nastavenou mezí [-10;10] a nezávislá proměnná x
Počet generací	500 evolučních cyklů
Velikost populace	200 jedinců
Výška stromu	6
Pravděpodobnost křížení	0.7
Pravděpodobnost mutace	0.28
Pravděpodobnost reprodukce	0.02
Metoda křížení	Tournament (velikost 10)

Tabulka 2: Konfigurace GP pro Sextic problém

5.2.2 Quintic a Sextic problem

Problém označovaný jako Quintic a Sextic problem jsou označením pro všechny polynomiální funkce pátého a šestého řádu. V této práci se jednalo o funkce definované vztahy 1 a 2.

$$x^5 - 2x^3 + x \quad (1)$$

$$x^6 - 2x^4 + x^2 \quad (2)$$

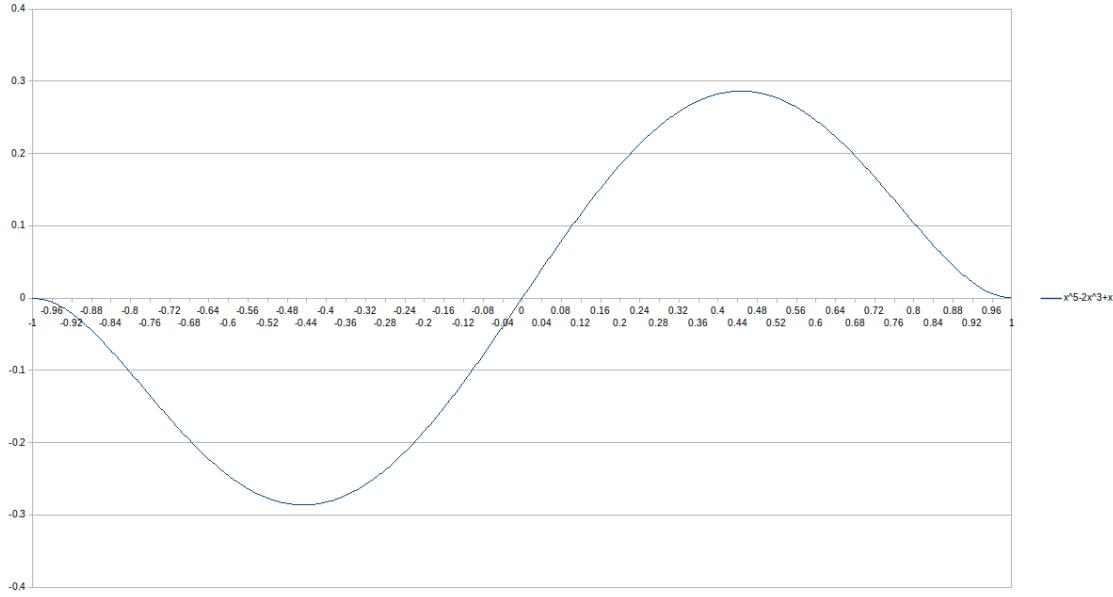
Pro účely testování navržené aplikace bylo použito následujících nastavení evolučního běhu. Aby nebyly výsledky testování zkreslené, byla jednotlivá syntéza funkcí opakována 20 krát. Konfigurace k approximaci Quintic a Sextic funkce je uvedená v tabulce 5.2.2

Simulace approximace pro oba problémy byla opakována 20-krát. Na obrázku jsou zobrazena data a jejich proložení vsemi syntetizovanými funkcemi. Lze si všimnout, že všechny nalezené syntetizované funkce celkem věrohodně approximují původní funkce.

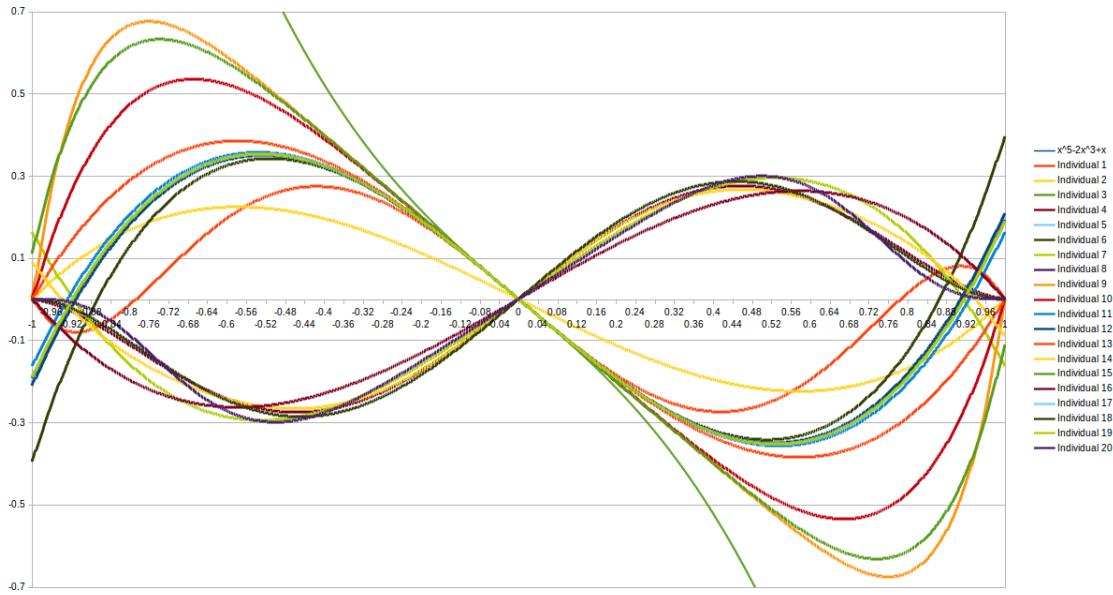
Mezi nejlepší approximaci funkce Sextic patřily výsledné funkce $(x^3 - x) * (x^3 - x)$ a $(x^3 - x) * (x^3 - x)$, které po úpravě (roznásobení) skutečně dávají původní funkci 2. Stejně tak při hledání optimální funkce Quintic 1 problemu bylo během opakovaného běhu programu dosaženo optimálních řešení v podobě funkce $x - (((x+x)*(x*x)) - (x^5))$

5.3 Vyhodnocení

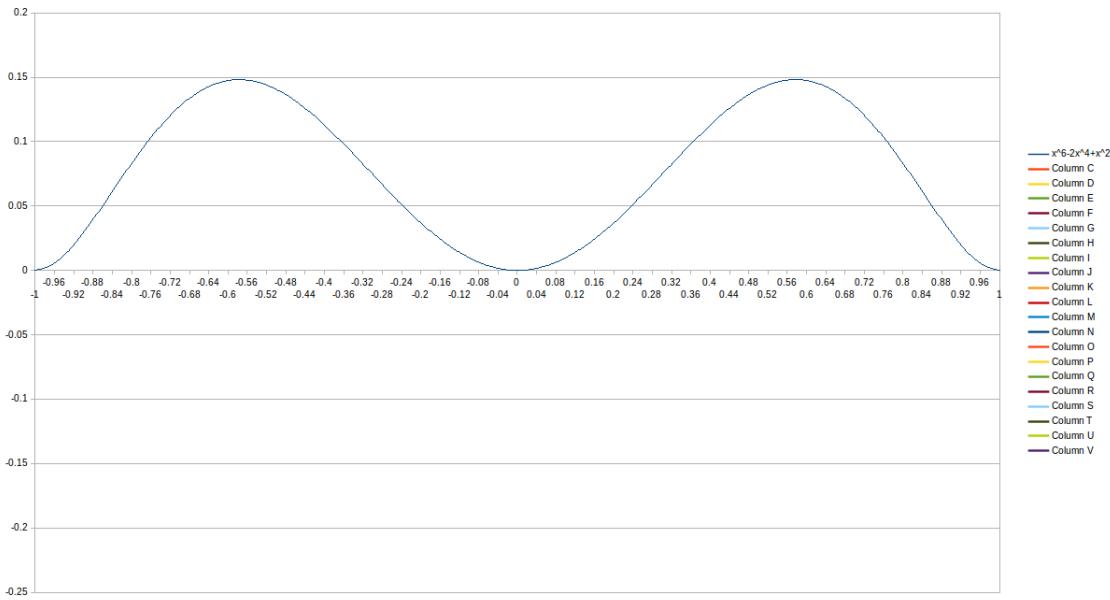
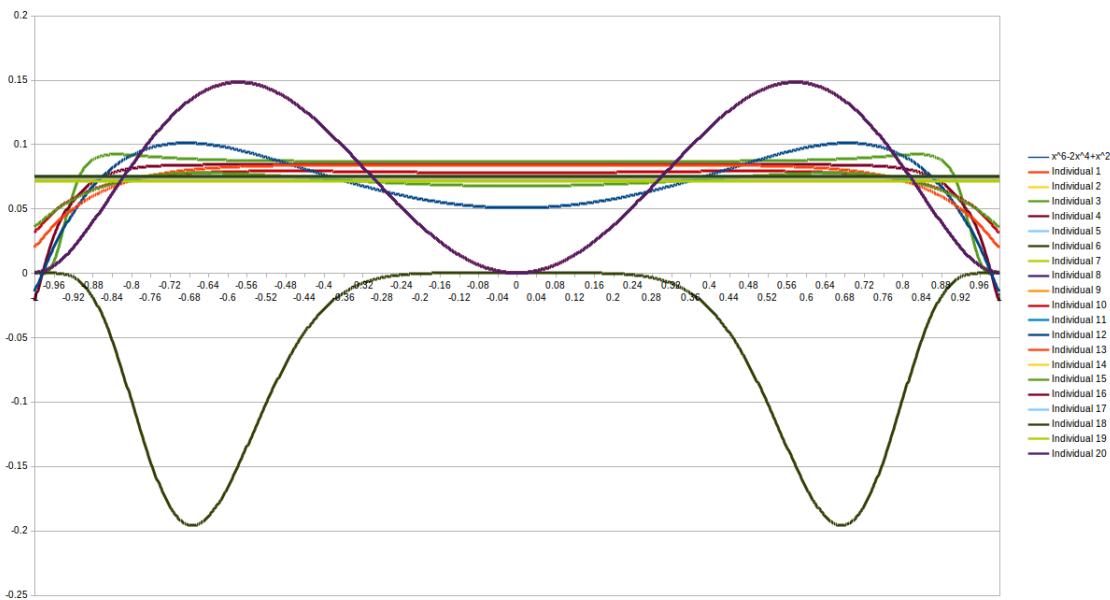
Nějaké zhodnocení

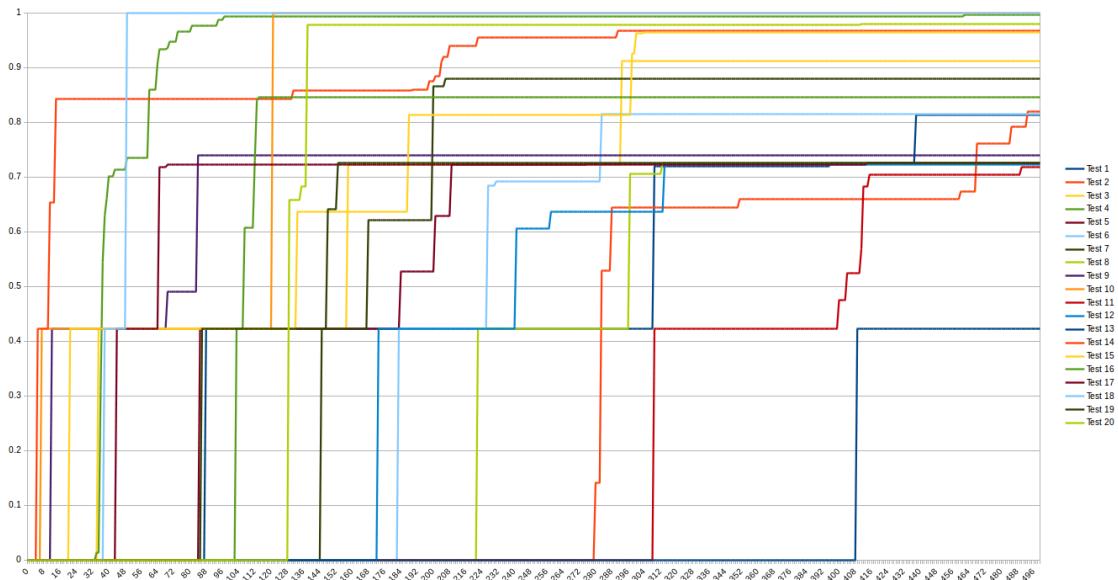


Obrázek 15: Průběh funkce $x^5 - 2x^3 + x$

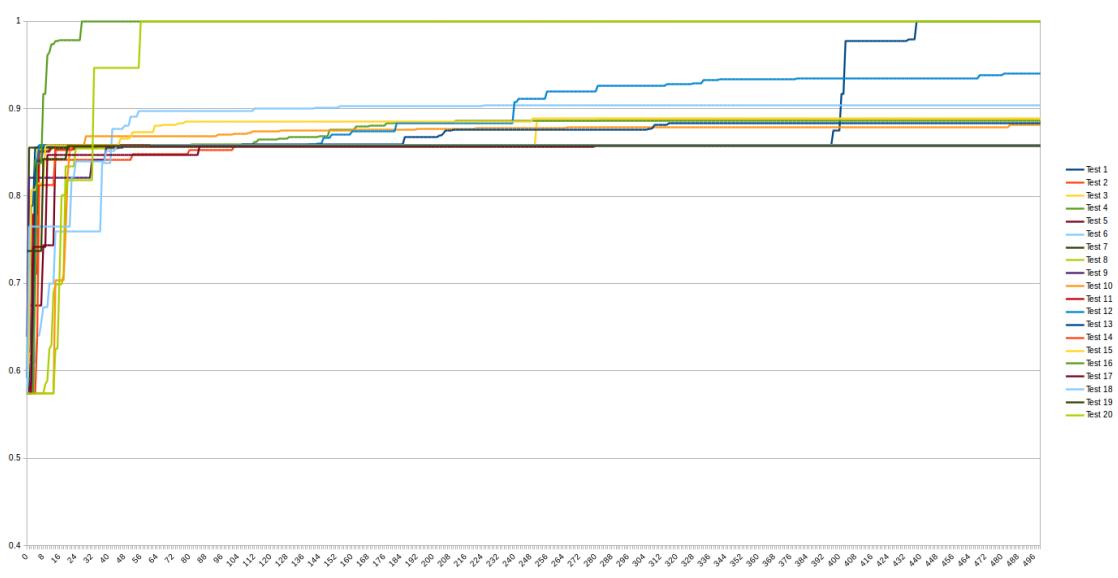


Obrázek 16: Výsledná aproksimace funkce $x^5 - 2x^3 + x$

Obrázek 17: Průběh funkce $x^6 - 2x^4 + x^2$ Obrázek 18: Výsledná aproximace funkce $x^6 - 2x^4 + x^2$



Obrázek 19: Vývoj fitness nejlepších nalezených řešení Quintic funkce



Obrázek 20: Vývoj fitness nejlepších nalezených řešení Sextic funkce

6 Závěr

Tak doufám, že Vám tato ukázka k něčemu byla. Další informace najdete v publikacích [5, 6].

Zdeněk Gold

7 Reference

- [1] Koza, J. R., *Genetic programming. On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.
- [2] Dostál, M., *Evaluční výpočetní techniky*, Olomouc, Přírodovědecká fakulta: 2007.
- [3] Zelinka, I., Oplatkova, Z., Šeda, M., Ošmera, P., Včelař, F., *Evaluční výpočetní techniky, principy a aplikace*, Praha, BEN - technická literatura: 2009.
- [4] Koza, J. R., *Genetic programming II. Automatic Discovery of Reusable Programs*, Cambridge, MA: MIT Press, 1994.
- [5] Goossens, Michel, *The L^AT_EX companion*, New York: Addison, 1994.
- [6] Lamport, Leslie, *A document preparation system: user's guide and reference manual*, New York: Addison-Wesley Pub. Co., 2015.
- [7] Koza J.R, *Genetic Programming*, MIT Press, ISBN 0-262-11189-6, 1998
- [8] Koza J.R, Bennet F.H., Andre D., Keane M., *Genetic Programming III*, Morgan Kaufmann pub., ISBN 1-55860-543-6, 1999
- [9] Lampinen Jouni, Zelinka, Ivan, *New Ideas in Optimization and Mechanical Engineering Design Optimization by Differential Evolution. Volume 1*, London: McGraw-Hill, 1999. 20 p. ISBN 007-709506-5
- [10] Koza, John R., *Genetic Programming On the Programming of Computers by Means of Natural Selection, Volume 6*, London, England: The MIT Press, 1998. 819 p. ISBN 0262111705, 9780262111706
- [11] Langdon W. B., Poli R., *Foundaments of Genetic Programming*, Amazon: Springer, 2001. 253 p. ISBN 3-540-42451-2