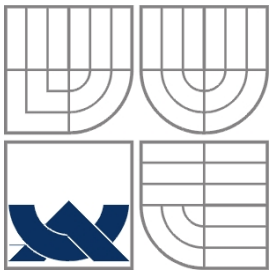


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

GRAMATICKÁ EVOLUCE – JAVA

GRAMMATICAL EVOLUTION – JAVA

DIPLOMOVÁ PRÁCE
DIPLOMA THESIS

AUTOR PRÁCE **Bc. PAVEL BEZDĚK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADOMIL MATOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2009

ZADÁNÍ ZÁVĚREČNÉ PRÁCE

(na místo tohoto listu vložte originál anebo kopii zadání Vaší práce)

ABSTRAKT

Předmětem mé diplomové práce je realizace gramatické evoluce v programovacím jazyce Java pro řešení úloh aproximace funkcí a syntézy logických obvodů. Aplikace je prakticky využita pro testování a sběr dat v kontextu s použitím odlišných účelových funkcí a paralelní gramatické evoluce. Data jsou analyzována a vyhodnocena.

ABSTRACT

The object of my thesis is the realization of grammatical evolution in the Java programming language for solving problems of approximation of functions and synthesis of logical circuits. The application is practical used for testing and gathering data in context of using different purpose function and parallel grammatical evolution. The data are analyzed and evaluated.

KLÍČOÁ SLOVA

Java, gramatická evoluce, aproximace funkcí, syntéza logických obvodů.

KEYWORDS

Java, grammatical evolution, approximation of functions, synthesis of logic circuits.

PODĚKOVÁNÍ

Velmi rád bych poděkoval vedoucímu práce, panu Ing. Radomilu Matouškovi, Ph.D., za metodické vedení, přátelskou pomoc, neocenitelné rady a inspiraci.

OBSAH

1 Úvod

Relativně dlouho byla problematika optimalizace řešena dnes již klasickým matematickým aparátem, který umožňuje nalezení optimálního řešení pro problémy jednoduššího charakteru a pro složitější problémy většinou řešení suboptimální. To že klasické optimalizační metody jsou méně úspěšné pro určitou třídu úloh, překračující určitý stupeň obtížnosti, implikuje fakt, že je potřeba mnohem výkonnějších metod, které by ulehčili řešení složitých optimalizačních úloh.

V posledním čtvrtstoletí vznikla množina algoritmů nového typu, tzv. evolučních algoritmů, jejichž jméno vychází z filozofie, na jejímž základě byly tyto algoritmy vyvinuty. Na tyto algoritmy je kladen požadavek velmi dobré znalosti optimalizované problematiky a správně nadefinované účelové funkce. V práci se zabývám tzv. gramatickou evolucí, která je jednou z nejnovějších metod spadajících do evolučních algoritmů a může být chápána jako typ genetického programování založeného na gramatice. Pomocí gramatické evoluce můžeme vytvořit programy v libovolném jazyku, pokud použijeme Backus-Naurovu formu.

Gramatická evoluce nepracuje s jedním kandidátem řešení daného problému, ale s celou skupinou různých řešení, přičemž tato řešení jsou v gramatické evoluci reprezentována binárními řetězci. Na skupinu těchto řešení jsou aplikovány tři základní operace, selekce, křížení a mutace, přičemž některé z nich mají více variant, jednotlivé typy těchto operací a jejich varianty jsou podrobně popsány v kapitole s názvem „Gramatická evoluce“.

S vývojem evolučních technik a rozvojem teoretické informatiky se ukázalo, že ruku v ruce s evolučními technikami půjdou i nové počítačové technologie založené na paralelizaci. Aplikační část práce integruje podporu pro souběžné úlohy prostřednictvím svých vláken, kde vlákno je jednotkou provádění, respektive aplikace umí danou úlohu řešit vícekrát, souběžně a nezávisle, přičemž využívá potenciál více procesorů.

V kontextu s podporou využití více procesorů a moderními počítačovými technologiemi jsem implementoval paralelní gramatickou evoluci patřící mezi tzv. paralelní evoluční algoritmy, které dovolují řešit větší a složitější problémy, přičemž často vedou rychleji k řešení a současně konvergují k lepším výsledkům. Modely paralelní gramatické evoluce, které jsem implementoval, jsou podrobněji popsány v rámci kapitoly „Gramatická evoluce“.

Aplikační část implementuje dvě úlohy použití gramatické evoluce, tzv. aproximaci funkcí, neboli prokládání naměřených dat vhodnou křivkou, a tzv. syntézu logických obvodů, která se zabývá návrhem logického obvodu, jestliže známe jeho chování. Kapitoly s názvem „Aproximace funkcí“ a „Syntéza logických obvodů“ podrobněji rozebírají podstatu těchto úloh a kapitola s názvem „Implementace v jazyce Java“ podrobněji popisuje způsob implementace, třídy a jejich metody, přičemž tuto kapitolu lze chápat jako druh programátorské dokumentace.

V kapitole s názvem „Testy a statistika“ jsem pro testování úlohy aproximace funkcí zvolil dva problémy, data prvního jsou generována polynomem a data druhého trigonometrickou funkcí. V testech sleduji jednak jak si gramatická evoluce vede při aproximaci výše uvedených vygenerovaných dat a jednak vliv hodnotícího kritéria. Dále jsem testoval přínos paralelní gramatické evoluce na výsledná řešení a vliv změny jejich parametrů.

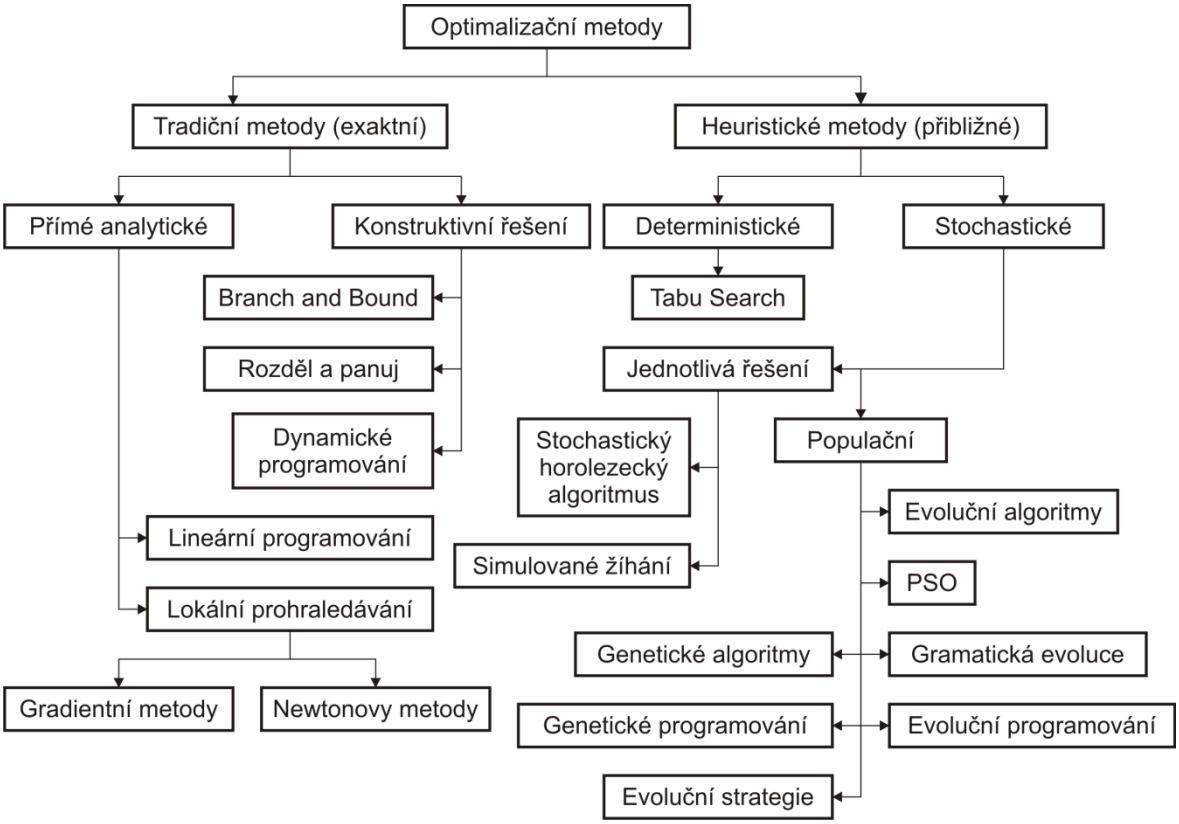
V poslední kapitole s názvem „Aplikace na reálných problémech“ jsem vyzkoušel gramatickou evoluci, respektive realizovanou aplikaci, na skutečných problémech. Pro úlohu aproximace funkcí jsem se pokusil navrhnout model výnosu cibule pro dvě oblasti jižní Austrálie, dále jsem řešil problém aproximace píku. Pro úlohu syntéza logických obvodů jsem navrhnul logický obvod úplné binární sčítačky.

2 Úvod do problematiky EVT

Většina problémů inženýrské praxe může být definována jako optimalizační úloha, přičemž řešený problém lze převést na matematickou úlohu danou vhodným funkčním předpisem, jejíž optimalizace vede k nalezení argumentů účelové funkce. Pro úspěšné řešení takových problémů byla v posledním čtvrtstoletí vyvinuta třída velice výkonných algoritmů, které umožňují řešit velmi složité problémy efektivním způsobem. Tato třída algoritmů se nazývá evoluční algoritmy.

Z hlediska nejobecnějšího členění evoluční algoritmy patří k algoritmům heuristickým, které můžeme dále rozdělit na deterministické a stochastické. Deterministické algoritmy na stejný vstup reagují vždy stejně a v každém jejich kroku je vždy jednoznačně definován krok následující, algoritmy stochastické se liší v tom, že některé jejich kroky využívají náhodné operace a to znamená, že výsledky řešení se v jednotlivých běžích programu mohou lišit.

Evoluční algoritmy lze podle jejich strategie rozdělit do dvou tříd na metody založené na bodové strategii (simulované žíhání, horolezecký algoritmus a zakázané prohledávání) a metody založené na strategii populace (gramatická evoluce a genetické programování).



Obr. 1: Příklad možného dělení optimalizačních metod [Vesterstrom, Riget, 2002].

Evoluční algoritmy umějí řešit optimalizační problémy, jakými například mohou být nalezení optimálního nastavení vah neuronových sítí, optimálního nastavení parametrů regulátoru nebo optimální distribuce materiálu ve výrobě, za předpokladu, že lze daný problém převést na matematický daný vhodným funkčním předpisem. Evoluční algoritmy umějí takovéto, mnohdy velmi složité, problémy řešit efektivním způsobem tak elegantně, že se staly velmi oblíbenými v mnohých oblastech využití i díky jednoduchosti s jakou lze evoluční algoritmy naprogramovat, např. v obyčejném tabulkovém procesoru.

2.1 Historie EVT

Začátek historie evolučních algoritmů se obvykle datuje do poloviny 70. let, kdy se poprvé objevili genetické algoritmy, případně do poloviny let 60., kdy byly poprvé s úspěchem použity tzv. evoluční strategie. Původními otci evolučních výpočetních technik jsou osobnosti jako A. M. Turing a N. A. Barricelli. [4]

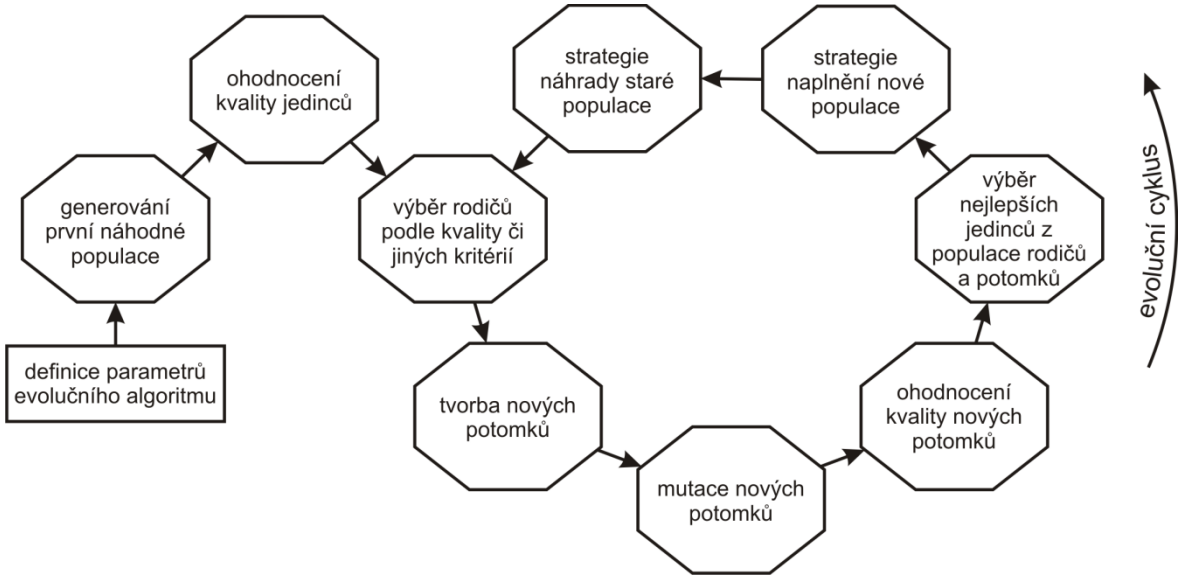
Stopy využití některých kroků inspirovaných přírodními jevy, které se obecně považují za součást evoluce, nalézáme v první polovině 60. let, kdy tři studenti univerzity v Berlíně, kteří se zabývali konstrukcí převodovek, přišli s myšlenkou náhodně kombinovat dvojice existujících konstrukcí ve snaze nalézt konstrukci s lepšími užitnými vlastnostmi. Tato operace beze sporu připomíná přírodní křížení v rámci téhož biologického druhu. Jiným, tentokrát již plně vědomým, příkladem využití evoluce pocházejícím z druhé poloviny 60. let, tedy z doby plné optimistických představ o schopnostech umělé inteligence a počítačů vůbec, je evoluční programování. Zde šlo o evoluci chování konečných automatů a vyvinutí jejich schopnosti predikovat změny prostředí, v němž se automat nachází.

Po dalších zhruba deseti letech se setkáváme v literatuře se zcela novým pohledem na problematiku evolučních technik. Americký teoretický biolog John Holland vydává svoji slavnou knihu (Holland, 1975), ve které shrnuje svůj dlouhodobý výzkum snažící se algoritmicky odpovědět na původně jednoduchou otázku: „Proč se navzájem liší jedinci patřící k témuž biologickému druhu?“. Tato kniha položila základ genetickým algoritmům, které byly později rozvíjeny, modifikovány a aplikovány na širokou třídu úloh nejrůznějšího charakteru.

V knize Davida Goldberga (Goldberg, 1989) se setkáváme s genetickými algoritmy systematicky studovanými z technického pohledu, tedy z pohledu, který se snaží chápat genetické algoritmy jako techniku obecně aplikovatelnou na širokou třídu úloh. Mezi další významné publikace patří knihy Johna Kozy (Koza, 1992, 1994) zakladatele genetického programování a kniha Gramatická evoluce od M. O’Neilla a C. Ryana. Tyto knihy úzce navazují na genetické algoritmy, jejichž cílem je automatizované generování počítačových programů řešících určitou konkrétní úlohu nebo skupinu úloh. Mezi další studie patří optimalizační algoritmy inspirované chováním skutečných biologických systémů, například tzv. mravenčí kolonie (ACO).

2.2 Základní charakteristiky EVT

Evoluční výpočetní techniky jsou numerické algoritmy, které vycházejí ze základních principů Darwinovy a Mendelovy teorie evoluce, podle které se jednotlivé druhy vyvíjejí tak, že jsou z rodičů plozeni potomci, kteří při svém vzniku podléhají mutacím.



Obr. 2: Obecný cyklus evolučního algoritmu.

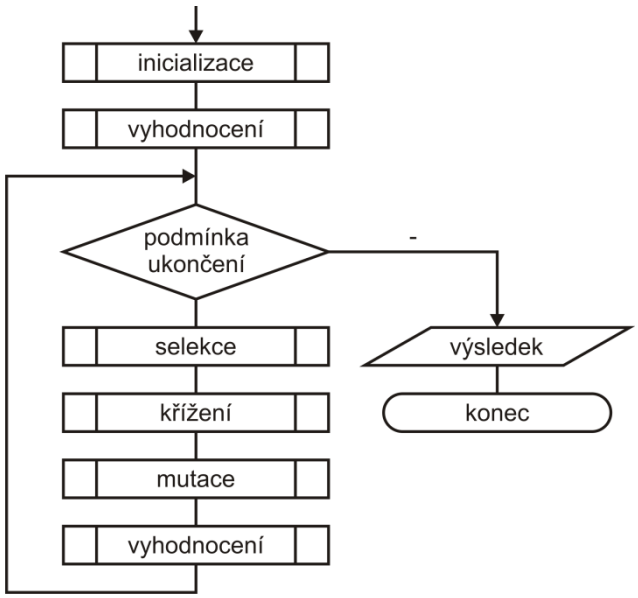
Rodiče a potomci nevhodní pro aktuální životní prostředí vymírají po tzv. generacích, čímž uvolňují místo novým lepším potomkům, kteří se stávají novými rodiči. Tuto charakteristiku dobře vystihuje citát z publikace (Davis a Steenstrup, 1987):

„*V přírodní evoluci je základní úlohou biologického druhu vyhledávání výhodných adaptací vůči složitému a dynamicky se měnícímu prostředí. 'Znalost', která charakterizuje každý biologický druh, byla získána vývojem a je shrnuta v chromozomech každého jedince.*"

Tento citát v sobě nese známku optimalizace, vyhledávané adaptace (změny) musí být výhodné ve smyslu nějakého hodnotícího kritéria, v přírodě je zpravidla tímto hodnotícím kritériem schopnost reprodukce limitovaná množstvím potravy a dalších životně důležitých zdrojů v dané oblasti. Je zřejmé, že ne všichni jedinci v populaci svého druhu musí být stejně kvalitní, avšak předpokládá se, že kvalitu jedince (jeho schopnost přežití a reprodukce) lze vždy určit.

2.3 Algoritmus EVT

Evoluční výpočetní techniky nepracují s jedním kandidátem řešení daného problému, ale s množinou, tj. s populací jedinců, kandidátů $x_{t,\,i}$, ve tvaru $G(t) = \{x_{t,\,1}, x_{t,\,2}, ..., x_{t,\,N}\}$, kde t je vývojový čas a N označuje rozsah populace. Vývojový čas běží v diskrétních krocích a v tomto časovém pohledu se o posloupnosti populací hovoří jako o generacích. Všichni jedinci jsou v populaci implementováni pomocí stejných datových struktur, přičemž každý jedinec je ohodnocen svoji mírou kvality.



Obr. 3: Vývojový diagram algoritmu EVT.

Algoritmus EVT běží v cyklu, který simuluje vývojový čas, na počátku je vývojový čas nastaven na nulu a v kroku *inicializace* je vhodným způsobem vytvořena počáteční populace jedinců. Nejobvyklejším způsobem inicializace je vytvoření množiny náhodných jedinců, jiným způsobem je sestavení počáteční populace z jedinců vytvořených způsobem využívající apriorní znalosti o úloze.

Operace *vyhodnocení* v sobě skrývá výpočet kvality všech jedinců v populaci. Navíc se obvykle vyhledá nejlepší jedinec a vypočtou se i jisté statistické vlastnosti populace, jako průměrné ohodnocení jedinců, rozptyl a podobně.

Operace *selekce* simuluje proces přirozeného výběru, zápasu o přežití, vybírá tedy jedince z předchozí populace do nové, obvykle na základě jejího ohodnocení, označovaného jako fitness. Jedinci jsou v rámci selekce vybíráni zpravidla pravděpodobnostním mechanismem, přičemž lepší jedinci mají vyšší šanci být vybráni za prvky následující generace.

Selekce je silný nástroj pro zlepšování kvality populace, avšak jejím důsledkem je pouhé kopírování jedinců. Noví jedinci ve vývojovém procesu vznikají pouze na základě změnových operátorů, které využívají genetických akcí známých z živé přírody. Základními změnovými operátory jsou *křížení* a *mutace*.

Křížení generuje jednoho či více jedinců z dvou či více jedinců, rodičů, přičemž do operace křížení vstupují zpravidla dva rodiče. Celý proces křížení může být realizován tak, že nad každým selektovaným jedincem se provede náhodný pokus a příslušný jedinec se s určitou pravděpodobností stane kandidátem na páření. Jiný přístup za pár prohlásí dva po sobě jdoucí jedince tak, jak byli zařazeni do množiny selektovaných jedinců.

Druhým změnovým operátorem je mutace, která přichází na řadu po operaci křížení a vytváří nové jedince malou změnou původního jedince, mutace je tedy operací unární upravující konkrétního jedince, tento krok je ekvivalentem biologické mutace genů jedince.

Ne všichni selektovaní jedinci však musí projít těmito změnovými operacemi. Jedinci jsou pro tyto operace vybíráni s jistými pravděpodobnostmi. Pravděpodobnost křížení říká, jak často se bude křížení provádět. Pokud je pravděpodobnost nulová, potomci jsou přesnými kopiemi svých rodičů. Jestliže je pravděpodobnost 100 %, pak všichni potomci jsou vytvořeni křížením. Křížení předpokládá, že potomkům bude předána dobrá část genetického materiálu a spolu s novou částí povede k nalezení lepších řešení. Pravděpodobnost mutace říká, jak velká část genetického materiálu bude v rámci procesu mutace změněna. Mutace byla zavedena jako prevence před stagnací v lokálním extrému, neměla by být příliš intenzivní, protože by algoritmus mohl mít rysy náhodného prohledávání. Konkrétně doporučené hodnoty jsou pro pravděpodobnost křížení 80–95 % a pro mutaci 0,5–1 % (Beasley, 1992).

3 Gramatická evoluce

Jednou z nejnovějších metod spadajících do evolučních výpočetních technik je tzv. gramatická evoluce (Ryan a kol., 1998), (O'Neill a Ryan, 2001). Jde o metodu, která má mnohé společné jak s genetickými algoritmy, tak s genetickým programováním a může být chápána jako typ genetického programování založeného na gramatice. Oproti genetickému programování je gramatická evoluce obecnější v tom, že je navržena tak, aby byla použitelná pro hledání programů v libovolném jazyku, pokud použijeme Backus-Naurovu formu (BNF). S tím je i úzce spojena reprezentace jedinců, na rozdíl od stromů používaných v genetickém programování používá gramatická evoluce binární reprezentaci jedinců.

Tab. 1: Typické možnosti využití gramatické evoluce.

Úloha	Hledaný algoritmus	Vstup	Výstup
indukce posloupnosti	analytický předpis	index	element posloupnosti
symbolická regrese množiny dat	matematický výraz	nezávislé proměnné	závislé proměnné
optimální řízení	řídící strategie	stavové proměnné	akční veličiny
identifikace a predikce	matematický model systému	nezávislé proměnné	závislé proměnné
klasifikace	rozhodovací strom	hodnoty atributů	přiřazení do třídy
učení se cílenému individuálnímu chování	program popisující chování	vstup ze senzorů	akce organismu
odvození kolektivního chování	program popisující chování jedince	informace o vztahu jedince ke zbytku kolektivu	akce jedince v kolektivu

3.1 Backus-Naurova forma

Backus-Naurova forma (BNF) je metasyntaxe používaná k vyjádření bezkontextové gramatiky, která se používá pro popis formálních jazyků. Autoři John Backus a Peter Naur vytvořili bezkontextovou gramatiku, s jejíž pomocí definovali syntaxi programovacích jazyků využitím dvou typů pravidel: lexikálního a syntaktického. John Backus vytvořil tuto notaci, aby vyjádřil gramatiku programovacího jazyku ALGOL. BNF se často využívá k zápisu (notaci) gramatik počítačových programovacích jazyků, sad instrukcí a komunikačních protokolů, ale také jako notace zastupující části gramatik skutečných jazyků.

Gramatika BNF popisuje jazyk formou produkčních pravidel, ve kterých vstupují terminály, tj. atomické symboly, a neterminály, které jsou dále rozvinuty v jeden nebo více neterminálů a terminálů. Pravidla mají pevnou strukturu, kde na levé straně vystupují jednotlivé neterminály a na pravé straně je rozvoj příslušného neterminálu pomocí neterminálů a terminálů. Každý neterminál může mít více alternativních pravidel pro expandování. BNF je tedy sadou odvozených pravidel s následujícím zápisem:

```
<symbol> ::= <výraz se symboly>
```

kde `symbol` je neterminál a `výraz se symboly` sestává ze sekvence symbolů nebo sekvencí oddělených svislou čárou „|“, která indikuje možnost výběru. Celek představuje možnou náhradu za symbol vlevo. Symboly, které se na levé straně nikdy neobjeví, jsou terminály.

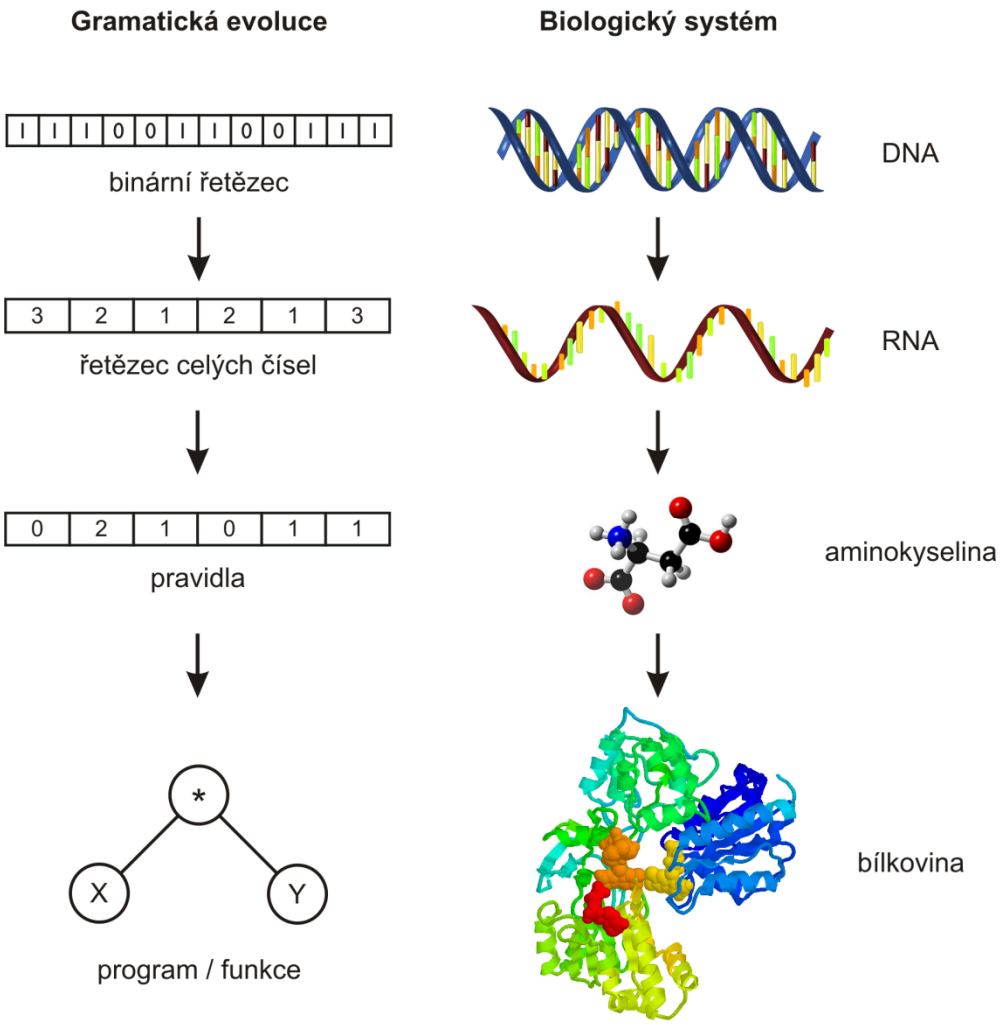
3.2 Biologický přístup

Genetický materiál DNA (deoxyribonukleová kyselina) je nositelem genetické informace všech organismů, s výjimkou nebuněčných, který ve své struktuře kóduje a buňkám zadává jejich program a tím předurčuje vývoj a vlastnosti celého organismu. DNA je biologická makromolekula, která je tvořena dvěma řetězci nukleotidů. Nukleotid vznikne spojením organické báze (A, T, G, C), pětiuhlíkatého cukru a fosfátu. Každá z 20 druhů aminokyselin, ze kterých se v buňkách syntetizují bílkoviny, je kódována třemi po sobě následujícími nukleotidy (bázemi), tzv. tripletem, který se označuje jako kodon.

K vytvoření bílkoviny ze sekvence nukleotidů v DNA se musí nejprve přepsat sekvence DNA do sekvence RNA, tento přepis se nazývá transkripce. Část molekul RNA pak po přesunu z jádra do cytoplazmy slouží jako šablona pro translaci, ta probíhá na ribozomech. Tímto způsobem na základě genetického kódu vznikají nové bílkoviny.

Aplikace produkčních pravidel na neterminály ještě nekompletního programu v gramatické evoluci je analogická roli aminokyselin, které se spojují dohromady, aby transformovali rostoucí molekuly bílkovin do konečné funkční podoby.

Výsledkem projevu genetického materiálu ve spojení s prostředím je fenotyp, čili jak organismus v daném znaku skutečně vypadá. V gramatické evoluci je fenotypem počítačový program, který je generován z genetického materiálu, genotypu, procesem považovaným za genotyp-fenotyp mapování.



Obr. 4: Srovnání systému gramatické evoluce a biologického genetického systému.

3.3 Mapování z genotypu do fenotypu

Programy v gramatické evoluci nejsou zapsány přímo ve stromové struktuře, jako je tomu u genetického programování, ale pomocí lineárního genomu, označovaného jako chromozom, což může být například posloupnost celých čísel. Pro převod z genotypu do fenotypu se používá Backus-Naurova forma formou definice gramatiky a operace modulo *n*, kde n je určeno daným počtem možností výběru. Gramatikou budeme nazývat čtveřici G = {N, T, P, S}, kde:

N je konečná množina neterminálních symbolů,
T je konečná množina terminálních symbolů, přičemž $N \cap T = 0$,
S je počáteční symbol, $S \in N$,
P je množina přepisovacích pravidel.

Uvažujme příklad, ve kterém se vyskytují operace {+, –, *, /, ^}, proměnné {X, Y} a celá čísla od 0 do 7. Dohromady tvoří množinu terminálů T = {+, -, *, /, ^, X, Y, 0, 1, 2, 3, 4, 5, 6, 7} a množinu neterminálů, která obsahuje symboly N = {expr, op, var, num}. Počáteční startovací symbol S = expr. Potom gramatika generující matematické výrazy může vypadat takto:

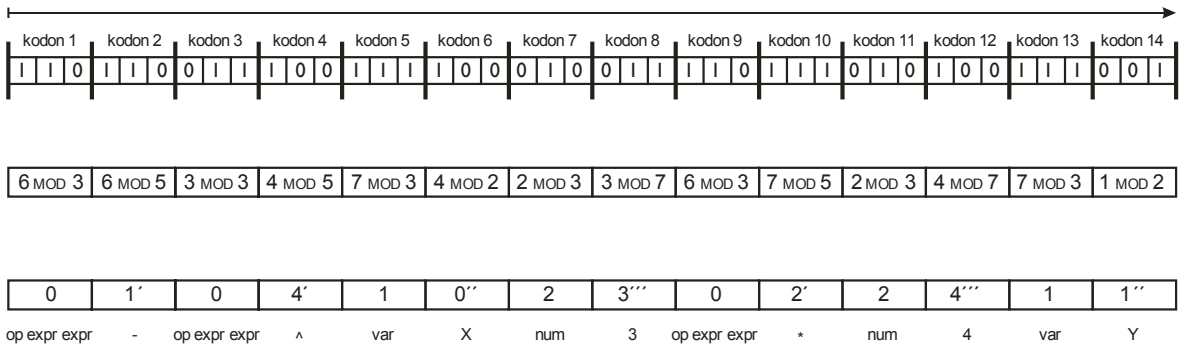
```
<expr>   :==  <op> <expr> <expr>          (0)
            |  <var>                      (1)
            |  <num>                      (2)
<op>     :==  +                           (0´)
            |  -                          (1´)
            |  *                          (2´)
            |  /                          (3´)
            |  ^                          (4´)
<var>    :==  X                           (0´´)
            |  Y                          (1´´)
<num>    :==  0|1|2|3|4|5|6|7       (0´´´,…, 7´´´)
```

Tato gramatika se nyní použije pro dekódování chromozomu, který má v gramatické evoluci takovou funkci, že reprezentuje posloupnost pravidel tak, jak budou postupně aplikována během generování programu. Chromozom je reprezentován pomocí binárního řetězce, který je formálně členěn na podřetězce, které se nazývají kodony, kódující celá čísla. Tyto kodony jsou postupně čteny od začátku chromozomu a na základě jejich hodnoty je použito odpovídající pravidlo pro rozvinutí aktuálního nejlevějšího neterminálu (používá generování do hloubky, a proto v derivačním stromu nejdříve expanduje neterminály s největší hloubkou). Vzhledem k tomu, že hodnota kodonu může být větší než počet pravidel použitelných pro jednotlivé neterminály, je číslo pravidla, které se pro rozvinutí neterminálu použije, stanoveno pomocí výrazu:

```
pravidlo = hodnota_kodonu MOD počet_pravidel_daného_neterminálu
```

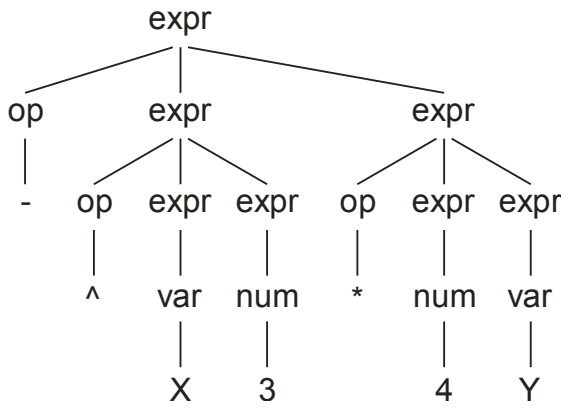
Čtení chromozomu pokračuje zleva doprava, dokud nedojde k situaci, kdy program je hotov, respektive všechny neterminály už byly přepsány na terminály, nebo program není hotov, respektive je třeba ještě rozvinout jeden nebo více neterminálů, ale už byl přečten poslední kodon chromozomu. V prvním případě, kdy v chromozomu zbývají nějaké nepoužité kodony, se jednoduše zbytek chromozomu ignorujeme. Ve druhém případě se můžeme pokusit dokončit odvození programu, abychom daného jedince mohli ohodnotit. Odvození můžeme dokončit například tak, že chybějící kodony budeme číst znovu od začátku chromozomu a aby nedošlo k nekonečnému nebo příliš dlouhému cyklení, používá se omezení na počet průchodů chromozomem.

Celý proces sestavení programu, pomocí gramatiky z předchozí stránky, si ukážeme na chromozomu na následujícím obrázku. Generování programu začíná rozvinutím počátečního startovacího symbolu `expr`, pro který máme tři pravidla. Protože první kodon chromozomu má hodnotu 6, bude se `expr` přepisovat pomocí pravidla (0) na `op expr expr`. Následuje rozvinutí symbolu `op`, z pěti možných pravidel vybereme pro druhý kodon s hodnotou 6 pravidlo (1´) a `op` přepíšeme na „-“. Analogicky pokračujeme rozvíjením dalších nejlevějších neterminálů, dokud není celý program hotov.

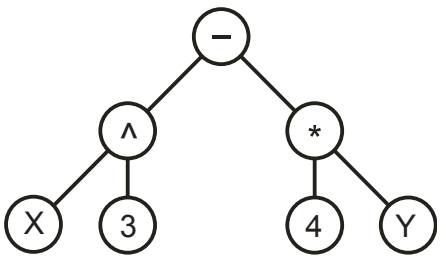


Obr. 5: Příklad chromozomu, hodnot kodonů a odpovídajících pravidel.

Tato pravidla se mohou zapsat ve formě stromu, který je často čitelnější. Následující obrázky zobrazují příklad derivačního stromu, který byl vytvořen postupnou derivací pravidel pomocí Backus-Naurovy formy a jeho ekvivalent již s finálními použitými hodnotami.



Obr. 6: Derivační strom pro chromozom z předchozího obrázku.



Obr. 7: Derivační strom z předchozího obrázku v čitelnější podobě s výslednou funkcí.

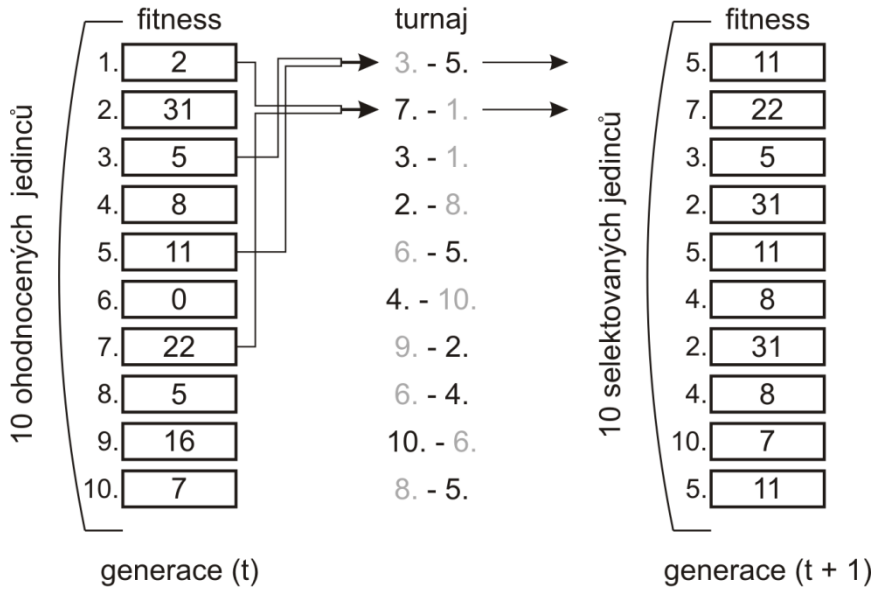
Zajímavým rysem popsané reprezentace je to, že explicitně odděluje prohledávaný prostor (binární řetězce) od prostoru řešení (programy). Díky mapování genotypu do fenotypu je gramatická evoluce blíže přirozenému konceptu genetického kódování, než kterýkoliv jiný evoluční algoritmus.

3.4 Selekce

Úkolem selekce je upřednostňovat kvalitnější jedince před horšími a jejím prostřednictvím vlastně dochází k výběru rodičů pro křížení, a proto je selekce jeden z klíčových okamžiků pro dobrý běh evolučního algoritmu. Podle Darwinovy evoluční teorie pouze nejlepší přežívají, tedy jako vhodní rodiče by se měli vybírat ti lepší. Na druhou stranu i horší jedinec může přinést vhodné geny pro budoucí vývoj. Gramatická evoluce používá selekci turnajovou.

3.4.1 Turnajová selekce

Turnajová selekce zcela opouští náhodný či polonáhodný výběr s pravděpodobnostmi souvisejícími s kvalitou jedinců a nahrazuje ho nelítostnou soutěží jedinců zápasících o přežití. Algoritmicky se postupuje tak, že z původní populace se zcela náhodně vyberou skupinky jedinců, mezi nimiž se uspořádá turnaj, přičemž velikost skupinky se označuje jako sílá selekce. Vítěz, tj. nejlépe ohodnocený jedinec ve skupince, se stane prvkem selektované populace. Tento postup se opakuje tak dlouho, dokud selektovaná populace nemá potřebný počet jedinců.



Obr. 8: Turnajová selekce o síle 2 znázorněná přímo na populaci jedinců.

3.5 Křížení

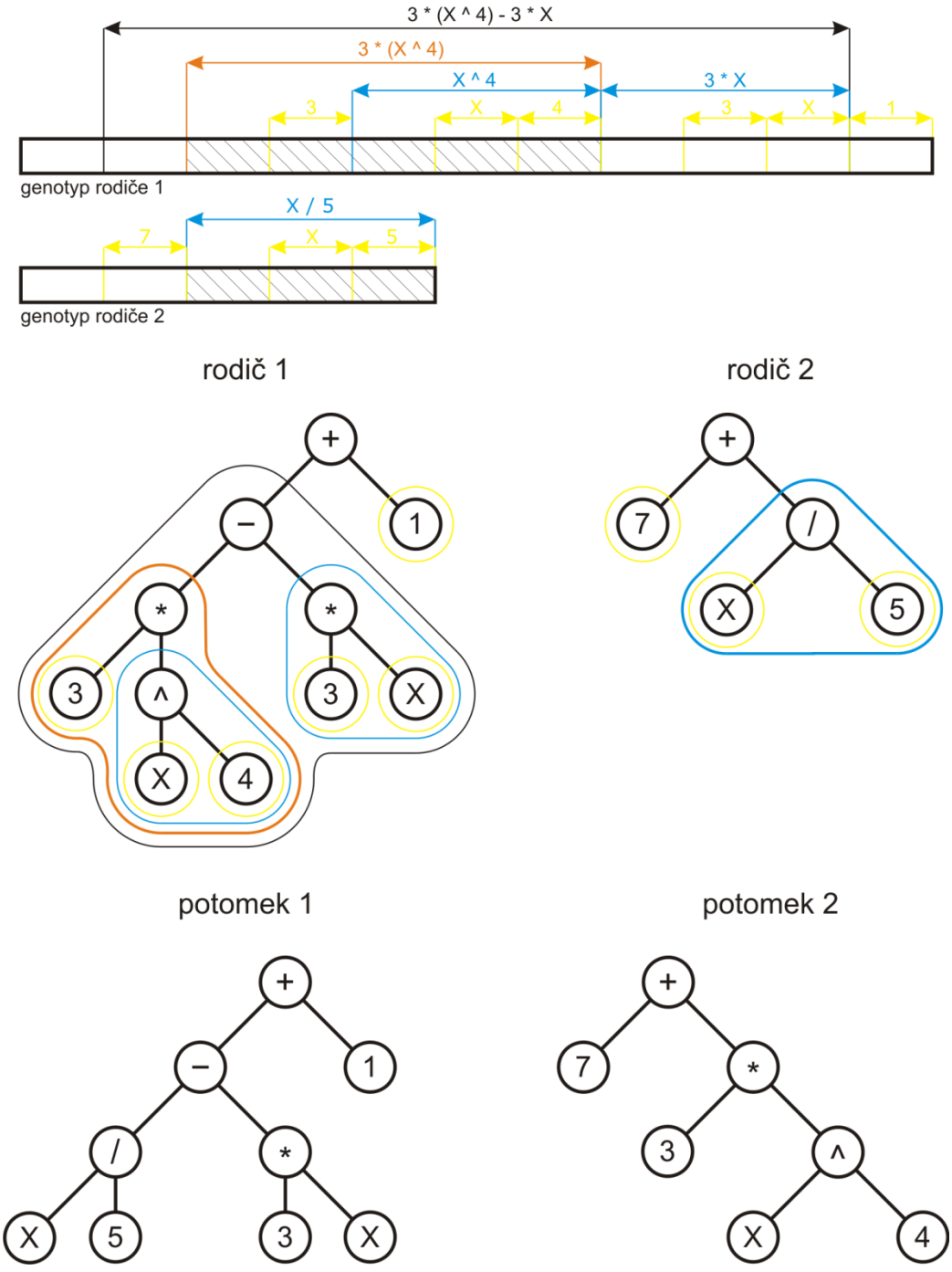
Gramatická evoluce, jako základní rekombinační operátor, používá křížení. Do této operace vstupují dva jedinci a výsledkem je jeden nebo dva potomci. Jeden z možných postupů, který se v gramatické evoluci používá a který je ekvivalentem křížení v genetickém programování je, že v každém z rodičovských chromozomů se náhodně vybere po jednom podstromu, které se vzájemně zamění. Tento typ křížení budeme označovat jako křížení strukturální. Druhým možným postupem je, že rodičovské řetězce si vzájemně prohodí sekvence bitů za náhodně zvoleným bodem křížení, tento princip je ekvivalentem křížení v genetických algoritmech a budeme ho označovat jako křížení v náhodném bodě.

3.5.1 Křížení strukturální

Při použití gramatiky závisí výsledný fenotyp jednak na hodnotě genu a jednak na jeho kontextu. Při křížení v náhodném bodě obvykle dochází ke změně kontextu a tedy i ke změně fenotypu. Křížení je tedy destruktivní, neboť nové části chromozomu kódují jiný fenotyp než v původním jedinci. Pokud se však v chromozomu označí úseky, které kódují podstromy fenotypu, potom jedinci, kteří vznikli výměnou těchto úseků, nejsou v rámci kontextu změněny, a proto křížení není destruktivní.

Tento způsob křížení pracuje podobně jako přímé křížení podstromů funkcí, nicméně stále se vyměňují pouze části chromozomu, tedy genotyp a fenotyp zůstává oddělený. Po křížení je tedy vzniklý jedinec kombinací podstromů obou rodičů, ale známe jeho genotyp, který tuto strukturu může kdykoliv vytvořit.

Následující obrázek ukazuje příklad křížení dvou chromozomů. V každém z rodičovských chromozomů je náhodným mechanismem vybrán úsek chromozomu, který kóduje určitý podstrom fenotypu, v derivačních stromech rodičů jsou vyznačeny všechny možné podstromy, které se v rámci procesu křížení mohou zaměnit, přičemž vybraný podstrom je v každém rodiči vyznačen tučně a v příslušných chromozomech vyšrafováním.

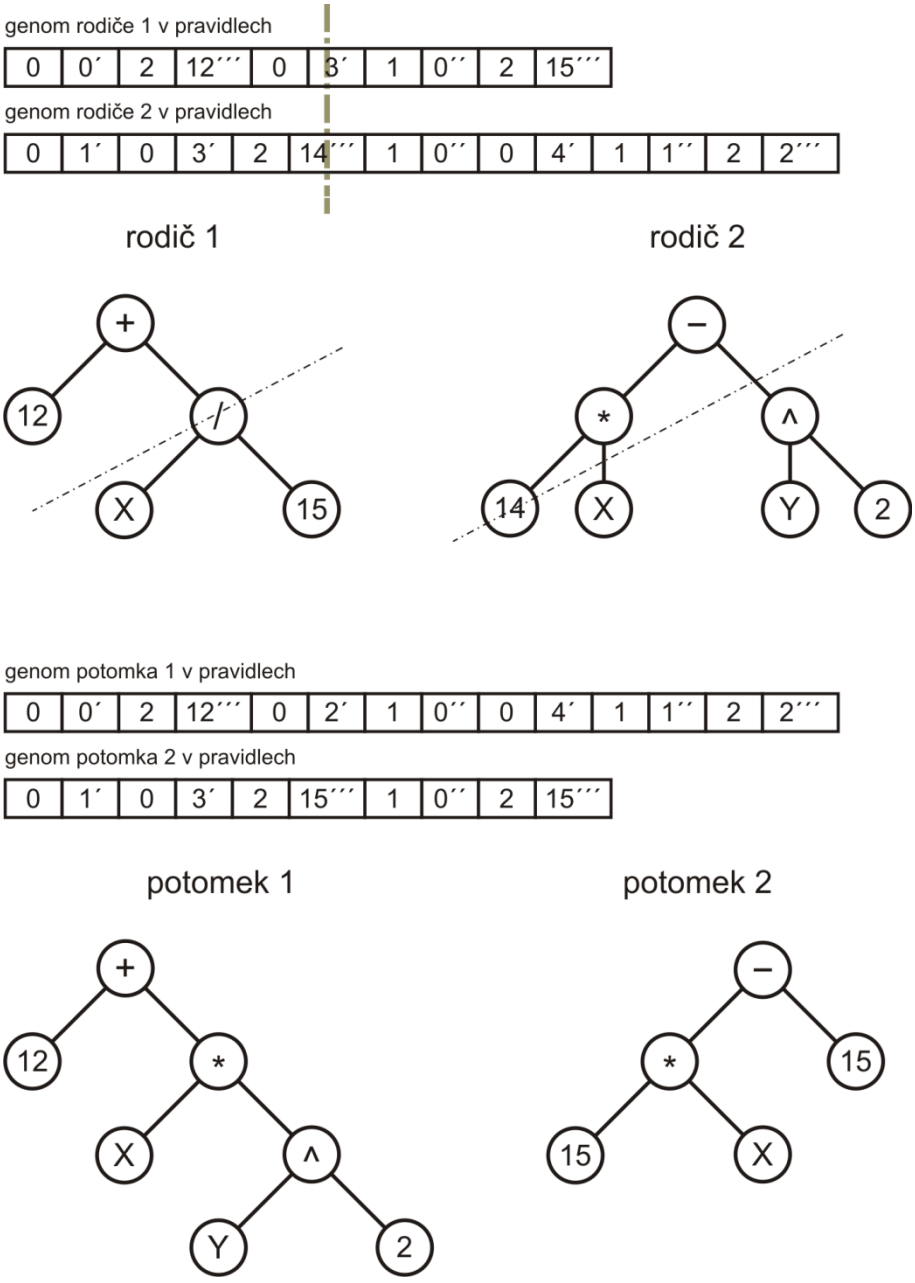


Obr. 9: Křížení strukturální znázorněné přímo na programech.

3.5.2 Křížení v náhodném bodě

Druhým možným postupem křížení, které se v gramatické evoluci používá, je tzv. křížení v náhodném bodě. Podle efektu, který má tento operátor na křížené stromové struktury rodičovských jedinců, se také nazývá vlnové křížení. Křížení funguje tak, že si rodičovské řetězce vzájemně prohodí sekvence bitů za náhodně zvoleným bodem křížení.

Následující obrázek znázorňuje příklad křížení dvou chromozomů v místě vyznačeném dělicí čarou, pro jednoduchost jsou zapsány jako posloupnost pravidel a pro mapování z genotypu do fenotypu používá gramatiku z kapitoly 3.3, ale s rozdílem, že terminály skupiny `num` jsou kódovaný více bity než ostatní terminály a neterminály. V místě křížení se chromozomy rozdělí na dvě části, první reprezentuje kostru nového jedince a druhá větve, které byly z rodičovského stromu oříznuty. Křížení se dokončí tak, že kostra nového jedince bude doplněna uzly a podstromy, které budou generovány použitím kodonů ze zbývající části druhého rodiče.



Obr. 10: Křížení v náhodném bodě znázorněné přímo na programech.

3.6 Mutace

Druhým rekombinačním operátorem, který gramatická evoluce používá, je mutace, která vytváří nové jedince malou změnou původního jedince, je to tedy operace unární upravující konkrétního jedince. Jeden z možných postupů je mutace strukturální, která cíleně ovlivňuje strukturální i nestrukturální geny, jiným postupem je mutace náhodných bitů, která provádí inverzi náhodně zvolených bitů v chromozomu.

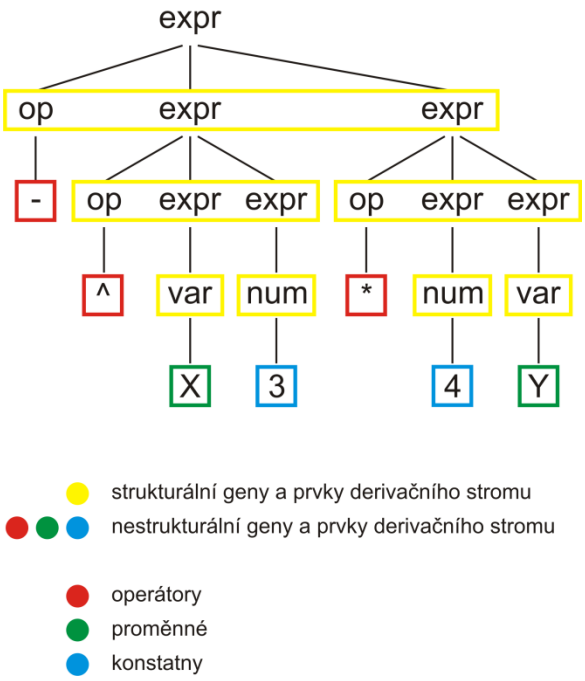
3.6.1 Mutace strukturální

Mutaci lze rozdělit na mutaci strukturálních a nestrukturálních genů. Mutace jednoho strukturálního genu ovlivní ostatní geny, její míra by proto měla být malá nebo dokonce i nulová, protože nové struktury lze vytvořit progresivním křížením. Rozlišení mutací umožňuje nastavení poměrně vysoké mutace nestrukturálních genů a tedy i rychlejší prohledávání. Přitom nehrozí poškození již nalezené struktury.

Následující obrázek znázorňuje příklad lineárního genomu, na němž jsou barevně odlišeny jednotlivé typy genů. Strukturální geny jsou označeny žlutou barvou a nestrukturální geny barvami červenou, zelenou a azurovou, přičemž tyto tři barvy vzájemně odlišují typy terminálních symbolů. Každý typ terminálních symbolů, mezi které patří operátory, proměnné a konstanty, mohou mutovat s jinou mírou pravděpodobnosti.

Obr. 11: Znázornění jednotlivých typů genů na genomu.

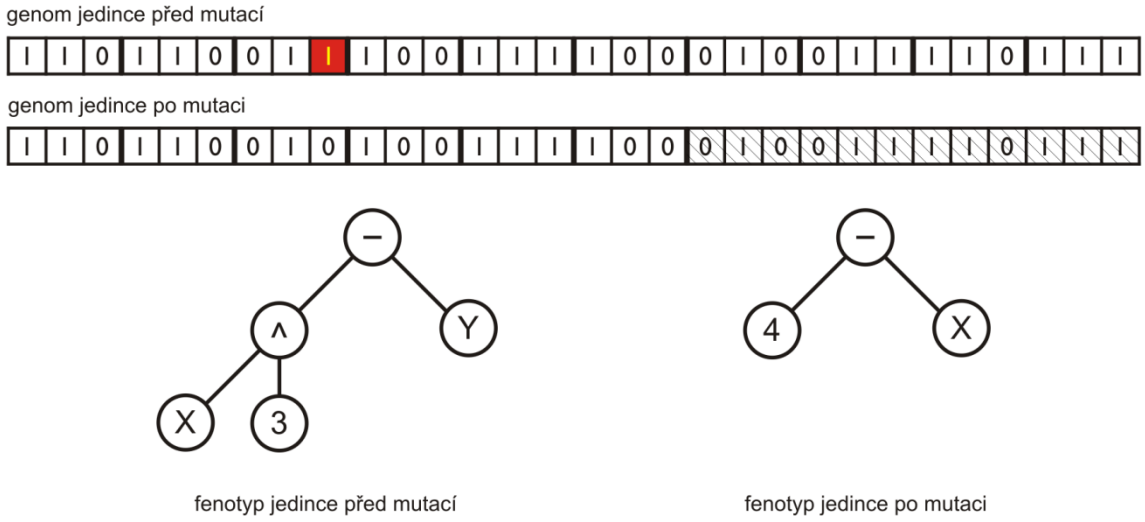
Následující obrázek znázorňuje jednotlivé typy genů v derivačním stromu programu pro genom z předchozího obrázku, přičemž každý obdélník v derivačním stromu odpovídá právě jednomu genu, respektive typu genu, v genomu. Pravidla programu jsou tedy kódovány pomocí čtrnácti kodonů, z nichž sedm je strukturálních a sedm nestrukturálních.



Obr. 12: Znázornění jednotlivých typů genů v derivačním stromu programu.

3.6.2 Mutace náhodných bitů

Tato mutace má velmi jednoduchou podobu, algoritmicky probíhá tak, že z celkového počtu bitů v celé populaci, dle faktoru mutace, se určí počet bitů, které budou invertovány. Nejprve tedy proběhne volba jedince a následně pozice bitu, který bude invertován, tento postup se bude opakovat tak dlouho, dokud nebude invertován požadovaný počet bitů.

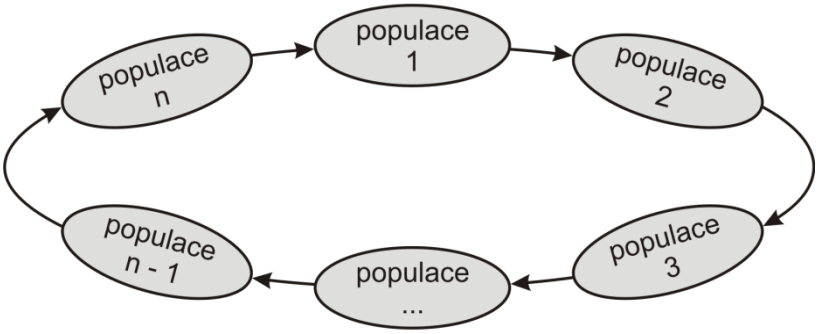


Obr. 13: Mutace náhodného bitu znázorněná přímo na genomu.

3.7 Paralelní gramatická evoluce

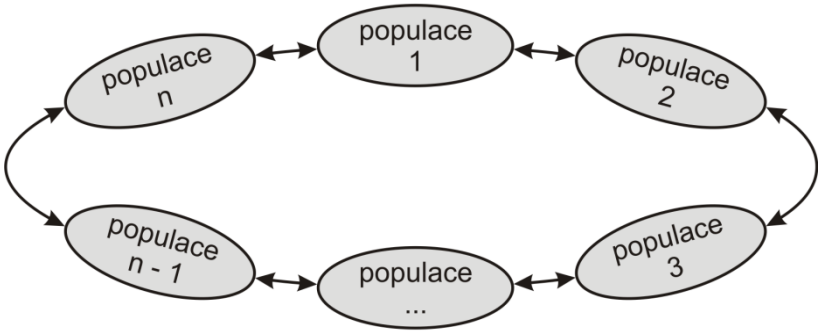
Paralelní evoluční algoritmy jsou výkonné stochastické prohledávací strategie inspirované přírodou, dovolují řešit větší a složitější problémy, přičemž často vedou rychleji k řešení a současně konvergují k lepším výsledkům. Používají se tři modely paralelní gramatické evoluce, mezi ně patří tzv. farmářský model, migrační model a difusní model. Paralelní gramatická evoluce pracuje s nezávislými podmnožinami populace, v nichž probíhá evoluce částečně izolovaně. Pojem paralelní nebo sekvenční se vztahuje k populačním strukturám, nikoli k hardwaru, na kterém jsou evoluční algoritmy implementovány.

Realizovaná aplikace implementuje určitý typ migračního modelu, kde migrací rozumíme nakopírování chromozomů z jedné populace do jiné, přičemž kopírované chromozomy jsou v populacích zachovávány. Existuje celá řada migračních struktur, aplikace implementuje tři základní struktury migrační paralelní gramatické evoluce: jednosměrnou kruhovou, obousměrnou kruhovou a vzájemnou strukturu. Všechny tyto tři struktury lze kombinovat se čtvrtým typem, který využívá tzv. master populace. Princip migrace, tedy kopírování chromozomů z jedné populace do jiné, je pro všechny struktury znázorněn na následujících obrázcích.

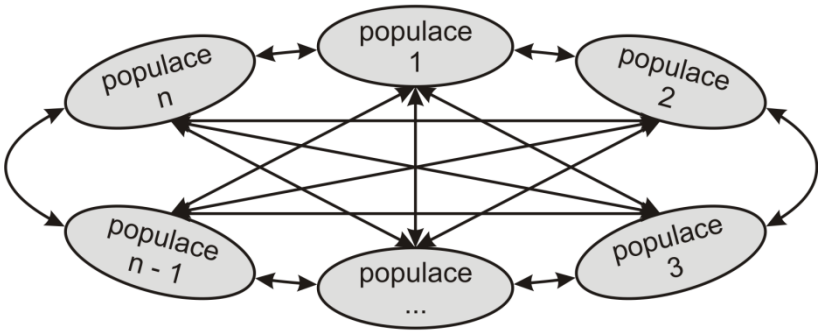


Obr. 14: Schematické znázornění jednosměrné kruhové paralelní struktury.

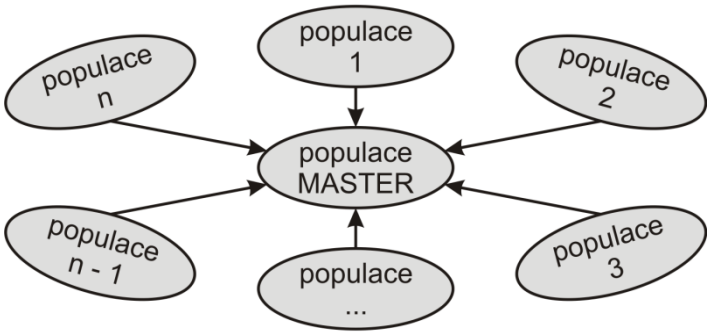
Algoritmicky se postupuje tak, že z populace, z které mají být chromozomy kopírovány, se vybere určitý počet chromozomů, přičemž výběr probíhá pomocí turnajové selekce, na základě ohodnocení a síly migrace, která je ekvivalentem síly selekce, a jejich počet je dán počtem migrantů. V populaci, do které mají být tyto chromozomy nakopírovány, se pomocí turnajové selekce, na základě ohodnocení a síly migrace, vybere příslušný počet nejhůře ohodnocených chromozomů, které jsou nahrazeny chromozomy z populace, z které jsou kopírovány. Tento proces se spouští periodicky, jednou za určitou dobu, která se nazývá frekvence migrace.



Obr. 15: Schematické znázornění obousměrné paralelní struktury.



Obr. 16: Schematické znázornění vzájemné paralelní struktury.



Obr. 17: Schematické znázornění paralelní struktury s master populací.

3.8 Elitismus

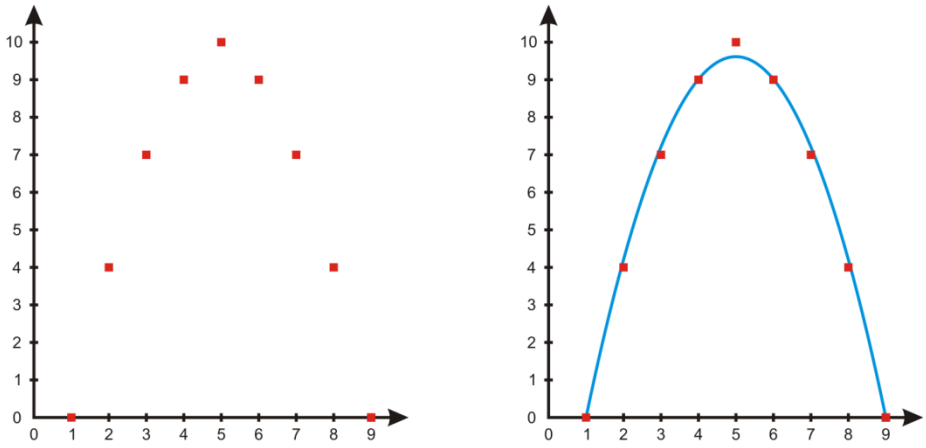
Kromě selekce se také zavádí elitismus. V každé populaci se vybere nejprve nejlepší jedinec (s nejlepší hodnotou účelové funkce), který se umístí do nové populace, pak se teprve provádí mechanismus selekce. Elitismus výrazně zlepšuje průběh evolučních algoritmů, protože zamezuje ztrátě nejlepších řešení.

4 Aproximace funkcí

Jedním ze základních úkolů numerických metod matematické analýzy je studium aproximací funkcí. Při numerickém řešení úloh matematické analýzy totiž často nahrazujeme danou funkci *f*, vystupující v řešené matematické úloze, jinou funkcí φ, která v nějakém smyslu napodobuje funkci *f* a snadno se přitom matematicky zpracovává či modeluje na počítači. Tuto φ funkci nazýváme aproximací funkce *f*.

V matematice tedy aproximovat funkce $f(x)$ znamená nahradit ji funkcí $\varphi(x)$, která je k $f(x)$ v jistém smyslu blízká, a proto píšeme $\varphi(x) \approx f(x)$. Cílem aproximace je tedy proložit data průběhem, který je generován nějakým vhodně navrženým modelem. Mezi základní numerická řešení matematické analýzy patří aproximace interpolací a aproximace metodou nejmenších čtverců.

Interpolace je taková aproximace, při níž $\varphi(x)$ nabývá v zadaných bodech x_i předepsaných hodnot $y_i = f(x_i)$. Metoda nejmenších čtverců je taková aproximace, při níž $\varphi(x)$ „prokládáme“ mezi zadanými body $[x_i, y_i]$ tak, aby „vzdálenost“ funkcí f a φ byla v jistém smyslu minimální. Je přitom charakteristické, že funkce φ body $[x_i, y_i]$ neprochází.



Obr. 18: Aproximace neznámé funkce, naměřená data proložená vhodnou křivkou.

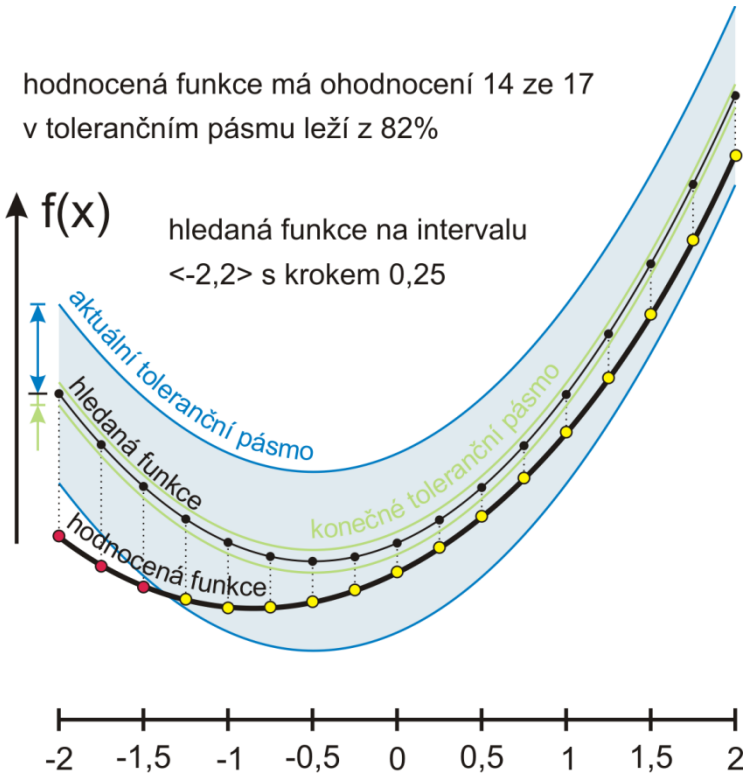
Alternativní metodou k metodám klasickým je použití evolučních algoritmů, případně metod symbolické regrese. V tomto směru již byla testována celá řada algoritmů, moje práce se zabývá testováním životaschopnosti gramatické evoluce s různými přístupy ohodnocení kvality řešení, dále porovnáním gramatické evoluce s paralelní gramatickou evolucí, tedy vliv paralelních struktur na výsledky konečného řešení.

4.1 Účelová funkce

Úkolem účelové funkce je ohodnocování vzniklých výrazů a v kontextu s genetickými postupy použití tohoto ohodnocení jako měřítka kvality (fitness) každého jedince. Pro ohodnocování aproximace funkcí jsem popsal a v praktické části implementoval tři typy účelových funkcí, první pracuje s dynamickým tolerančním pásmem, která je také označována jako epsilonová trubice [10], druhá pracuje se součtem kvadratických odchylek a třetí se součtem absolutních odchylek.

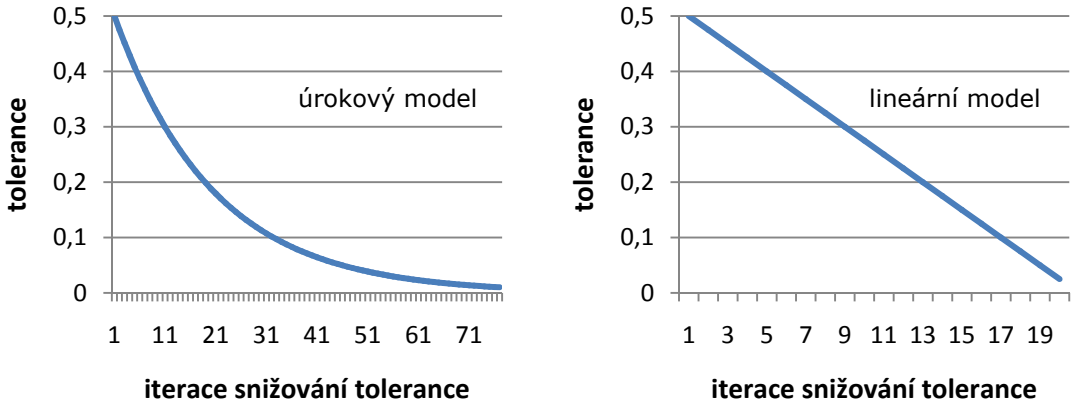
4.1.1 Dynamické toleranční pásmo

Okolo hledané funkce, definované funkčními hodnotami, je toleranční pásmo dané šířky. Vhodnost jedince je pak počet vyhovujících bodů ze všech kontrolních bodů. Vyhovující bod je takový, ve kterém hodnota generované funkce leží v tolerančním pásmu. Tento způsob hodnocení vytváří silný selekční tlak, algoritmus proto obvykle velmi rychle najde přibližné řešení.



Obr. 19: Princip účelové funkce znázorněný na hodnocené funkci.

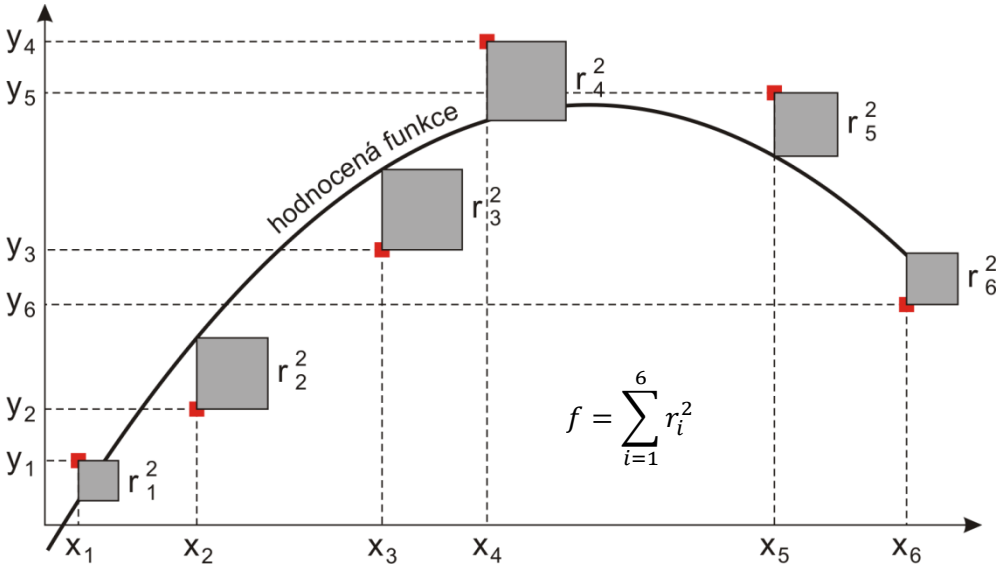
Algoritmicky se postupuje tak, že zpočátku evolučního procesu se nastaví počáteční šířka tolerančního pásma na dvojnásobek hodnoty definované uživatelem. Dojde-li ke splnění podmínky snížení tolerance, která říká, že v tolerančním pásmu musí ležet alespoň určitá část bodů, potom dojde ke snížení aktuální velikosti tolerance. Tolerance může být snižována buď tzv. úrokovým modelem, kdy se hodnota aktuální tolerance vždy o část sníží, nebo lineárním modelem, kdy se hodnota aktuální tolerance vždy snižuje o konstantní hodnotu.



Obr. 20: Dva různé přístupy snižování tolerančního pásma.

4.1.2 Součet kvadratických odchylek

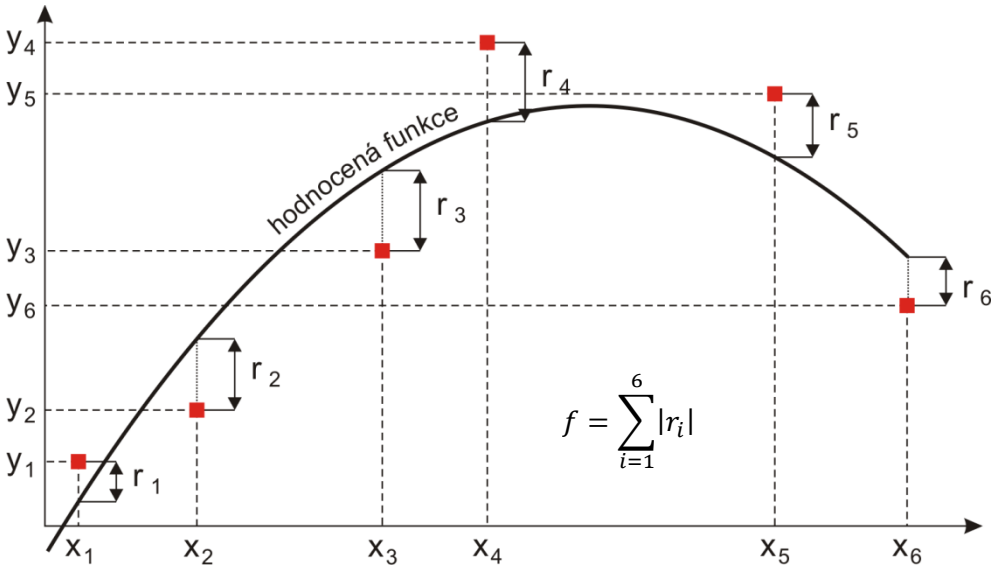
V každém bodě hodnocené funkce, definované funkčními hodnotami, se vypočítá druhá mocnina odchylky této funkční hodnoty od hledané funkční hodnoty, tato odchylka se nazývá reziduum. Potom hodnotou účelové funkce je součet čtverců, druhých mocnin, reziduí, respektive součet kvadratických odchylek.



Obr. 21: Princip účelové funkce znázorněný na hodnocené funkci.

4.1.3 Součet absolutních odchylek

V každém bodě hodnocené funkce, definované funkčními hodnotami, se vypočítá absolutní odchylka této funkční hodnoty od hledané funkční hodnoty, tato odchylka se nazývá reziduum. Potom hodnotou účelové funkce je součet reziduí, respektive součet absolutních odchylek.



Obr. 22: Princip účelové funkce znázorněný na hodnocené funkci.

5 Syntéza logických obvodů

Gramatická evoluce není vhodná pouze jako nástroj pro nejrůznější analýzy matematických funkcí, ale také, a to zejména pro řešení nejrůznějších technologických problémů, především pak v optimalizaci výrobních procesů, návrh elektronických obvodů a podobně. Logické obvody rozdělujeme na dvě základní skupiny, kombinační obvody a sekvenční obvody, přičemž realizovaná aplikace implementuje syntézu kombinačních i sekvenčních logických obvodů.

5.1 Kombinační obvody

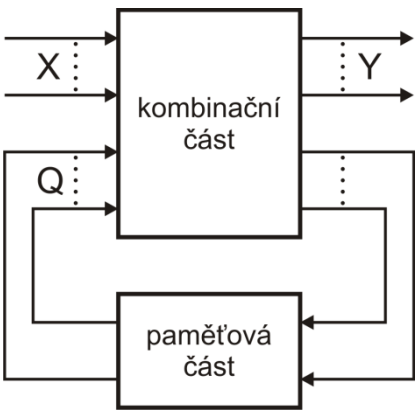
U kombinačních obvodů jsou funkční hodnoty jednoznačně určeny kombinacemi hodnot vstupných proměnných a nezávisí na jejich předchozích hodnotách. Závislost výstupních funkčních hodnot na hodnotách vstupních proměnných se prostřednictvím aplikace zadává pravdivostní tabulkou, přičemž pro jejich realizaci používá základní logické prvky (AND, NAND, OR,…).

Tab. 2: Příklad pravdivostní tabulky pro booleovskou paritní funkci.

X1	X2	X3	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

5.2 Sekvenční obvody

U sekvenčních obvodů jsou funkční hodnoty určeny nejen kombinacemi hodnot vstupních proměnných, ale také jejich časově předcházejícími kombinacemi hodnot. Tyto předcházející hodnoty jsou v sekvenčních obvodech uchovány v paměťové části obvodu. Realizovanou aplikací lze navrhovat asynchronní sekvenční obvody, to znamená, že každá změna vstupních a výstupních proměnných není řízena synchronizačními impulsy, které by zajišťovaly stejné okamžiky změn všech proměnných.



Obr. 23: Sekvenční obvod, kde X, Y a Q označují vstupní, výstupní a stavové proměnné.

5.3 Účelová funkce

Jako účelová funkce byla zvolena funkce, která provádí výpočet Hammingovy vzdálenosti tak, že namísto „pravda“ a „nepravda“ se pro výpočet berou hodnoty 1 a 0. Vstupem účelové funkce je aktuálně syntetizovaná funkce, ze které byl na základě vstupů vygenerován výstup a podle účelové funkce byl spočítán rozdíl mezi výstupem aktuálně syntetizované funkce a výstupem vygenerované funkce.

$$f = n - \sum_{i=1}^{n} |T_i - P_i|$$

T_i … výstup pravdivostní tabulky
P_i … odezva syntetizovaného programu
n … počet všech možných kombinací

Následující tabulka, na příkladu, ukazuje princip výpočtu hodnoty účelové funkce. Pravdivostní tabulka má tři vstupní proměnné X1, X2 a X3, jednu výstupní proměnnou Y a výstup vygenerované funkce značí Y*. Teoreticky maximální možná hodnota účelové funkce je 8 a minimální 0, přičemž maximální hodnota reprezentuje program, který plně splňuje pravdivostní tabulku. Hodnota účelové funkce pro vygenerovanou funkci Y* je 6.

Tab. 3: Pravdivostní tabulka, Y a Y je výstup syntetizovaného a vygenerovaného programu.*

X1	X2	X3	Y	Y*
0	0	0	1	1
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

6 Jazyk Java

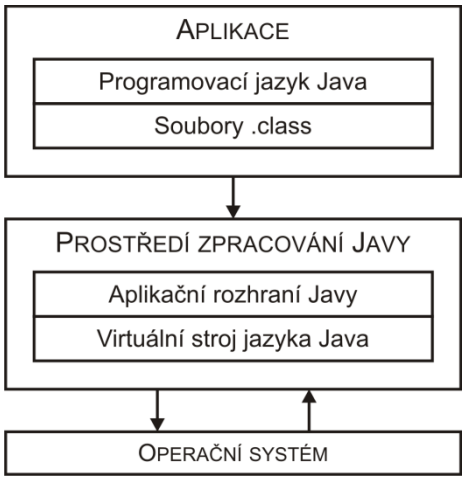
Podle společnosti Sun je Java „jednoduché, robustní, objektově orientované, na platformě nezávislé, více vláknové, dynamické univerzální programovací prostředí“. Java své uplatnění nachází v širokém spektru odběratelů a lze ji najít jak v jednoduchých aplikacích pracovní plochy, tak v ohromných chlazených sálových počítačích.

6.1 Architektura jazyka Java

Programovací jazyk Java je jen jednou z mnoha součástí prostředí Javy. Je to podkladová architektura, která Javě poskytuje mnoho výhod, například nezávislost na použité platformě. Celá architektura Javy je ve skutečnosti kombinací následujících čtyř součástí:

- programovací jazyk Java,
- formát souboru `.class`,
- aplikační programové rozhraní Javy,
- virtuální stroj jazyka Java.

Je-li vyvíjena aplikace v Javě, je psán kód v programovacím jazyku Java. Ten je posléze překládán do souboru typu `.class`. Výsledné soubory `.class` jsou spouštěny v prostředí virtuálního stroje jazyka Java. Kombinace virtuálního stroje jazyka Java s třídami výkonného jádra jazyka Javy je známa jako prostředí pro zpracování jazyka Java.



Obr. 24: Prostředí pro zpracování jazyka Java.

6.2 Virtuální stroj jazyka Java

Je to software, který vystupuje jako hardwarové zařízení a který umožňuje spouštění apletů v Jazyce Java, respektive převádí jejich bajtový kód do nativních instrukcí pro hostitelský počítač. Klíčovou předností jazyka Java je tedy skutečnost, že překladač jazyka Java generuje optimalizovanou sadu instrukcí nazývanou bajtovým kódem, který je posloupností formátovaných bajtů. Program obsahující takový kód je interpretován systémem nazývaným virtuální stroj jazyka Java, díky tomu lze program generovaný na jedné platformě používat na jakékoli jiné platformě, na niž je instalován právě virtuální stroj jazyka Java.

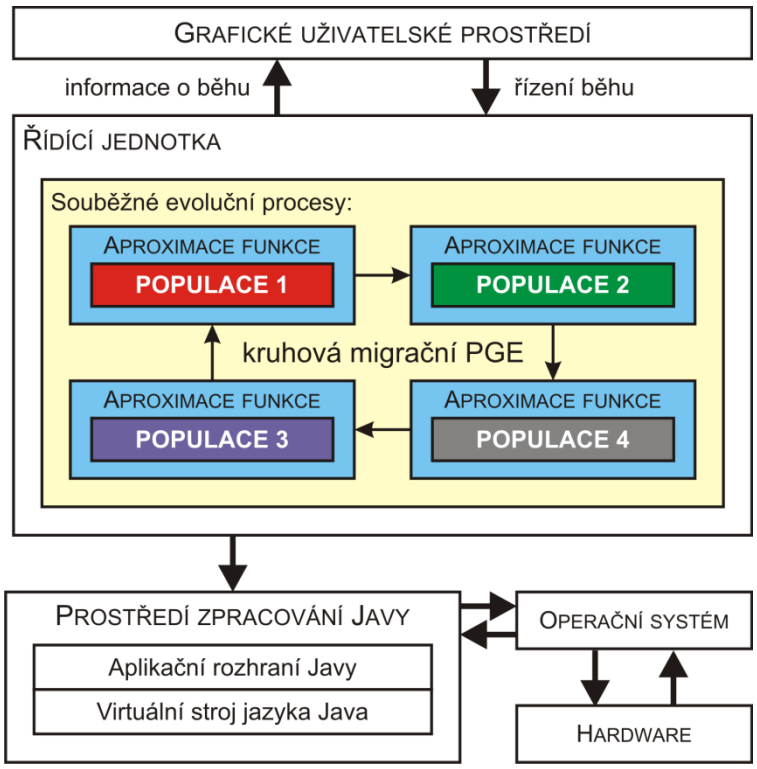
Tento přístup má však i své nevýhody, proti programovacím jazykům, které provádějí tzv. statickou kompilaci (např. C++), je start programů psaných v Javě pomalejší, protože prostředí musí program nejprve přeložit a potom teprve spustit. Je však možnost využít mechanismů JIT a HotSpot, kdy se často prováděné nebo neefektivní části kódu přeloží do strojového kódu a program se zrychlí. Na zrychlení se také podílí nové přístupy ke správě paměti

7 Implementace v jazyce Java

V této kapitole je podrobněji popsán způsob implementace, třídy a jejich metody. Celá aplikace, respektive třídy včetně grafického uživatelského rozhraní, je vytvořena ve vývojovém prostředí NetBeans IDE, což je open source projekt s rozsáhlou uživatelskou základnou a komunitou vývojářů, pomocí kterého programátoři mohou psát, překládat, ladit a šířit své programy. Cílem diplomové práce bylo implementovat třídy pro jednotlivé řešené úlohy, pomocí kterých lze vytvářet populace jedinců, dále třídy, které budou s těmito populacemi v rámci jednoho evolučního procesu pracovat a také třídy, které budou všechny evoluční procesy spouštět, vyhodnocovat a řídit v čase.

Java umožňuje paralelní běh dvou či více částí programu, přičemž každá paralelně běžící část programu se v Javě nazývá vlákno. Realizovaná aplikace umožňuje paralelní běh více vláken, kde vláknem je jeden evoluční proces na jedné příslušné populaci, díky čemuž aplikace dokáže těžit z výhod moderních víceprocesorových počítačů.

Následující obrázek zobrazuje nástin paralelní aplikační struktury včetně prostředí pro zpracování jazyka Java (Java Runtime Environment – JRE), kde souběžné evoluční procesy znázorňují vlákna. Řídící jednotka zajišťuje pozastavování těchto procesů, aby bylo možné vyhodnotit a uživateli prostřednictvím grafického uživatelského rozhraní zobrazit dosud nejlepší nalezené řešení, eventuelně jiné statistické a informační údaje o populacích. Další činnosti řídící jednotky je periodicky spouštět algoritmus paralelní gramatické evoluce (PGE).



Obr. 25: Nástin paralelní aplikační struktury včetně JRE.

Jak již bylo poznamenáno, v této kapitole je podrobněji popsán způsob implementace, tedy podkapitoly rozebírají smysl jednotlivých tříd, význam některých jejich důležitých položek a metod. Dále jsem zde uvedl jen třídy určené pro úlohu aproximace funkcí, a to proto, že třídy pro úlohu syntézy logických obvodů jsou k nim v mnoha směrech analogické. Ještě je třeba poznamenat, že jsem zde neuvedl žádný popis metod grafického uživatelského rozhraní, jednak z důvodu jejich rozsáhlosti a také proto, že s gramatickou evolucí přímo nesouvisí.

7.1 Třída Population

Třída `Population` je předchůdcem tříd `PopulationAF` a `PopulationSLC`, to jsou třídy pro reprezentaci populací pro úlohy aproximace funkcí a syntézy logických obvodů. Třída obsahuje společné položky a metody pro obě tyto úlohy. Základní je položka s názvem `chromosome`, je to pole objektů datového typu `BitVector` z knihovny `Colt`, která vznikla v laboratořích jaderného institutu CERN ve Švýcarsku, pomocí tohoto datového typu lze reprezentovat uspořádanou množinu bitů dané velikosti, tedy jednoho jedince, potom pole s názvem `chromosome` reprezentuje celou populaci jedinců. Mezi další důležité položky patří `rules`, která reprezentuje hodnoty pravidel jednoho přečteného jedince, při mapování z genotypu do fenotypu jsou hodnoty pravidel přidávány do této položky, přičemž tento přístup umožňuje vícenásobné provádění programu bez nutnosti opakovaného mapování z genotypu do fenotypu. Poslední významnou položkou je `subTreesBlocks`, je to pole polí proměnné velikosti, do kterého se pro každého jedince, během mapování z genotypu do fenotypu, ukládají intervaly na množině bitů jedince, které mají možnost se v rámci strukturálního křížení vyměnit.

metoda:	`Population`	
hlavička:	`public Population()`	
parametry:	-	
popis:	Konstruktor třídy, inicializuje datové prvky objektu a volá metodu pro vytvoření počáteční populace náhodných jedinců.	
metoda:	`generateRandomPopulation`	
hlavička:	`private void generateRandomPopulation(int populationSize, int minBitCount, int maxBitCount)`	
parametry:	`populationSize`	počet jedinců v populaci
	`minBitCount`	minimální počet bitů jedince
	`maxBitCount`	maximální počet bitů jedince
popis:	Metoda inicializuje položku `chromosome`, vytváří tedy počáteční populaci náhodných jedinců, jejichž velikost je v daném intervalu náhodná.	
metoda:	`codonToByte`	
hlavička:	`protected byte codonToByte(byte codonSize, byte[] artificialArray) throws IndexOutOfBoundsException`	
parametry:	`codonSize`	počet bitů, které kódují právě čtený kodon
	`artificialArray`	vypočítané mocniny základu binární soustavy
popis:	Metoda se pokusí přečíst hodnotu následujícího kodonu aktuálně čteného jedince, pokud už byl ale přečten poslední bit chromosomu, pokračuje se jeho čtením znovu od začátku nebo se vyhodí výjimka, která je ošetřena na jiném místě.	
metoda:	`tournamentSelectionHigherIsBetter`	
hlavička:	`protected void tournamentSelectionHigherIsBetter (float[] fitness)`	
parametry:	`fitness`	pole ohodnocení všech jedinců v populaci
popis:	Metoda realizující turnajovou selekci, respektive na základě ohodnocení jedinců se vybírají prvky selektované populace, přičemž čím je hodnota prvku pole vyšší, tím je ohodnocení lepší. Jakmile má selektovaná populace stejný počet prvků jako populace původní, je původní populace nahrazena populací selektovanou.	
metoda:	`tournamentSelectionLowerIsBetter`	
hlavička:	`protected void tournamentSelectionLowerIsBetter(float[] fitness)`	
parametry:	`fitness`	pole ohodnocení všech jedinců v populaci
popis:	Metoda realizující turnajovou selekci, respektive na základě ohodnocení jedinců se vybírají prvky selektované populace, přičemž čím je hodnota prvku pole nižší, tím je ohodnocení lepší. Jakmile má selektovaná populace stejný počet prvků jako populace původní, je původní populace nahrazena populací selektovanou.	

metoda:	`crossoverStructural`
hlavička:	`protected void crossoverStructural()`
parametry:	-
popis:	Metoda realizující strukturální křížení, v cyklu jsou pro křížení vybrání vždy dva po sobě jdoucí jedinci, přičemž výběr intervalu z `subTreesBlocks`, tedy množiny po sobě jdoucích bitů v chromozomu, je náhodný a přitom každý interval může mít stejnou pravděpodobnost výběru, nebo delší či kratší intervaly mohou mít větší pravděpodobnost výběru. Po výběru po jednom intervalu pro každého z rodičů jsou prohozeny jim odpovídající sekvence bitů. Ale pokud by byla překročena maximální dovolená velikost potomka, je křížením vytvořen jen jeden potomek a druhým potomkem je rodič, u něhož by byla překročena dovolená velikost.
metoda:	`crossoverRandomCrossPoint`
hlavička:	`protected void crossoverRandomCrossPoint()`
parametry:	-
popis:	Metoda realizující křížení v náhodném bodě, v cyklu jsou pro křížení vybrání vždy dva po sobě jdoucí jedinci, přičemž rodičovské řetězce si vzájemně prohodí sekvence bitů za náhodně zvoleným bodem křížení.
metoda:	`mutationStructuralCodon`
hlavička:	`protected void mutationStructuralCodon(float mutationProbability, byte codonSize) throws IndexOutOfBoundsException`
parametry:	`mutationProbability` pravděpodobnost s jakou má být kodon zmutován `codonSize` počet bitů, které kódují právě mutovaný kodon
popis:	Metoda realizuje strukturální mutaci, nejprve se provede náhodný pokus, kdy je vygenerováno reálné číslo na intervalu od 0 (zahrnuto) do 1 (není zahrnuto) a pokud je toto číslo menší než hodnota pravděpodobnosti mutace a současně lze přečíst poslední bit kodonu, potom je zmutovat jeden bit v rámci mutovaného kodonu, ale pokud nelze přečíst poslední bit kodonu, potom se vyhodí výjimka. Je-li maximální počet průchodů chromozomem větší jak jedna, potom se pokračuje analogicky od začátku chromozomu.
metoda:	`mutationRandomBits`
hlavička:	`protected void mutationRandomBits()`
parametry:	-
popis:	Metoda realizující mutací náhodných bitů, nejprve se stanoví celkový počet bitů v rámci celé populace, potom pomocí faktoru mutace je vypočten počet bitů, které budou invertovány a v cyklu se vždy náhodně vybere nejprve chromozom a potom pozice bitu, tento bit je následně invertován.
metoda:	`getMigrantsHigherIsBetter`
hlavička:	`protected BitVector[] getMigrantsHigherIsBetter(float[] fitness, int migrantCount, int migrantPower)`
parametry:	`fitness` pole ohodnocení všech jedinců v populaci `migrantCount` počet jedinců určených k migraci `migrantPower` počet jedinců z kterých bude vybrán jeden migrant
popis:	Metoda z populace, pro kterou je volána, vrací pole objektů datového typu `BitVector`, tedy množinu chromozomů, jejíž počet je určen parametrem `migrantCount`. Prvky této množiny jsou vybírány na základě turnajové selekce, což znamená, že ze skupinky chromozomů o velikosti dané parametrem `migrantPower` je vždy vybrán jeden chromozom s nejvyšší hodnotou prvku v poli `fitness`. Tento postup se opakuje, dokud množina nemá potřebný počet prvků, potom je tato množina chromozomů metodou vrácena.

metoda:	`getMigrantsLowerIsBetter`
hlavička:	`protected BitVector[] getMigrantsLowerIsBetter(float[] fitness, int migrantCount, int migrantPower)`
parametry:	`fitness` pole ohodnocení všech jedinců v populaci `migrantCount` počet jedinců určených k migraci `migrantPower` počet jedinců z kterých bude vybrán jeden migrant
popis:	Metoda z populace, pro kterou je volána, vrací pole objektů datového typu `BitVector`, tedy množinu chromozomů, jejíž počet je určen parametrem `migrantCount`. Prvky této množiny jsou vybírány na základě turnajové selekce, což znamená, že ze skupinky chromozomů o velikosti dané parametrem `migrantPower` je vždy vybrán jeden chromozom s nejnižší hodnotou prvku v poli `fitness`. Tento postup se opakuje, dokud množina nemá potřebný počet prvků, potom je tato množina chromozomů metodou vrácena.
metoda:	`placeMigrantsForLower`
hlavička:	`protected int[] placeMigrantsForLower(BitVector[] migrants, float[] fitness, int migrantPower)`
parametry:	`migrants` pole chromozomů `fitness` pole ohodnocení všech jedinců v populaci `migrantPower` počet jedinců z kterých bude vybrán jeden migrant
popis:	Metoda v populaci, pro kterou je volána, nahrazuje prvky pole položky `chromosome`. Ty jsou vybírány na základě turnajové selekce, což znamená, že ze skupinky jedinců o velikosti dané parametrem `migrantPower` je vždy vybrán jeden jedinec s nejnižší hodnotou prvku v poli `fitness` a ten je nahrazen jedním prvkem z pole `migrants`, celkem je nahrazeno tolik chromozomů, kolik jich přináší parametr `migrants`. Metoda vrací pole indexů chromozomů, které byly nahrazeny, aby byly tyto chromozomy následně ohodnoceny.
metoda:	`placeMigrantsForHigher`
hlavička:	`protected int[] placeMigrantsForHigher(BitVector[] migrants, float[] fitness, int migrantPower)`
parametry:	`migrants` pole chromozomů `fitness` pole ohodnocení všech jedinců v populaci `migrantPower` počet jedinců z kterých bude vybrán jeden migrant
popis:	Metoda v populaci, pro kterou je volána, nahrazuje prvky pole položky `chromosome`. Ty jsou vybírány na základě turnajové selekce, což znamená, že ze skupinky jedinců o velikosti dané parametrem `migrantPower` je vždy vybrán jeden jedinec s nejvyšší hodnotou prvku v poli `fitness` a ten je nahrazen jedním prvkem z pole `migrants`, celkem je nahrazeno tolik chromozomů, kolik jich přináší parametr `migrants`. Metoda vrací pole indexů chromozomů, které byly nahrazeny, aby byly tyto chromozomy následně ohodnoceny.

7.2 Třída PopulationAF

Třída `PopulationAF` je potomkem třídy `Population` a reprezentuje populaci pro úlohu aproximace funkcí. Tato třída tedy využívá dědičnosti, což představuje možnost přidat k základní třídě `Population` další vlastnosti a vytvořit tak odvozenou třídu, která je specializovanější. Rozšiřující položkou je `numbers`, je to pole proměnné velikosti, do kterého jsou během mapování z genotypu do fenotypu přidávány přečtené konstanty. Dále třída obsahuje celou řadu statických položek, které jsou společné pro všechny instance této třídy, jedná se například o nastavení typu selekce, křížení nebo mutace, počet bitů kódující terminální a neterminální kodony, dále položky kódující gramatiku, pomocné položky a celou řadu dalších.

metoda:	`PopulationAF`
hlavička:	`public PopulationAF()`
parametry:	-
popis:	Konstruktor třídy, inicializuje datové prvky objektu a volá konstruktor předchůdce.
metoda:	`initStaticFields`
hlavička:	`public static void initStaticFields()`
parametry:	-
popis:	Statická metoda inicializující všechny statické položky třídy.
metoda:	`selection`
hlavička:	`public void selection(float[] fitness)`
parametry:	`fitness` pole ohodnocení všech jedinců v populaci
popis:	Metoda volající metodu pro zvolený typ selekce.
metoda:	`modification`
hlavička:	`public void modification()`
parametry:	-
popis:	Metoda volající metodu pro zvolený typ křížení a mutace.
metoda:	`mutationStructural`
hlavička:	`private void mutationStructural()`
parametry:	-
popis:	Metoda zajišťující strukturální mutaci v rámci celé populace.
metoda:	`mutationStructural_expr`
hlavička:	`private void mutationStructural_expr()`
parametry:	-
popis:	Metoda zajišťující mutaci strukturálních a nestrukturálních kodonů. Algoritmicky probíhá tak, že je postupně generován program a během tohoto generování se na každý přečtený terminál zavolá metoda `mutationStructuralCodon`, která s danou pravděpodobností v kodonu, odpovídající tomuto terminálu, zmutuje jeden bit. Neterminální kodony jsou s danou pravděpodobností mutovány vždy dřív, než dojde k jejich přečtení a tedy určení na jaký neterminál má být rozvinut.
metoda:	`codonToIntForConstant`
hlavička:	`private int codonToIntForConstant() throws IndexOutOfBoundsException`
parametry:	-
popis:	Metoda vrací hodnotu následujícího kodonu kódující konstantu.
metoda:	`codonToIntForIntegerConstant`
hlavička:	`private int codonToIntForIntegerConstant() throws IndexOutOfBoundsException`
parametry:	-
popis:	Metoda vrací hodnotu následujícího kodonu kódující celou část konstanty.
metoda:	`codonToIntForRealConstant`
hlavička:	`private int codonToIntForRealConstant() throws IndexOutOfBoundsException`
parametry:	-
popis:	Metoda vrací hodnotu následujícího kodonu kódující reálnou část konstanty.
metoda:	`constantInteger`
hlavička:	`private int constantInteger()`
parametry:	-
popis:	Metoda volá `codonToIntForConstant` a její vrácenou hodnotu přepočítá na celou konstantu v daném intervalu, potom metoda vrátí skutečnou hodnotu konstanty.

metoda:	`constantReal`
hlavička:	`private double constantReal()`
parametry:	-
popis:	Metoda volá `codonToIntForConstant` a její vrácenou hodnotu přepočítá na reálnou konstantu v daném intervalu, potom metoda vrátí skutečnou hodnotu konstanty.
metoda:	`constantIntegerReal`
hlavička:	`private double constantIntegerReal()`
parametry:	-
popis:	Metoda nejprve volá `codonToIntForIntegerConstant` a její vrácenou hodnotu přepočítá na celou konstantu v daném intervalu, ke které se připočítá reálná konstanta, která je získána voláním metody `codonToIntForRealConstant` a přepočítáním její návratové hodnoty na intervalu od 0 do 1.
metoda:	`constantCustom`
hlavička:	`private double constantCustom()`
parametry:	-
popis:	Metoda, na základě pravidla uživatelské konstanty, pro následující kodon vrátí hodnotu odpovídající uživatelské konstanty.
metoda:	`readChromosome`
hlavička:	`public boolean readChromosome(int chromosomeIndex)`
parametry:	`chromosomeIndex` index chromozomu
popis:	Metoda zajišťující přepis konkrétního chromozomu na posloupnost pravidel, která budou v rámci provádění programu, respektive počítání funkčních hodnot, aplikována. Metoda volá `expression`, která rozvíjí neterminál expr, který zároveň reprezentuje startovací symbol. Metoda vrací logickou hodnotu pravda, jestliže bylo možné konkrétní chromozom přepsat na pravidla a hodnotu nepravda za předpokladu, že to nebylo možné.
metoda:	`expression`
hlavička:	`private void expression()`
parametry:	-
popis:	Metoda volá `getRuleNonterminal`, která vrací pravidlo neterminálu, jenž je přidáno do dynamického pole `rules` a na jeho základě dojde k větvení dle gramatiky, kterou uživatel definoval. Tato metoda také zajišťuje plnění položky `subTreesBlocks`, která pro každý chromozom udává intervaly, které budou mít možnost se v rámci strukturálního křížení vyměnit.
metoda:	`getRuleNonterminal`
hlavička:	`private byte getRuleNonterminal()`
parametry:	-
popis:	Metoda vrací pravidlo neterminálu pro následující kodon.
metoda:	`getRuleOperatorBinary`
hlavička:	`private byte getRuleOperatorBinary()`
parametry:	-
popis:	Metoda vrací pravidlo binárního operátoru pro následující kodon.
metoda:	`getRuleOperatorUnary`
hlavička:	`private byte getRuleOperatorUnary()`
parametry:	-
popis:	Metoda vrací pravidlo unárního operátoru pro následující kodon.

metoda:	`getRuleVariable`
hlavička:	`private byte getRuleVariable()`
parametry:	-
popis:	Metoda vrací pravidlo proměnné pro následující kodon.
metoda:	`getRuleConstantType`
hlavička:	`private byte getRuleConstantType()`
parametry:	-
popis:	Metoda vrací pravidlo typu konstanty pro následující kodon.
metoda:	`readLastChromosomeToString`
hlavička:	`public String readLastChromosomeToString()`
parametry:	-
popis:	Metoda jednak zajišťuje přepis posledního chromozomu na posloupnost pravidel a jednak tyto pravidla převádí na řetězce, čímž postupně vytváří konečnou podobu matematického výrazu, který je zobrazen uživateli. Smysl přepisu posledního chromozomu spočívá v tom, že poslední prvek položky `chromosome` je vyhrazen pro nejlepší řešení v rámci populace. Metoda volá `expressionToString`, která rozvíjí neterminál expr, který zároveň reprezentuje startovací symbol.
metoda:	`exprToString`
hlavička:	`private String exprToString ()`
parametry:	-
popis:	Metoda volá `getRuleNonterminal`, která vrací pravidlo neterminálu, jenž je přidáno do dynamického pole `rules` a na jeho základě dojde k větvení dle gramatiky, kterou uživatel definoval. Metoda vrací vzniklý řetězec.
metoda:	`opBinExprExprToString`
hlavička:	`private String opBinExprExprToString()`
parametry:	-
popis:	Metoda volá `getRuleOperatorBinary`, která vrací pravidlo binární operátoru, jenž je přidáno do dynamického pole `rules`. Metoda vrací vzniklý řetězec.
metoda:	`opUnExprToString`
hlavička:	`private String opUnExprToString()`
parametry:	-
popis:	Metoda volá `getRuleOperatorUnary`, která vrací pravidlo unárního operátoru, jenž je přidáno do dynamického pole `rules`. Metoda vrací vzniklý řetězec.
metoda:	`varToString`
hlavička:	`private String varToString()`
parametry:	-
popis:	Metoda volá `getRuleVariable`, která vrací pravidlo proměnné, jenž je přidáno do dynamického pole `rules`. Metoda vrací řetězec odpovídající proměnné.
metoda:	`constantIntegerToString`
hlavička:	`private String constantIntegerToString()`
parametry:	-
popis:	Metoda volá `constantInteger`, která vrací hodnotu celé konstanty, jenž je přidána do dynamického pole `numbers`. Metoda vrací řetězec konstanty.

7.3 Třída ApproximationFunction

Třída reprezentuje jeden evoluční proces pro úlohy aproximace funkcí. Implementuje rozhranní `Runnable`, pomocí kterého třída získává vlastnosti vlákna. Základní položku třídy je `population` typu `PopulationAF`, je potomkem třídy `Population`, a reprezentuje populaci pro úlohu aproximace funkcí. Dále třída obsahuje důležité položky jako `fitness`, pole uchovávající ohodnocení všech jedinců v populaci, `tolerationValues`, pole uchovávající funkční hodnoty s připočítanou aktuální tolerancí pro horní i dolní mez tolerančního pásma. Dále třída obsahuje statické položky, které jsou společné pro všechny instance této třídy, jedná se například o nastavení typu ohodnocení, dále položky uchovávající hodnoty proměnných a funkční hodnoty.

metoda:	`ApproximationFunction`
hlavička:	`public ApproximationFunction()`
parametry:	-
popis:	Konstruktor třídy, inicializuje datové prvky objektu.
metoda:	`initStaticFields`
hlavička:	`public static void initStaticFields()`
parametry:	-
popis:	Statická metoda inicializující všechny statické položky třídy.
metoda:	`getVariableValues`
hlavička:	`public static double[][] getVariableValues()`
parametry:	-
popis:	Statická metoda vrací pole hodnot proměnných uživatelem vybrané úlohy.
metoda:	`getFunctionalValues`
hlavička:	`public static double[] getFunctionalValues(double[][] pVariableValues)`
parametry:	`pVariableValues` pole hodnot proměnných
popis:	Statická metoda vrací pole funkčních hodnot pro pole hodnot proměnných.
metoda:	`calculateTolerationValues`
hlavička:	`private void calculateTolerationValues ()`
parametry:	-
popis:	Metoda inicializuje položku `tolerationValues` a vypočítá její počáteční hodnoty.
metoda:	`run`
hlavička:	`public void run()`
parametry:	-
popis:	Protože třída získala vlastnosti vlákna implementací rozhraní `Runnable`, které obsahuje pouze jednu metodu `run`, a proto tuto metodu musíme v třídě implementovat. Metoda zajišťuje provádění algoritmu gramatické evoluce.
metoda:	`valuationPopulation_dynamicToleration`
hlavička:	`private void valuationPopulation_dynamicToleration()`
parametry:	-
popis:	Metoda v cyklu volá `dynamicToleration`, pomocí které ohodnotí všechny jedince v populaci za použití účelové funkce dynamické toleranční pásmo.
metoda:	`valuationPopulation_leastSquares`
hlavička:	`private void valuationPopulation_leastSquares()`
parametry:	-
popis:	Metoda v cyklu volá `leastSquares`, pomocí které ohodnotí všechny jedince v populaci za použití účelové funkce součet kvadratických odchylek.

metoda:	`dynamicToleration`
hlavička:	`private void dynamicToleration(int i)`
parametry:	`i` index chromozomu
popis:	Metoda volá `population.readChromosome` s parametrem `i`, pokud její návratová hodnota je pravda, potom lze chromozom ohodnotit, pokud je nepravda, chromozom je ohodnocen nejhorším možným ohodnocením. Ohodnocení probíhá v cyklu v kontextu s velikostí pole hodnot proměnných a volá metodu `expr`, přičemž její návratové hodnota leží nebo neleží v tolerančním pásmu.
metoda:	`leastSquares`
hlavička:	`private void leastSquares(int i)`
parametry:	`i` index chromozomu
popis:	Metoda volá `population.readChromosome` s parametrem `i`, pokud její návratová hodnota je pravda, potom lze chromozom ohodnotit, pokud je nepravda, chromozom je ohodnocen nejhorším možným ohodnocením. Ohodnocení probíhá v cyklu v kontextu s velikostí pole hodnot proměnných a volá metodu `expr`, druhá mocnina její návratové hodnoty je přičtena k hodnotě účelové funkce.
metoda:	`expr`
hlavička:	`private double expr()`
parametry:	-
popis:	Metoda přečte pravidlo neterminálu z `population.relus` a na jeho základě zavolá odpovídající metodu. Návratovou hodnotou metody je funkční hodnota, která vznikla rozvinutím přečteného pravidla.
metoda:	`opBinExprExpr`
hlavička:	`private double opBinExprExpr()`
parametry:	-
popis:	Metoda přečte pravidlo binárního operátoru z `population.relus` a na jeho základě provede binární operaci dvou hodnot získaných zavoláním metod `expr`.
metoda:	`opUnExpr`
hlavička:	`private double opUnExpr()`
parametry:	-
popis:	Metoda přečte pravidlo unárního operátoru z `population.relus` a na jeho základě provede unární operaci hodnoty získané zavoláním metody `expr`.
metoda:	`var`
hlavička:	`private double var()`
parametry:	-
popis:	Metoda přečte pravidlo proměnné z `population.relus` a vrátí její hodnotu.
metoda:	`evaluationDynamicRange`
hlavička:	`private void evaluationDynamicRange()`
parametry:	-
popis:	Metoda vyhodnotí všechny jedince v populaci, najde a uloží nejlepší řešení.
metoda:	`changeCurrentToleration`
hlavička:	`private void changeCurrentToleration()`
parametry:	-
popis:	Metoda, na základě splněných podmínek, zmenší toleranční pásmo.

7.4 Třída ControlUnitAF

Třída slouží pro řízení, správu a vyhodnocení evolučních procesů pro úlohy aproximace funkcí, dále zajišťuje aktualizaci všech informačních prvků grafického uživatelského rozhraní (GUI). Třída implementuje rozhranní `Runnable`, pomocí kterého získává vlastnosti vlákna. Základní položku třídy je `processes` typu `ApproximationFunction`, je to pole, které reprezentuje evoluční procesy pro úlohu aproximace funkcí. Dále třída obsahuje celou řadu položek pro řízení a běh procesů v rámci paralelní gramatické evoluce.

metoda:	`ControlUnitAF`
hlavička:	`public ControlUnitAF(GEView view)`
parametry:	`view` grafické uživatelské rozhraní
popis:	Konstruktor třídy, inicializuje datové prvky objektu.
metoda:	`initPGEStructure`
hlavička:	`private void initPGEStructure()`
parametry:	-
popis:	Metoda vytváří strukturu propojení procesů pro paralelní gramatickou evoluci.
metoda:	`createProcesses`
hlavička:	`private void createProcesses()`
parametry:	-
popis:	Metoda inicializuje položku `processes`.
metoda:	`runProcesses`
hlavička:	`private void runProcesses()`
parametry:	-
popis:	Metoda v cyklu spustí běh vlákna pro všechny procesy.
metoda:	`run`
hlavička:	`public void run()`
parametry:	-
popis:	Protože třída získala vlastnosti vlákna implementací rozhraní `Runnable`, které obsahuje pouze jednu metodu `run`, a proto tuto metodu musíme v třídě implementovat. Metoda pracuje v cyklu, kde pomocí metody `suspendProcesses` pozastaví běh všech procesů, potom je-li nastaveno, realizuje paralelní gramatickou evoluci, dále vyhodnotí procesy a aktualizuje GUI, nakonec obnoví běh všech procesů a běh vlastního vlákna na určitou dobu uspí.
metoda:	`suspendPerform`
hlavička:	`private void suspendPerform()`
parametry:	-
popis:	Metoda volá `suspendProcesses`, `updateGUI` a pozastaví běh vlastního vlákna.
metoda:	`suspendProcesses`
hlavička:	`private void suspendProcesses()`
parametry:	-
popis:	Metoda naplánuje pozastavení běhu všech procesů a počká do jejich pozastavení.
metoda:	`notifyPerform`
hlavička:	`public synchronized void notifyPerform()`
parametry:	-
popis:	Metoda v cyklu obnoví běh všech procesů a běh vlastního vlákna.

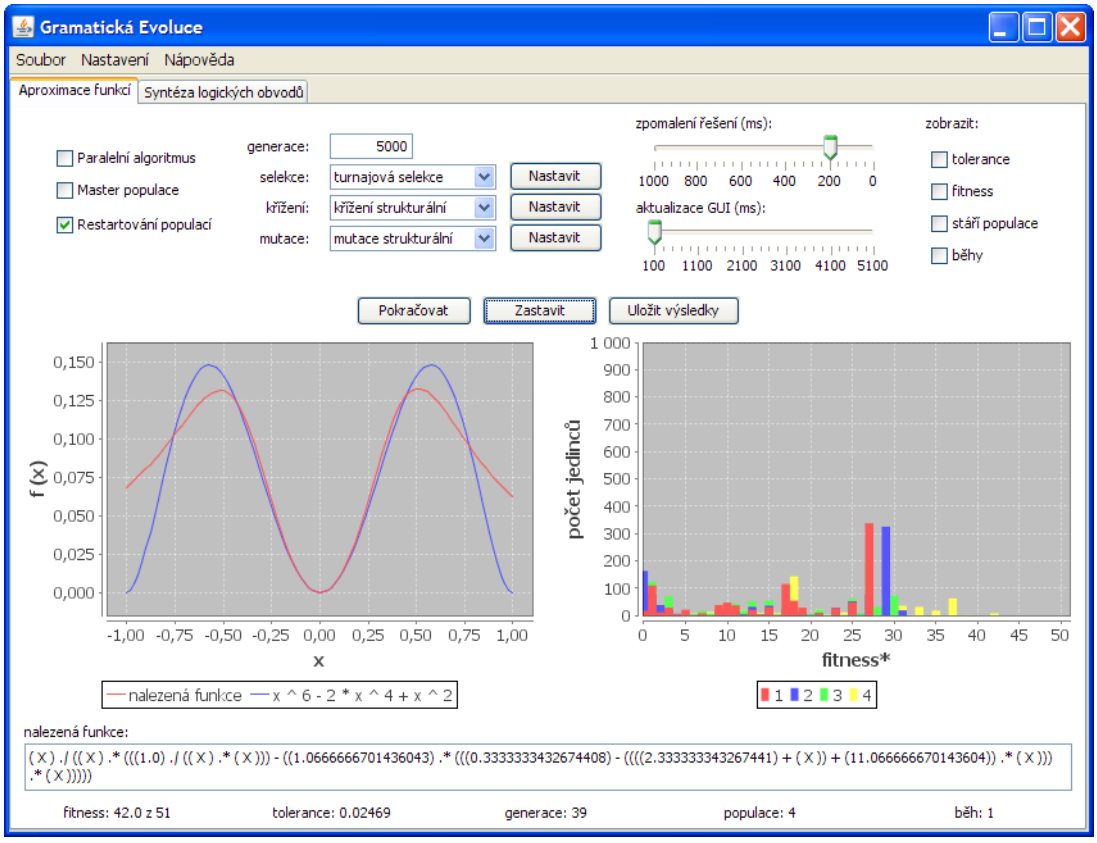
metoda:	`notifyProcesses`
hlavička:	`private void notifyProcesses()`
parametry:	-
popis:	Metoda v cyklu obnoví běh všech procesů.
metoda:	`parallelGrammaticalEvolution`
hlavička:	`private void parallelGrammaticalEvolution()`
parametry:	-
popis:	Metoda za předpokladu, že je splněna podmínka pro kopírování jedinců v rámci paralelní gramatické evoluce, to znamená, že od posledního kopírování uplynul daný počet generací na populaci, potom v cyklu v kontextu s položkou `PGEStructure` se z procesu, který je v ní zapsán jako zdrojový, zavoláním příslušné metody získá pole chromozomů o velikosti odpovídající počtu migrantů, které je následně uloženo zavoláním příslušné metody na proces, který je v položce `PGEStructure` zapsán jako cílový.
metoda:	`masterPopulation`
hlavička:	`private void masterPopulation()`
parametry:	-
popis:	Metoda za předpokladu, že je splněna podmínka pro kopírování jedinců v rámci paralelní gramatické evoluce s master populací, to znamená, že od posledního kopírování uplynul daný počet generací na populaci, potom se v cyklu z každého procesu, zavoláním příslušné metody, získá pole chromozomů o velikosti odpovídající počtu migrantů, které je následně uloženo zavoláním příslušné metody na proces pracující s master populací.
metoda:	`updateGUI`
hlavička:	`private void updateGUI()`
parametry:	-
popis:	Metoda zajišťuje aktualizaci všech informačních prvků GUI.

8 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní, známé pod zkratkou GUI (Graphic User Interface), je obrazové rozhraní k programu, jehož účelem je usnadnění používání programů, tak že poskytuje konzistentní vnější podobu programu a je vybaveno vstupními poli, tlačítky a dalšími prvky. V Javě jsou dvě grafická uživatelská prostředí, starší AWT (Abstract Windowing Toolkit) a novější JFC Swing (Java Foundation Classes). Já jsem se rozhodl použít prostředí Swing.

Grafický vzhled celé aplikace je navržen v NetBeans IDE 6.5. Hlavní okno aplikace má dvě záložky „Aproximace funkcí“ a „Syntéza logických obvodů“, přičemž každá slouží pro řešení příslušné úlohy. Prostřednictvím hlavního menu lze nastavit parametry populace, paralelní gramatické evoluce a restartování populací.

Prostřednictvím nastavení aproximace funkcí lze za data pro aproximaci zvolit data generovaná testovací funkcí dle výběru a to na zvoleném definičním oboru s krokem diskretizace, nebo mohou být zadány přímo od uživatele a eventuálně načteny ze souboru. Dále je možné vybrat hodnotící funkci a nastavit gramatiku, která bude použita pro dekódování lineárního genomu, lze ji modifikovat dle požadavků na výslednou funkci a současně změnou gramatiky dochází ke změně prostoru prohledávání.



Obr. 26: Grafické uživatelské rozhraní pro úlohy aproximace funkcí.

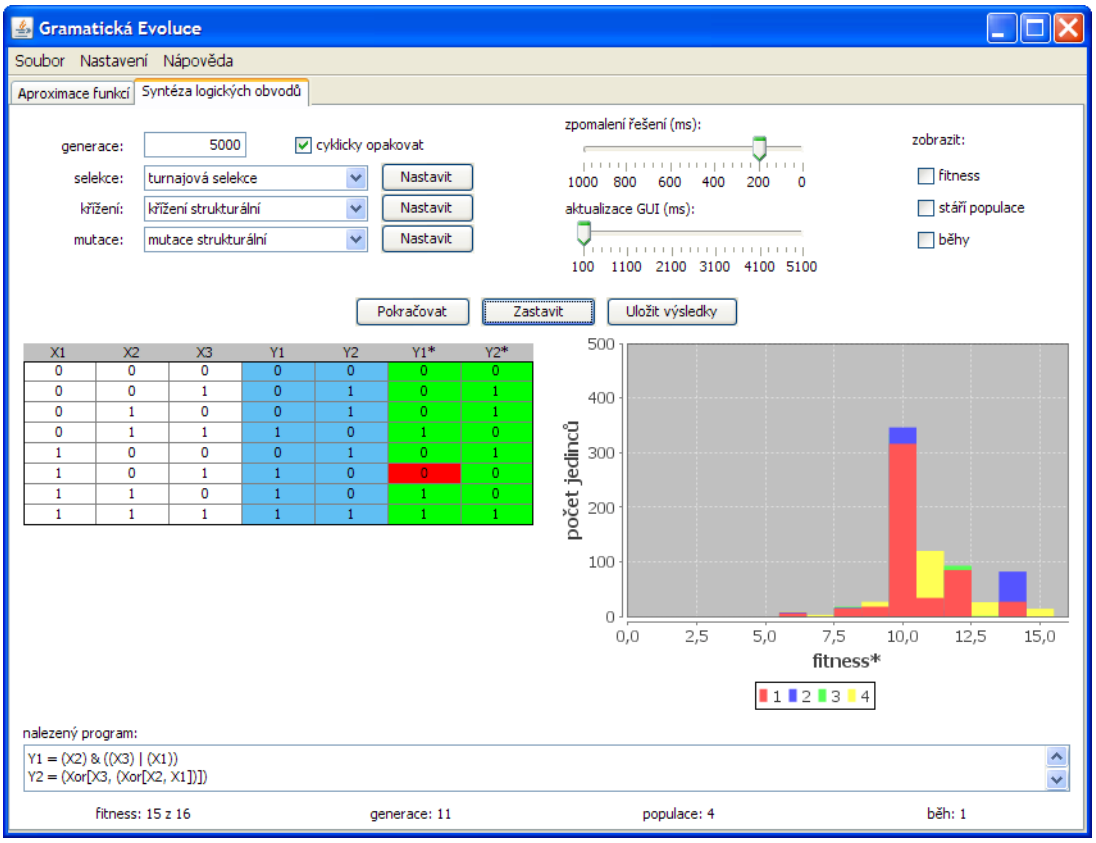
Zaškrtávací pole paralelní algoritmus, master populace a restartování populací určují, zda v rámci evolučního procesu bude probíhat paralelní gramatická evoluce a zda budou populace dle nastavení po určité době restartovány. Pole generace určuje, do jaké generace se budou populace v evolučním procesu vyvíjet. Rozbalovací seznamy selekce, křížení a mutace definují metody těchto operátorů, které budou v rámci algoritmu gramatické evoluce použity. Jezdcem zpomalení řešení se stanoví čas v milisekundách, o který se zpomalí každá generace evolučního procesu a jezdcem aktualizace GUI se stanoví časová perioda v milisekundách, v jaké budou všechny procesy vyhodnoceny a prostřednictvím GUI zobrazeny výsledky.

Zaškrtávací pole tolerance, fitness, stáří populace a běhy určují, zda budou zobrazeny samostatná okna s grafy zobrazující aktuální tolerance, ohodnocení, stáří a běh pro všechny populace ve vývojovém procesu.

V levé části GUI pro úlohu aproximace funkcí je zobrazen graf s hledanou a dosud nejlepší nalezenou funkci a pro úlohu syntézy logických obvodů je zobrazena pravdivostní tabulka se vstupy, výstupy a výstupy, které generuje dosud nejlepší nalezené řešení, přičemž zelenou barvou jsou vyznačena úspěšná a červenou neúspěšná místa v tabulce.

V pravé části GUI je pro obě úlohy zobrazen populační statistický histogram, je to sloupcový graf, jehož základna je tvořená možným rozsahem ohodnocení (fitness), a každému prvku této základny je přiřazena odpovídající četnost v populaci, respektive úhrn kolikrát se příslušná hodnota ohodnocení v populaci opakuje.

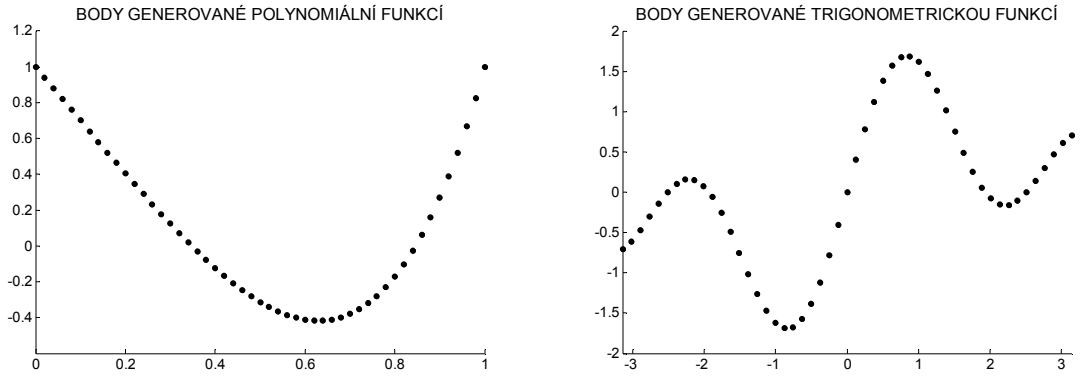
V dolní části jsou zobrazeny informace pro dosud nejlepší nalezené řešení, respektive nalezenou funkci nebo program, ohodnocení, tolerancí, generaci, populaci a běh, ve které bylo toto řešení nalezeno.



Obr. 27: Grafické uživatelské rozhraní pro syntézu logických obvodů.

9 Testy a statistika

Pro testování aproximace funkcí jsem zvolil dvě testovací úlohy. Data první úlohy, dále označována jako úloha 1, jsou generována polynomem čtvrtého řádu: $3 * x^4 - 3 * x + 1$, na intervalu od 0 do 1 s krokem diskretizace 0,02. Data druhé úlohy, dále označována jako úloha 2, jsou generována trigonometrickou funkcí $sin(x) + sin(2{,}5 * x)$, na intervalu od $-\pi$ do π s krokem diskretizace ~0,126. Vygenerované body pro tyto úlohy jsou vidět na následujícím obrázku.



Obr. 28: Vygenerované body pomocí polynomiální a trigonometrické funkce.

V testech sledujeme jednak jak si gramatická evoluce vede při aproximaci výše uvedených vygenerovaných dat a jednak vliv hodnotícího kritéria, respektive použité účelové funkce, na získané výsledky. Dále zkoumáme přínos paralelní gramatické evoluce na výsledná řešení a vliv změny jejich parametrů. Každá simulace aproximace funkce pomocí gramatické evoluce pracuje s jednou populací a paralelní gramatické evoluce se čtyřmi populacemi, přičemž evoluční vývoj je ukončen po dosažení jednoho tisíce generací. Takovouto simulaci budeme nazývat běh. V následujícím textu budou používány následující pojmy, které se vztahují k nejlepšímu řešení v čase, které s v rámci jednoho běhu vyskytuje:

Tab. 4: Použité zkratky a jejich význam.

Použité zkratky účelových funkcí, respektive hodnot účelových funkcí (pozorovatele):	
SDI	součet hodnot ležící v dynamickém tolerančním pásmu (sum dynamic interval)
SSE	součet kvadratických odchylek (sum square error)
SAE	součet absolutních odchylek (sum absolute error)
Použité zkratky pro další pozorovatele:	
FSSE	součet kvadratických odchylek pro nejlepší nalezené řešení v rámci běhu
SMI	součet hodnot, v rámci všech běhů, ležících v průměrném koncovém tolerančním pásmu získané z testu používající SDI účelovou funkci
Použité statistické zkratky:	
MEAN	průměr (mean)
MEDIAN	medián (median)
SD	standardní odchylka (standard deviation)
Použité evoluční zkratky:	
GE	gramatická evoluce
PGE	paralelní gramatická evoluce
PGE – JK	jednosměrná kruhová migrační PGE
PGE – V	vzájemná migrační PGE
PGE – MS	migrační PGE s master populací

Tab. 5: Seznam testů pro polynomiální úlohu.

úloha	test	účelová funkce	pozorovatel	typ grafu	obrázek
1		SDI	SDI v čase		
1	test 1	SSE	SSE v čase	XY spojnicový	29
1		SAE	SAE v čase		
1	test 2	data z testu 1	SSE	histogram	30
1	test 3	data z testu 1	SMI	sloupcový	
1	test 4	data z testu 1	průběhy 75 %	XY spojnicový	31
1	test 5	data z testu 1	FSSE 75 %	Box - Whisker	

Tab. 6: Seznam testů pro trigonometrickou úlohu.

úloha	test	účelová funkce	pozorovatel	typ grafu	obrázek
2		SDI	SDI v čase	XY spojnicový	
2	test 6	SSE	SSE v čase	XY spojnicový	32
2		SAE	SAE v čase	XY spojnicový	
2	test 7	data z testu 6	SSE	histogram	33
2	test 8	data z testu 6	SMI	sloupcový	
2	test 9	data z testu 6	průběhy 75 %	XY spojnicový	34
2	test 10	data z testu 6	FSSE 75 %	Box - Whisker	

Testování gramatické evoluce na datech generovaných polynomiální a trigonometrickou funkcí je postaveno na stejném postupu vyhodnocení. Nejprve se realizuje test 1, tedy pro všechny tří účelové funkce se provede 250 běhů a hodnoty účelových funkcí se zobrazí v čase pomocí grafů. Potom se vyhodnotí výsledná řešení všech tří účelových funkcí pomocí histogramů, kde základna je rozdělena na SSE intervaly podle následující tabulky a pro každý interval se do sloupce vynese součet běhů, u nichž hodnota SSE konečného řešení leží v tomto intervalu.

Tab. 7: SSE intervaly pro histogramy.

od	do
0	0,001
0,001	0,002
0,002	0,004
0,004	0,008
0,008	0,016
0,016	0,032
0,032	0,064
0,064	0,128
0,128	0,256
0,256	0,512
0,512	1,024

od	do
1,024	2,048
2,048	4,096
4,096	8,192
8,192	16,384
16,384	32,768
32,768	65,536
65,536	131,072
131,072	262,144
262,144	524,288
524,288	1048,576
1048,576	∞

Z 250 běhů získaných použitím účelové funkce SDI se vypočítá velikost průměrného tolerančního pásma a to tak, že se zprůměrují dosažená toleranční pásma všech běhů, přičemž tato hodnota je použita pro výpočet SMI, pomocí které lze mezi sebou porovnávat jednotlivé účelové funkce. V testech 4 a 9 jsou pro obě úlohy zobrazeny průběhy křivek, které generuje 75 % nejlepších konečných řešení a v testech 5 a 10 je těchto 75 % řešení statisticky, z pohledu SSE, vyhodnoceno.

Tab. 8: Seznam testů pro paralelní gramatickou evoluci.

test	struktura	populace	pozorovatel	typ grafu	obrázek
test 11	SGE JK PGE V PGE M PGE	4 x 250	SSE v čase	XY spojnicový	35
test 12	data z testu 11		SSE	histogram	35
test 13	data z testu 11		průběhy 75 % FSSE 75 %	XY spojnicový Box - Whisker	36
test 14	SGE M PGE M PGE M PGE	4 x 250 4 x 125 4 x 250 4 x 375	SSE v čase	XY spojnicový	37
test 15	data z testu 14		SSE	histogram	37

Vliv paralelní gramatické evoluce na výsledná konečná řešení je testován na datech generovaných trigonometrickou funkcí. Jako účelová funkce je pro všechny testy zvolena součet kvadratických odchylek. V rámci testu 11 je provedeno 250 běhů pro tři typy migračních struktur a 250 běhů běžné gramatické evoluce, ale se čtyřmi populacemi. Potom se hodnoty účelových funkcí zobrazí v čase pomocí grafů. V rámci testu 12 se výsledná řešení vyhodnotí pomocí histogramů, v testu 13 jsou zobrazeny průběhy křivek, které generuje 75 % nejlepších konečných řešení a dále je těchto 75 % řešení statisticky, z pohledu SSE, vyhodnoceno.

V testu 14 sledujeme vliv velikosti populací, jednak v rámci paralelní gramatické evoluce, ale také vůči běžné gramatické evoluci se čtyřmi populacemi. V rámci testu je provedeno 250 běhů pro všechny tři typy velikosti populací migrační gramatické evoluce a stejný počet běhů pro běžené gramatické evoluce, potom se hodnoty účelových funkcí zobrazí v čase pomocí grafů. V rámci testu 15 se výsledná řešení vyhodnotí pomocí histogramů.

Tab. 9: Seznam testů pro vliv změny parametrů PGE.

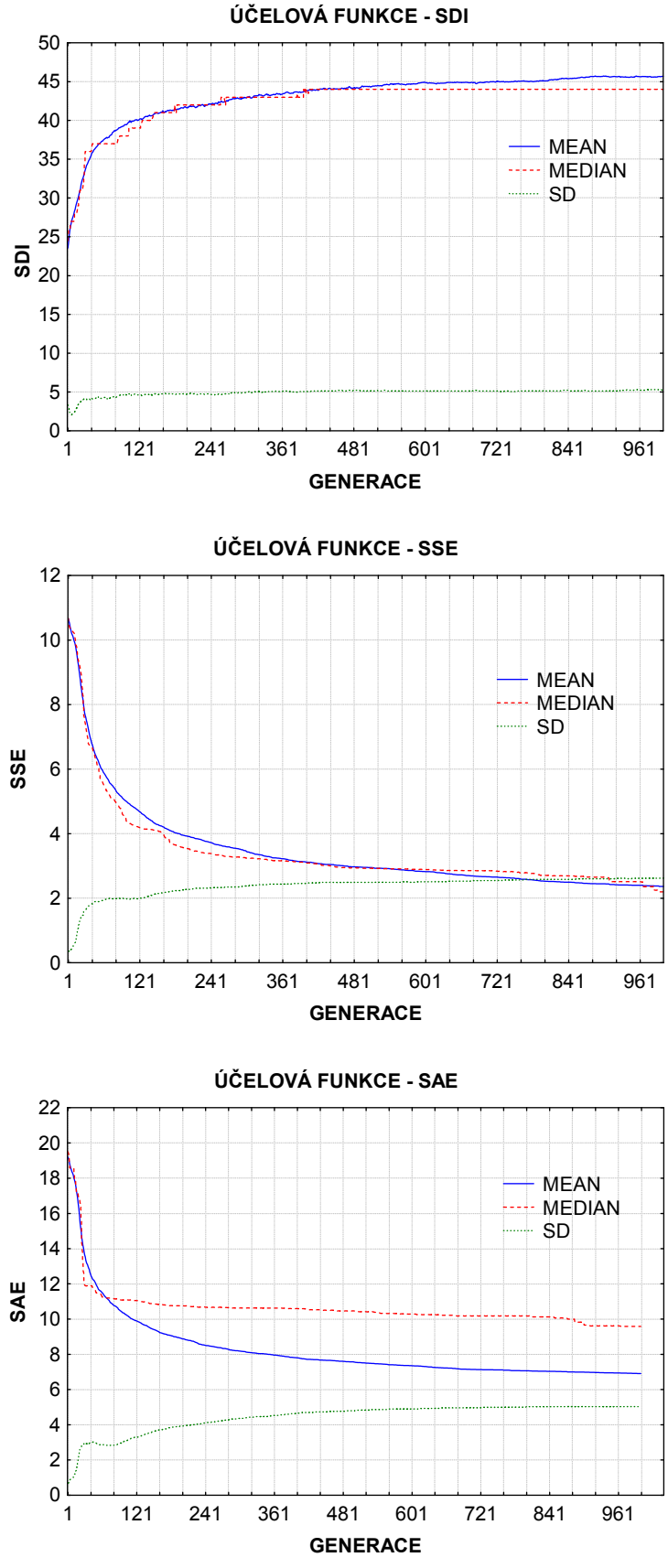
test	frekvence	počet migrantů	síla migrace	pozorovatel	obrázek
test 16	10, 50, 100	12	125	SSE	38
test 17	50	2, 12, 25	125	SSE	39
test 18	50	12	25, 125, 250	SSE	40

Testy 16, 17 a 18 se vztahují k paralelní gramatické evoluci a sledujeme v nich vliv změny frekvence migrace, počtu migrantů a síly migrace. Tyto tři testy pracují se stejnou strukturou migrační gramatické evoluce a to s master populací o stejné velikosti 4 x 250 jedinců a jako účelová funkce je použita součet kvadratických odchylek. Potom se hodnoty účelových funkcí zobrazí v čase pomocí grafů.

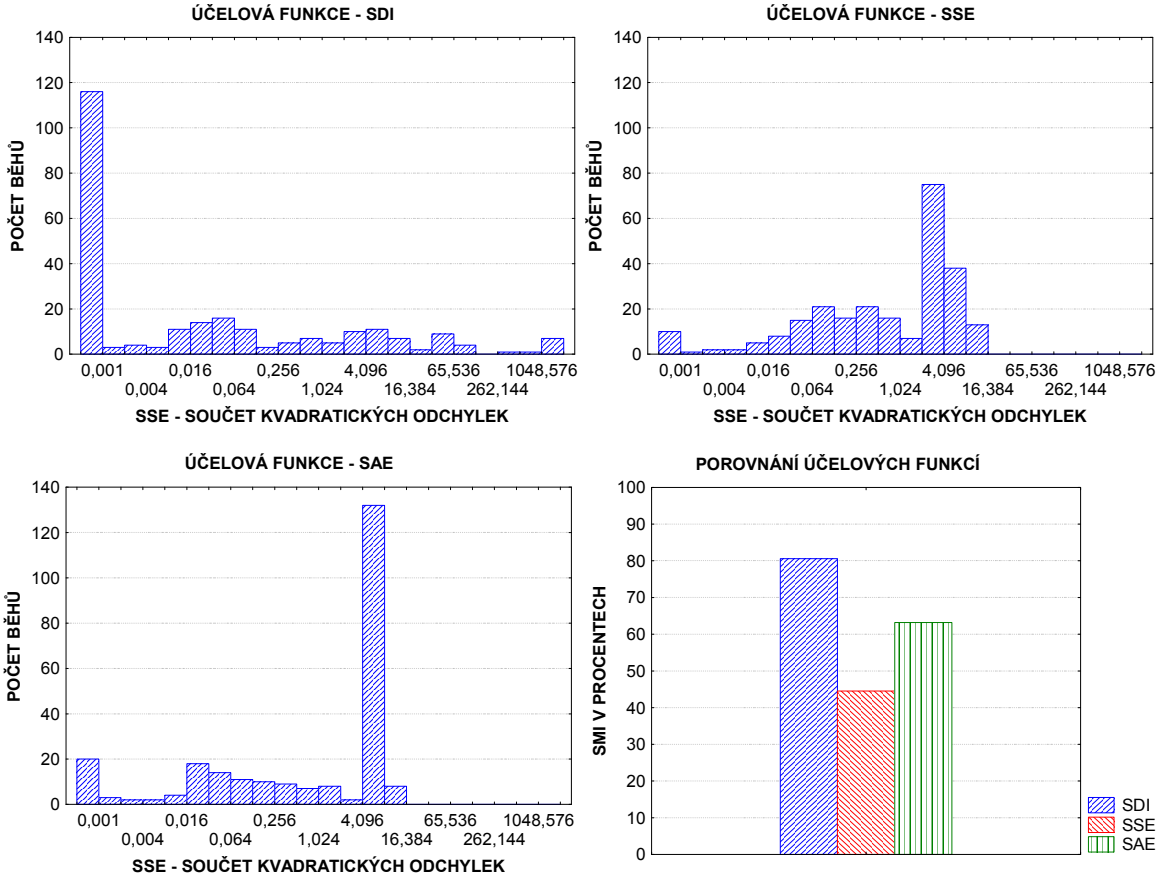
Tab. 10: Společné prvky nastavení provedených testů.

počet generací:	1000 (počáteční velikost chromozomů v intervalu od 16 do 100)
gramatika	T = {+, -, *, /, X, konstanty}
selekce:	turnajová (síla: 2)
křížení:	strukturální (faktor: 0,8; max. délka potomka: 400;
mutace:	strukturální (faktor: 0,8; operátory: 4 %, konstanty: 10 %)

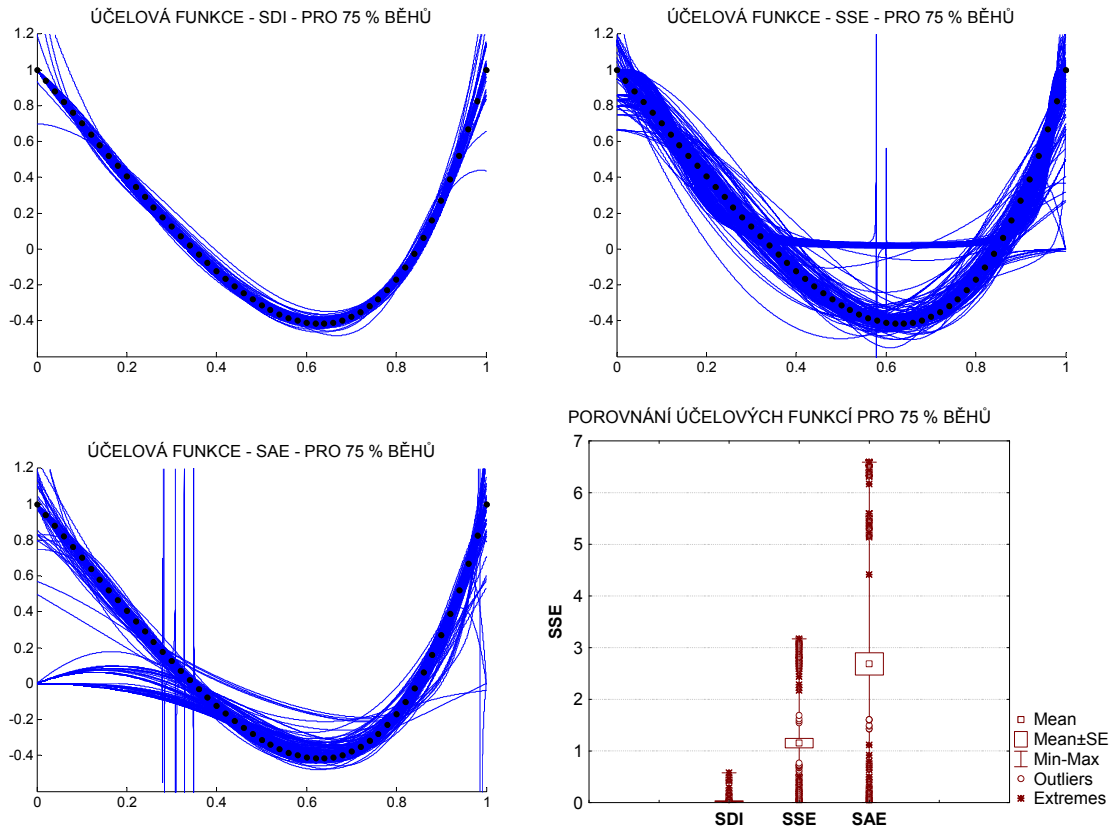
9.1 Polynomiální úloha



Obr. 29: Výsledky testu 1.

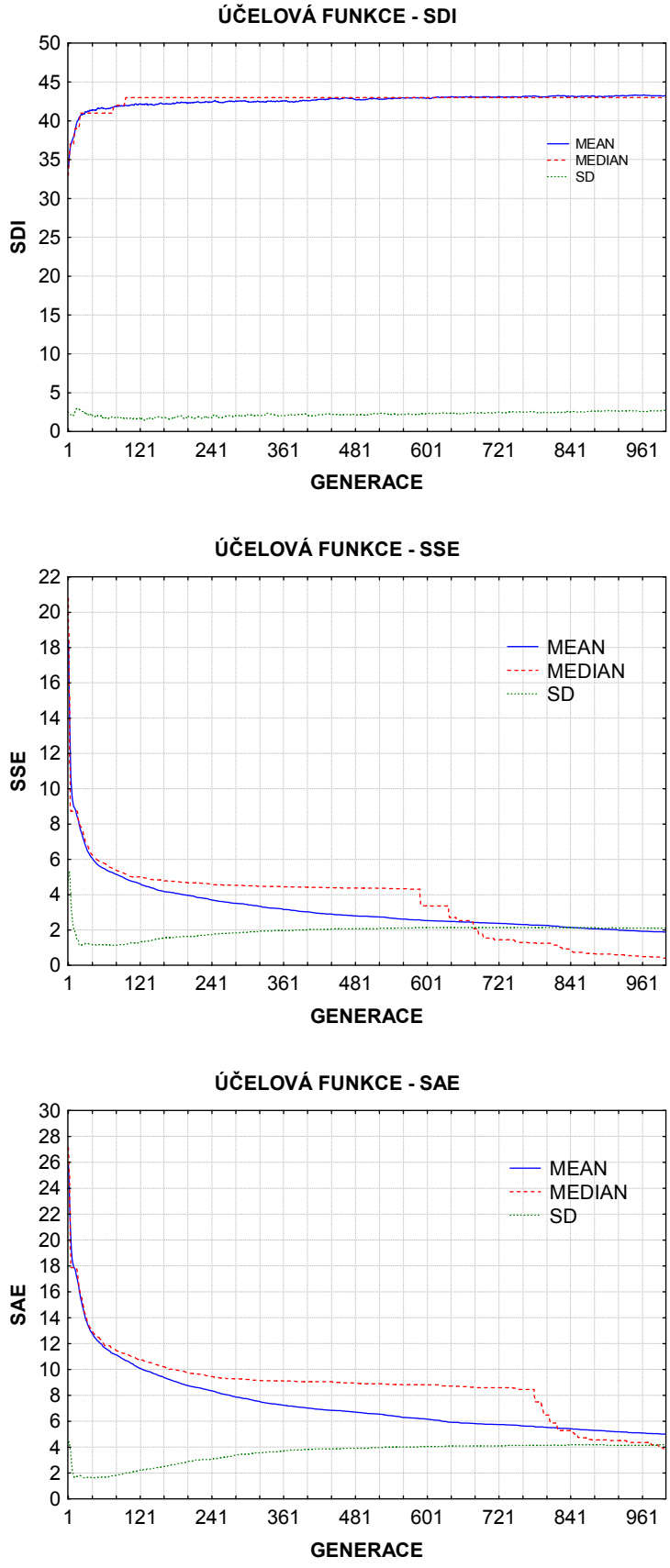


Obr. 30: Výsledky testů 2 a 3.

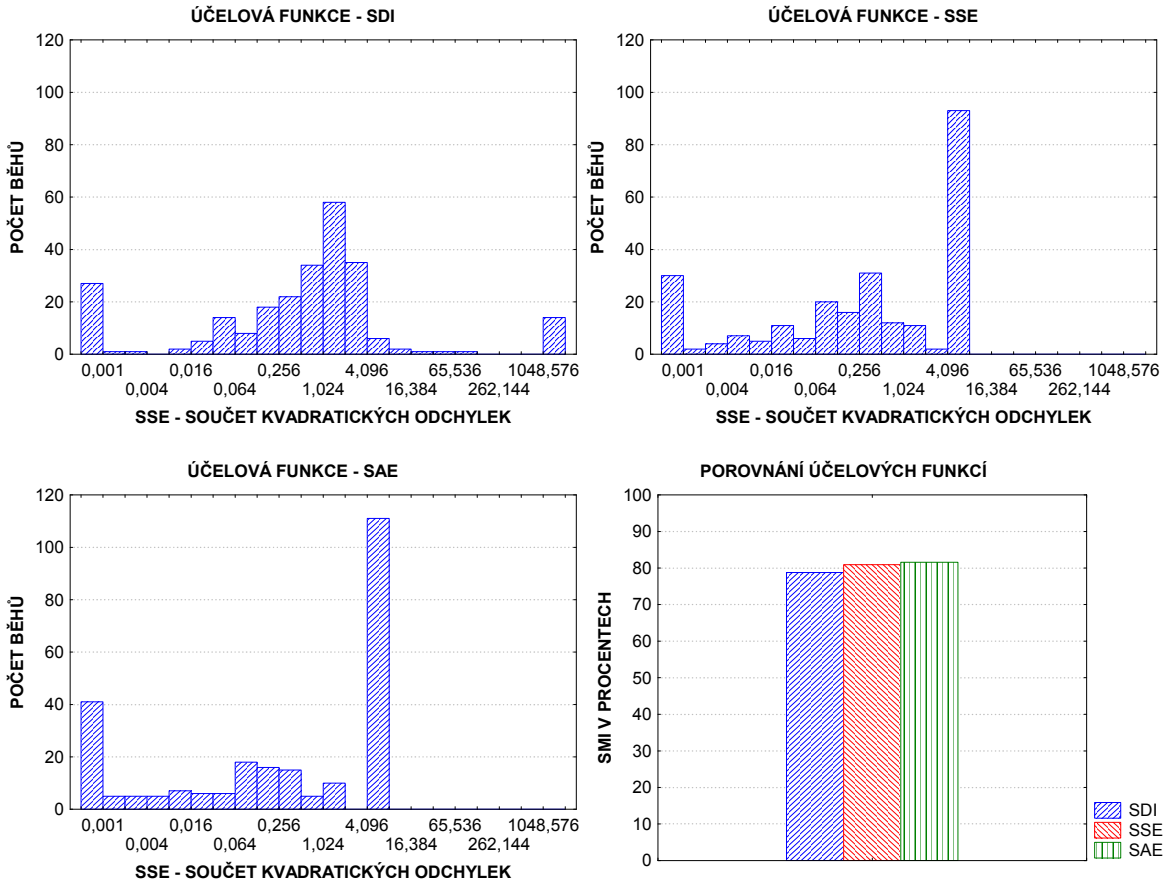


Obr. 31: Výsledky testů 4 a 5.

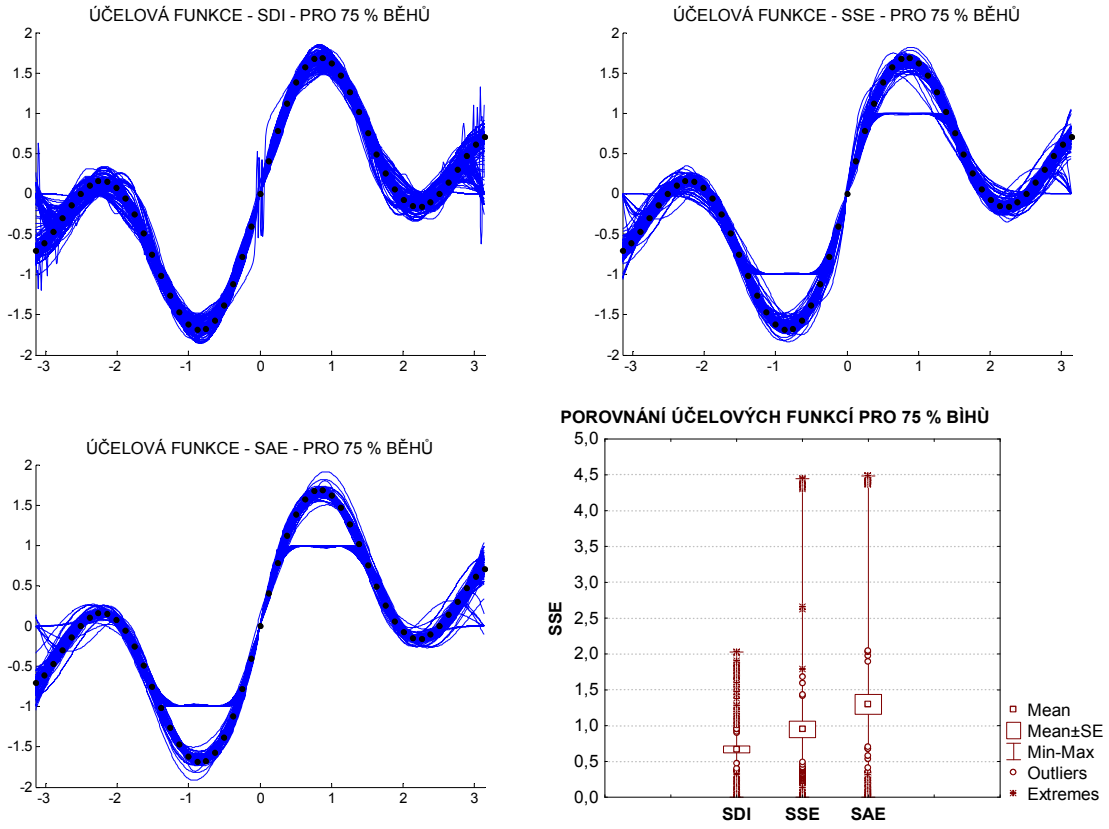
9.2 Trigonometrická úloha



Obr. 32: Výsledky testu 6.

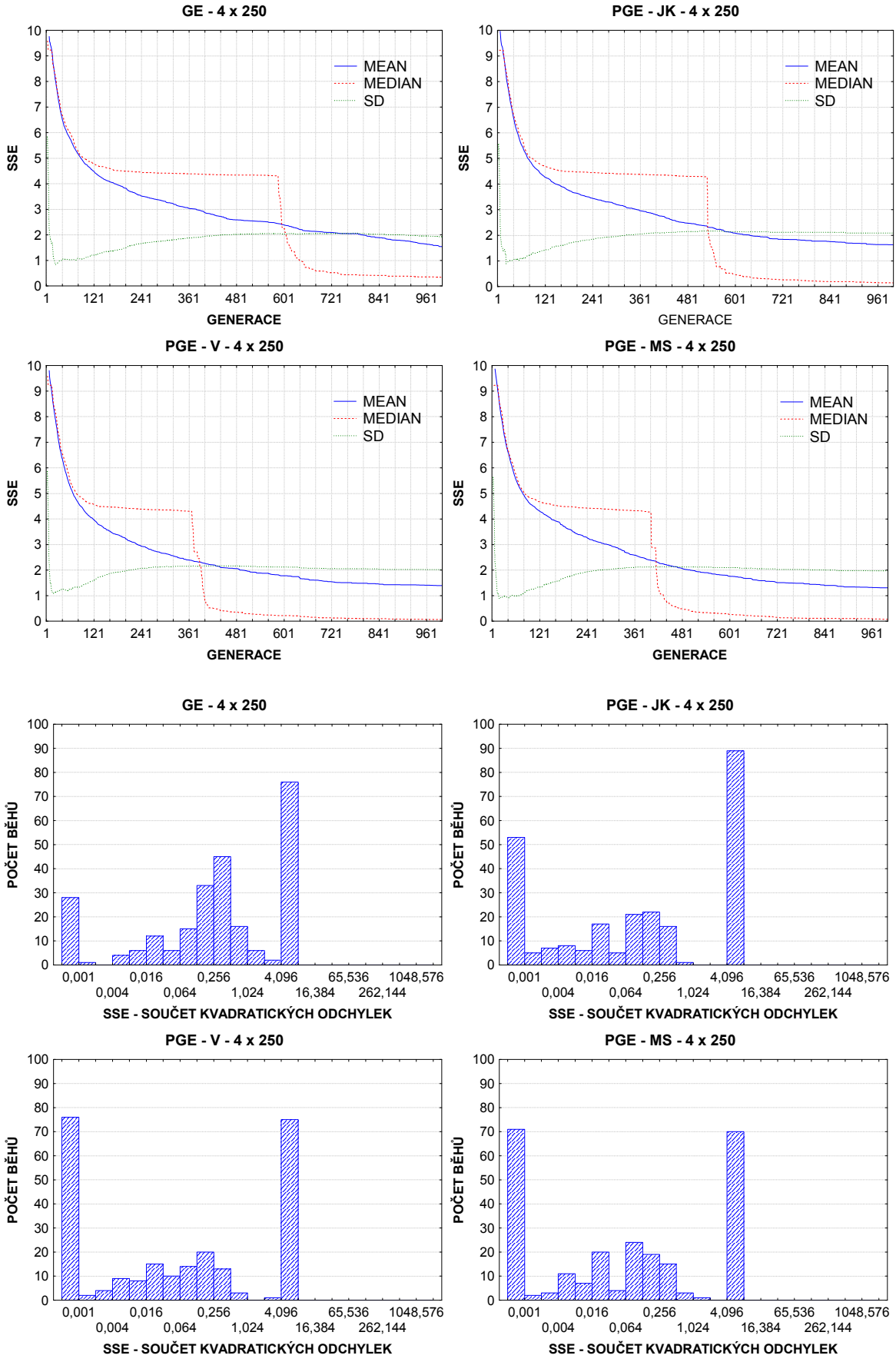


Obr. 33: Výsledky testů 7 a 8.

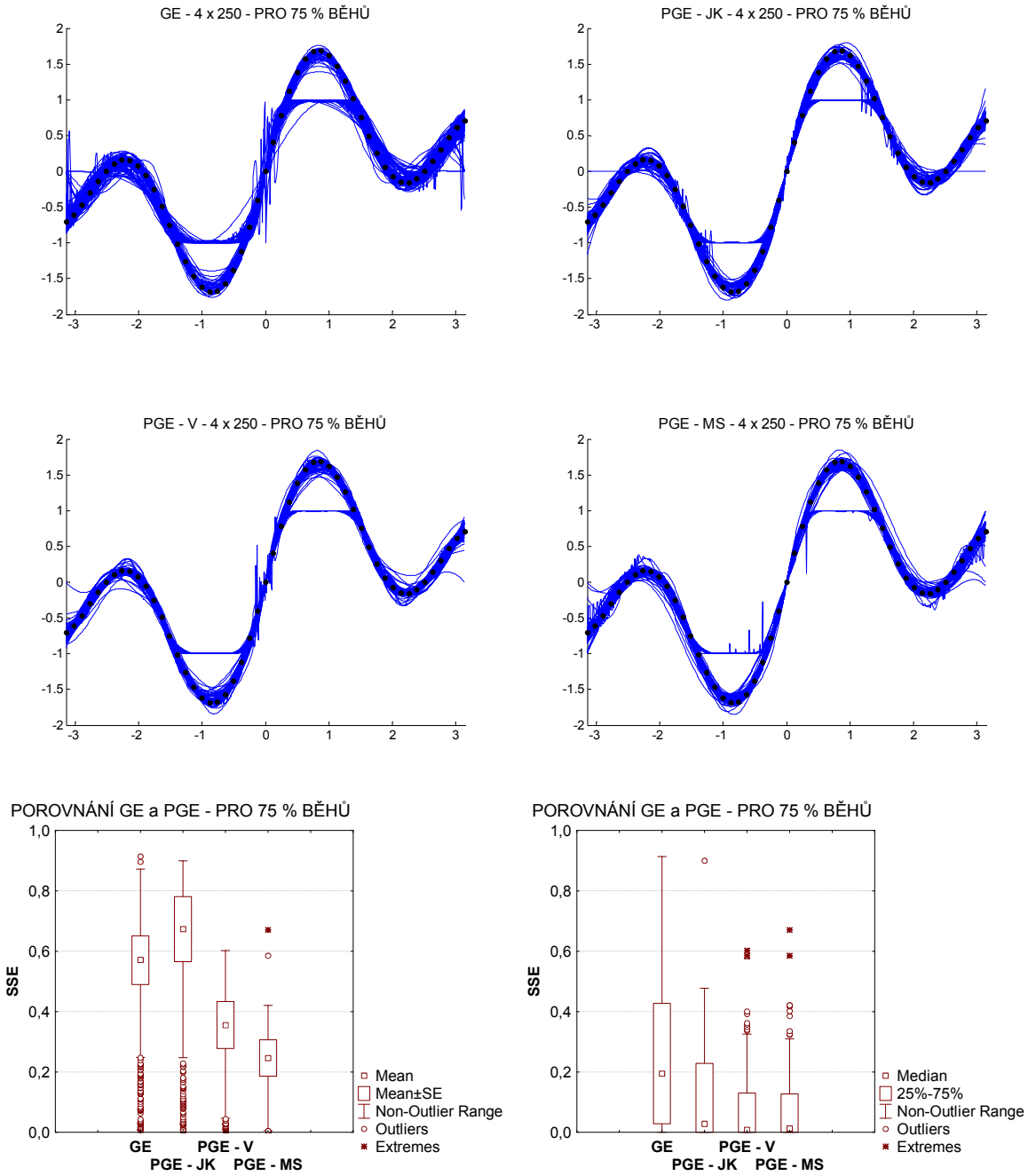


Obr. 34: Výsledky testů 9 a 10.

9.3 PGE – vliv migrační struktury



Obr. 35: Výsledky testů 11 a 12.

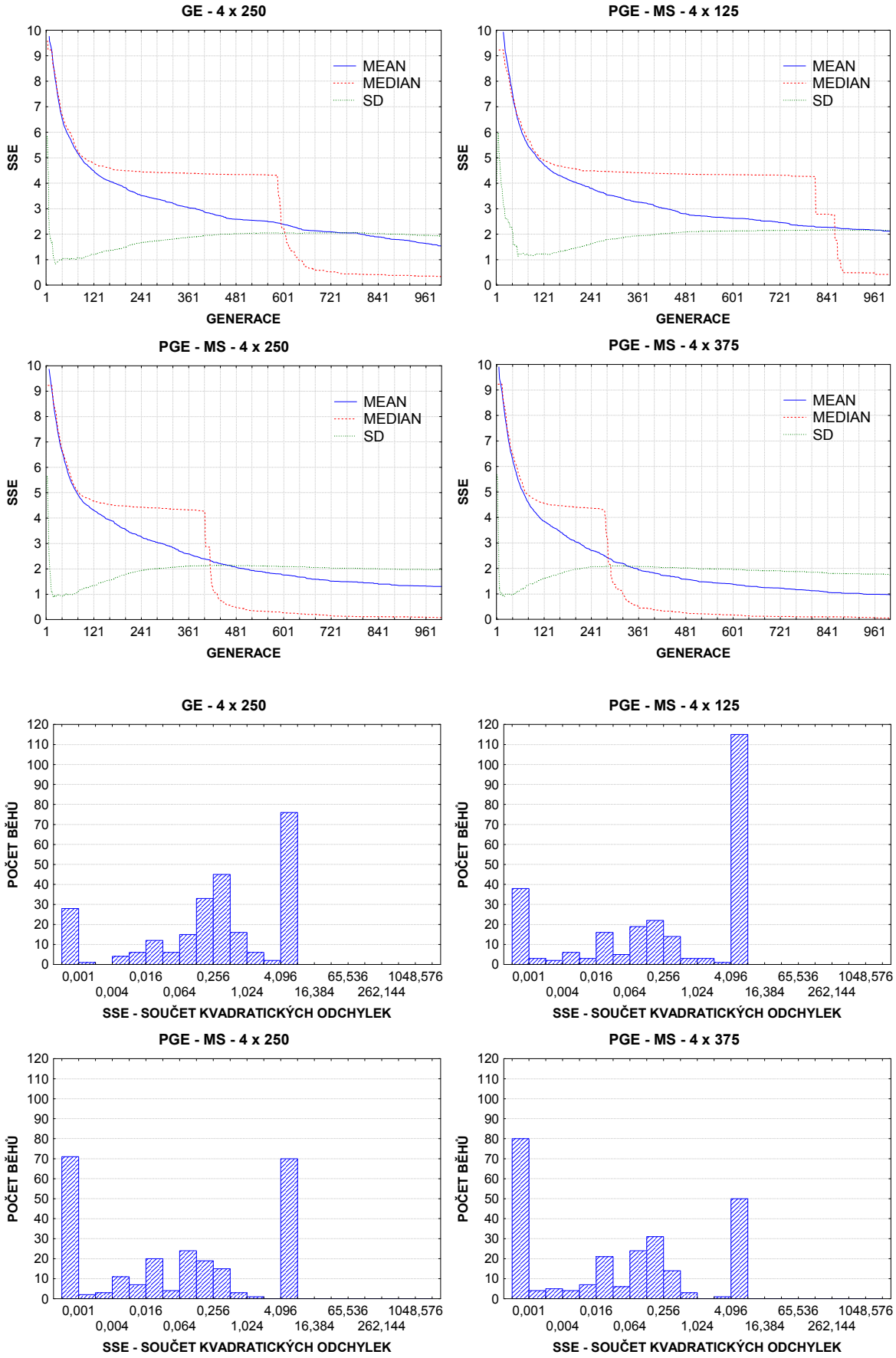


Obr. 36: Výsledky testu 13.

Tab. 11: Tabelované výsledky testu 13.

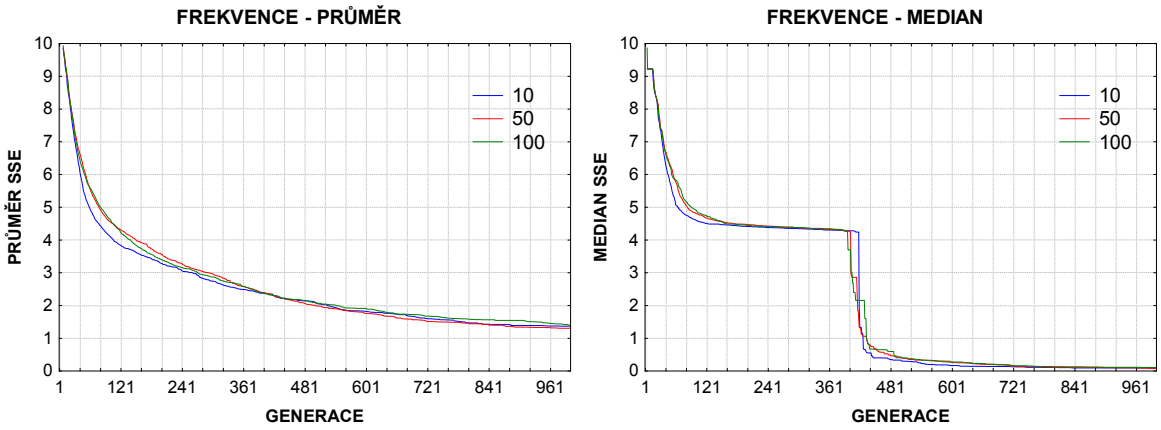
	GE	PGE - JK	PGE - V	PGE - MS
MEAN	0,570431167	0,6732195130	0,35563157200	0,2462534100
MEDIAN	0,194060267	0,0287681775	0,00824195822	0,0125012285
SD	1,107034220	1,4751763600	1,06109219000	0,8224810260

9.4 PGE – vliv velikosti populací

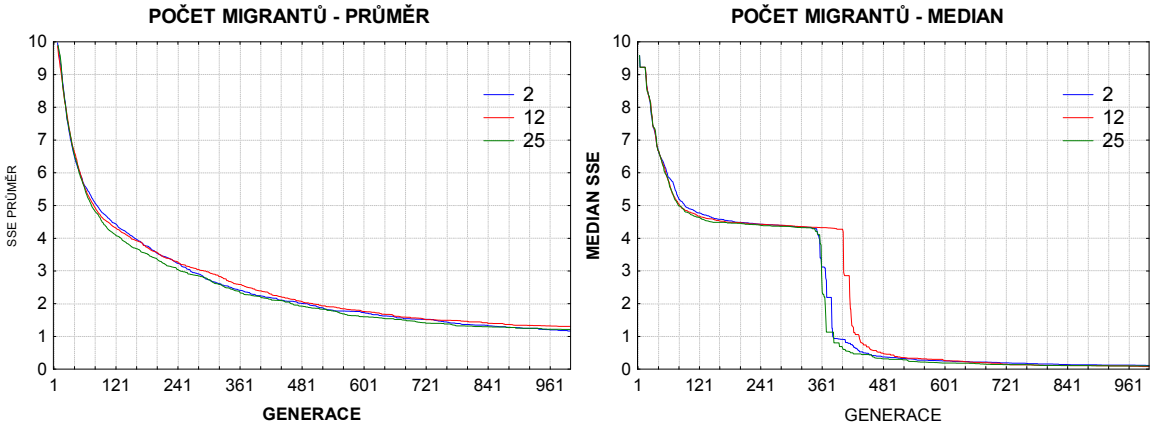


Obr. 37: Výsledky testů 14 a 15.

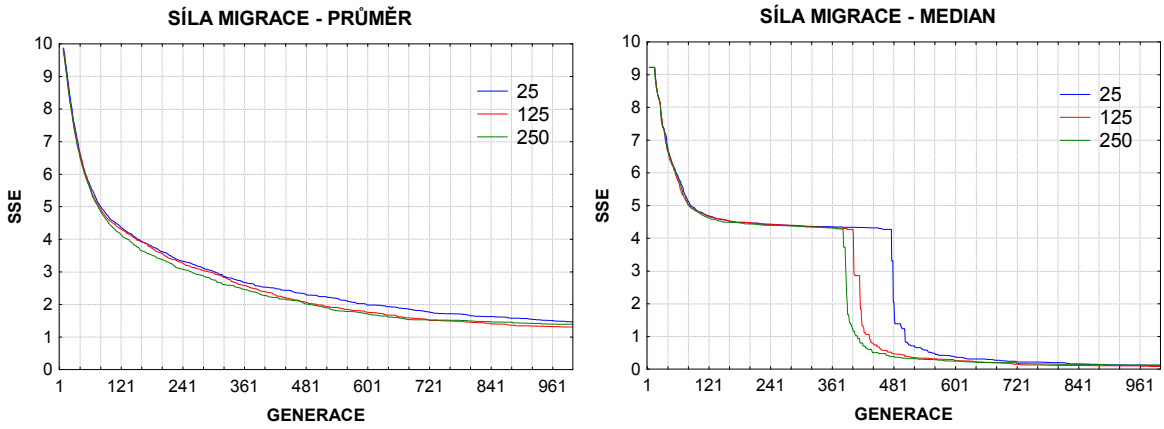
9.5 PGE – vliv změny parametrů



Obr. 38: Výsledky testu 16.



Obr. 39: Výsledky testu 17.



Obr. 40: Výsledky testu 18.

10 Aplikace na reálných problémech

10.1 Model výnosu úrody cibule [11]

Pro obecný vztah mezi výnosem pěstované plodiny y a hustotou sadby zeleniny x byly navrženy následující empirické regresní modely:

Model A (Bleasdale a Nelder, 1960): $y = (\beta_1 + \beta_2 x)^{-\frac{1}{\beta_3}}$

Model B (Holliday, 1960): $y = \frac{1}{\beta_1+\beta_2 x+\beta_3 x^2}$

Model C (Farazdaghi a Harris, 1968): $y = \frac{1}{\beta_1+\beta_2 x^{\beta_3}}$

Model D (asymptotický, Mead, 1979): $y = \frac{1}{\beta_1+\beta_2 x}$

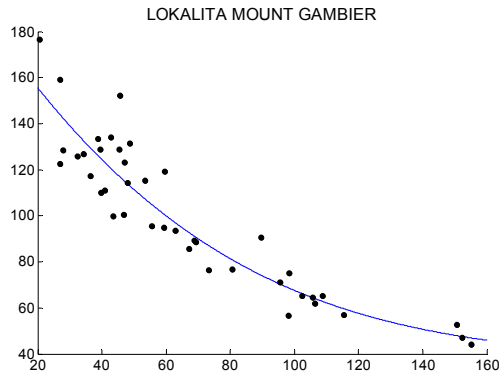
kde biologický výklad parametrů říká, že β_1 je mírou genetického potenciálu plodiny, nepotlačené vlivem konkurenčního plevele, zatímco β_2 je mírou potenciálu konkurenčního plevele. Následující naměřená data jsou ze dvou lokalit jižní Austrálie, data jsou v posloupnosti hustota sazenic na m^2 [počet sazenic/m^2] a výnos cibule [g/sazenici].

Lokalita Mount Gambier:

20,64	176,58	26,91	159,07	26,91	122,41	28,02	128,32	32,44	125,77	34,28	126,81
...	...	...	...	...	...	...	...	...	...	...	...
108,75	65,19	115,38	57,10	150,77	52,68	152,24	47,01	155,19	44,28		

Lokalita Uraidla:

22,30	148,57	25,86	125,30	29,09	150,69	29,74	147,42	31,68	117,10	31,68	116,64
...	...	...	...	...	...	...	...	...	...	...	...
119,92	61,60	120,89	26,30	126,71	61,20	138,99	41,60	146,75	45,20	160,97	46,40



Gramatika: T = {+, -, *, X, konstanty}

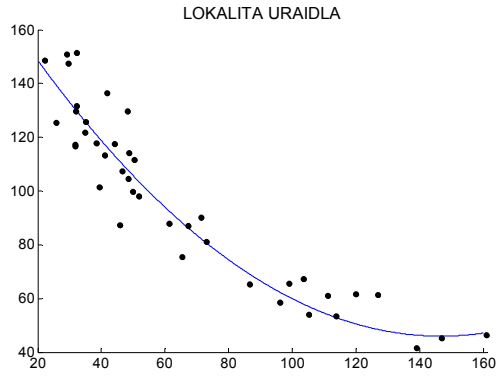
SSE: 5892,8496

Nalezená funkce:

Y = ((7.0) + ((13.0) .* (12.0))) - ((X) - (((2.0666666701436043) - ((((X) + ((((X) - (8.0)) - (((X) .* (0.0196078431372549)) .* ((((X) - (14.0)) - (0.8274509803921568)) .* (0.5843137254901961)))) .* (0.18823529411764706))) - (4.533333361148834)) .* (0.047058823529411764))) .* ((0.21176470588235294) .* ((((((X) .* (0.00784313725490196)) .* (((X) + (0.20392156862745098)) .* (0.17647058823529413))) + ((11.600000023841858) .* (14.0))) - ((X) + (X))) .* (0.39215686274509803)))))

Upravená nalezená funkce:

Y = $1.6652 \times 10^{-8}(-1423.39 + X)(-320.054 + X)(-320.054 + X)(36638.3 - 267.146\,X + X^2)$



Gramatika: T = {+, -, *, X, konstanty}

SSE: 4847,7373

Nalezená funkce:

Y = ((0.6901960784313725) .* (((14.600000023841858) .* (13.933333337306976)) - (X))) - (((((2.266666680574417) .* (15.600000023841858)) + (14.466666668653488)) - (X)) .* ((((0.06274509803921569) .* ((0.050980392156862744) .* ((((X) - (0.26666666666666666)) - (16.0)) + (X)))) - (0.8156862745098039)))

Upravená nalezená funkce:

Y = $183.64 - 1.87668\,X + 0.00639754\,X^2$

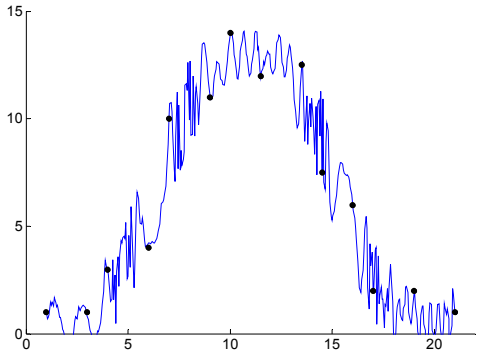
Obr. 41: Aproximace dat výnosu úrody cibule ze dvou lokalit jižní Austrálie.

10.2 Aproximace píku [11]

Úkolem je aproximovat pík, který je zadaný diskrétními hodnotami dle následující tabulky. Pro aproximaci dat jsem jako účelovou funkci zvolil součet kvadratických odchylek a použil jsem tři různé gramatiky, výsledky jsou zobrazeny na následujícím obrázku.

Tab. 12: Data pro aproximaci píku.

x	$f(x)$
1	1
3	1
4	3
6	4
7	10
9	11
10	14
11,5	12
13,5	12,5
14,5	7,5
16	6
17	2
19	2
21	1

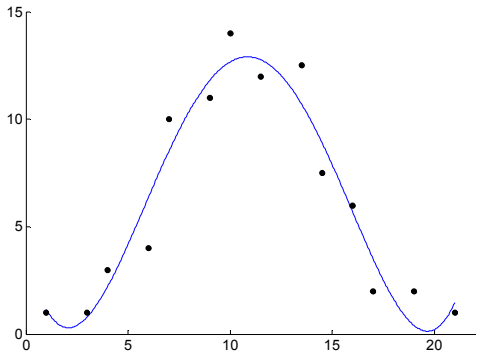


Gramatika: T = {+, -, *, /, sin, cos, X, konstanty}

SSE: 0,15772063

Nalezená funkce:

Y = (6.733333349227905) - (((7.600000023841858) + ((sin ((((cos (sin ((X) .* (X)))) ./ (5.600000023841858)) + ((X) + (5.333333343267441))) .* (((X) + (3.333333343267441)) ./ (2.9333333373069763)))) ./ (0.8784313725490196))) .* (sin (sin ((((X) + (sin (sin ((((8.133333340287209) + ((cos (((15.466666668653488) ./ (X)) .* (sin (sin (((X) + (cos ((sin (X)) ./ (6.3333333343267441)))) .* (7.0)))))) ./ (2.800000011920929))) ./ (sin (cos (X)))) - (4.933333337306976))))) + (2.6666666865348816)) ./ (2.8666666746139526)))));

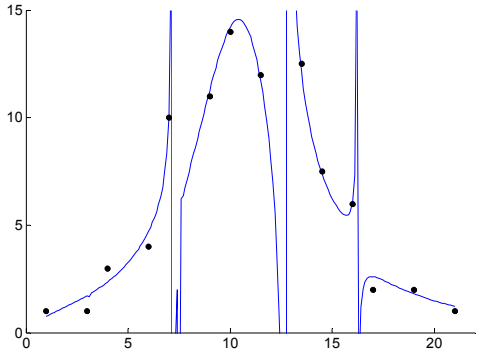


Gramatika: T = {+, -, *}

SSE: 21,873652

Nalezená funkce:

Y = (((((X) + (0.6549019607843137)) - (((0.23921568627450981) .* ((0.27450980392156865) + ((0.3058823529411765) .* (((X) + ((0.23921568627450981) + (X))) .* (((0.12941176470588237) .* ((14.0) - (X))) .* ((14.0) - (((0.6078431372549019) + (((0.12941176470588237) - ((X) - (X)) + (0.5803921568627451))) + ((X) + (X)))) .* (0.396078431372549))))) + ((0.03137254901960784) .* (((X) + (X)) + (((0.12549019607843137) .* ((10.0) - (X))) + (X))) + (((0.12549019607843137) .* ((4.0) - (X))) + (7.0)))))))) - (1.0))) + (0.8549019607843137)) .* ((0.1411764705882353) .* ((12.0) - (X)))) + (0.4745098039215686)) + (X);



Gramatika: T = {+, -, *, /}

SSE: 2,280012

Nalezená funkce:

Y = ((X) .* (5.0)) ./ ((4.0) + (((X) - (11.0)) .* (((X) - (9.0)) ./ ((3.0) .* ((10.0) ./ ((X) - (((X) - ((X) ./ ((((X) - (12.0)) + ((4.0) + ((((X) - (10.0)) ./ (((X) - (8.0)) ./ (12.0))) ./ (((X) - ((14.0) ./ ((X) .* (X)))) + (((X) - ((X) ./ (7.0))) - (X)))))) - (9.0)))) ./ (((X) ./ (X)) + ((4.0) .* (((X) - (7.0)) .* (((X) - (13.0)) ./ (X))))))))));

Obr. 42: Aproximace píku, tří možná řešení pro tři různé gramatiky.

10.3 Syntéza úplné binární sčítačky

Binární sčítačka je kombinační logický obvod realizující sčítání čísel reprezentovaných v binární číselné soustavě. Tvoří důležitou součást aritmeticko-logické jednotky (ALU) procesoru (CPU) počítače. Rozlišujeme poloviční a úplnou sčítačku, přičemž poloviční sčítačka realizuje sčítání dvou jednomístných binárních čísel a jejím výstupem jsou dva jednobitové sčítance, respektive jednobitový součet a jednobitový příznak přenosu do vyššího řádu. Úplná sčítačka realizuje sčítání dvou jednomístných binárních čísel s připočítáním přenosu z předcházejícího řádu, vstupem jsou tři jednobitové sčítance A, B, C_i, výstupem je jednobitový součet S a jednobitový příznak přenosu do vyššího řádu C_o.

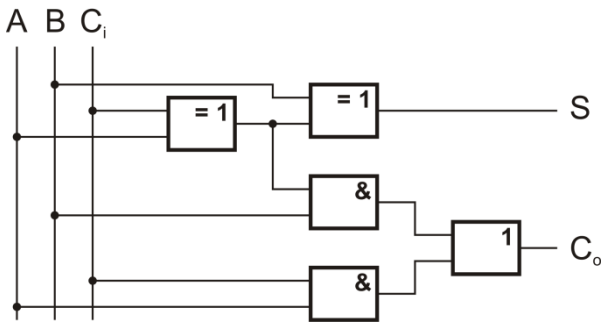
Úplnou sčítačku je možné složit z dvou poloviční sčítaček a hradla OR. Hradlo OR je možné bez vlivu na funkčnost nahradit hradlem XOR. Na vytvoření úplné sčítačka tak stačí dva typy hradel, což je praktické pro jejich realizaci. Úkolem je pomocí gramatické evoluce navrhnout logický obvod pro úplnou sčítačku, jejíž pravdivostní tabulka je dána následující tabulkou.

Tab. 13: Pravdivostní tabulka pro úlohu syntéza úplné binární sčítačky.

A	B	**C_i**	**S**	**C_o**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Aplikací gramatické evoluce na úlohu pro syntézu úplné binární sčítačky byl získán následující program, s rozdílem, že na místo názvů vstupních a výstupních proměnných X1, X2, X3, Y1 a Y2 jsem pro zřejmý význam použil názvy A, B, C_i, S a C_o. Blokové schéma programu je na následujícím obrázku.

C_o = (B & ((Xor[A, C_i])) | (C_i & A)
S = Xor[(Xor[A, C_i]), B]



Obr. 43: Blokové schéma úplné sčítačky dle návrhu gramatické evoluce.

11 Závěr

V praktické části jsem realizoval aplikaci včetně grafického uživatelského rozhraní pro tzv. gramatickou evoluci pro účely optimalizace, respektive pro automatické generování počítačových programů popsaných Backus-Naurovou formou. Implementoval jsem dvě úlohy, tzv. aproximaci funkcí, neboli prokládání naměřených dat vhodnou křivkou, a tzv. syntézu logických obvodů, která se zabývá návrhem logického obvodu, jestliže známe jeho chování.

Aplikace těží z výhod technologie Java v kontextu využití více procesorů prostřednictvím svých vláken a její realizace spočívala ve vytvoření tříd pro reprezentaci populací jedinců, které zajišťují aplikaci operátorů gramatické evoluce a také mapování z genotypu do fenotypu, dále třídy, které reprezentují evoluční proces, pracují s populacemi, vyčíslují a hodnotí generované programy, nakonec třídy, které tyto procesy spouští, vyhodnocují a řídí v čase. K použití programovacího jazyku lze závěrem říci, že je vhodnou volbou i pro tak výpočetně náročné aplikace jako je gramatická evoluce. Pro vyladění byl použit profiler za účelem vyladění algoritmu tak, aby byl co možná nejrychlejší, ale nelze říci, že jej již nelze urychlit.

Diplomová práce se zabývala problematikou gramatické evoluce, byl v ní rozebrán princip, algoritmus a operátory, jako jsou selekce, křížení a mutace, kde jsem popsal a také v praktické části implementoval pokrokové operátory strukturální křížení a strukturální mutaci. Dále jsem popsal některé možné struktury migrační paralelní gramatické evoluce a v aplikaci jsem implementoval čtyři následující struktury: jednosměrná kruhová, obousměrná kruhová, vzájemná a struktura s master populací.

Dále jsem gramatickou evoluci testoval na problému aproximace funkcí na dvou zvolených úlohách, data první byly generovány polynomiální funkcí a data druhé trigonometrickou funkcí. Sledoval jsem, jak si gramatická evoluce vede při aproximaci vygenerovaných dat a také vliv hodnotícího kritéria, kde jsem testoval tři účelové funkce: dynamické toleranční pásmo (epsilonová trubice), součet kvadratických odchylek a součet absolutních odchylek. K výsledkům lze říci, že pro danou polynomiální úlohu se účelová funkce dynamické toleranční pásmo ukázala jako nejvýhodnější, naproti tomu výsledky dané trigonometrické úlohy byly pro všechny tři účelové funkce více či méně vyrovnané a z pohledu počtu nejlepších řešení se jako nejvýhodnější ukázala účelová funkce součet absolutních odchylek. Dále jsem testoval přínos struktur migrační paralelní gramatické evoluce, k získaným výsledkům lze říci, že má určitý pozitivní přínos.

Rád bych ještě závěrem podotknul, že práce se mi líbila a rozhodně má smysl se touto problematikou i nadále zabývat a rozvíjet ji, a také vylepšovat a doplňovat aplikaci.

LITERATURA

[1] O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Springer, ISBN-13: 978-1402074448, 2003

[2] Vladimír Mařík, Olga Štepánková, Jiří Lažanský a kolektiv: Umělá inteligence (3), Academia, ISBN: 80-200-0472-6, 2001

[3] Vladimír Mařík, Olga Štepánková, Jiří Lažanský a kolektiv: Umělá inteligence (4), Academia, ISBN: 80-200-1044-0, 2003

[4] Ivan Zelinka, Zuzana Oplatková, Miloš Šeda, Pavel Ošmera, František Včelař: Evoluční výpočetní techniky – principy a aplikace, BEN, ISBN: 80-7300-218-3, 2009

[5] Jan Jelínek, Vladimír Zicháček: Biologie pro gymnázia, nakladatelství Olomouc, ISBN: 80-7182-070-9, 1999

[6] Brett Spell: Java Programujeme profesionálně, Computer Press, ISBN: 80-7226-667-5, 2002

[7] Pavel Herout: Učebnice jazyka JAVA, Kopp, ISBN: 80-7232-115-3, 2000

[8] Eubanks Brian: Java na maximum, Computer Press, ISBN: 80-251-1111-3, 2006

[9] Ivan Švarc, Miloš Šeda, Miluše Vítečková: Automatické řízení, CERM, ISBN: 978-80-214-3491-2

[10] Matoušek Radomil: Fitness Measure in GE: Epsilon Tube in Sybolic Regression Task, In procceeding of the 15th MENDEL soft-computing conference. Brno, Czech Republic, 2008, ISBN: 978-80-214-3884-2

[11] Meloun, M., Militky, J.: Kompendium statistickeho zpracovani dat, Academia Praha, CR, 2002, ISBN 80-200-1008-4

[12] Wikipedia - Otevřená encyklopedie, http://cs.wikipedia.org/

SEZNAM PŘÍLOH

vlastní text diplomové práce
aplikace – Java archív JAR
uživatelská dokumentace
vstupní data řešených úloh