Univerzita Pardubice
Fakulta ekonomicko-správní

Návrh a realizace systému pro genetické programování

Bc. Petr Sotona

Diplomová práce

2009

Univerzita Pardubice
Fakulta ekonomicko-správní
Ústav systémového inženýrství a informatiky
Akademický rok: 2008/2009

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Petr SOTONA**
Studijní program:	**N6209 Systémové inženýrství a informatika**
Studijní obor:	**Informatika ve veřejné správě**
Název tématu:	**Návrh a realizace systému pro genetické programování**

Zásady pro vypracování:

1. Základní pojmy z oblasti genetického programování
2. Principy genetických algoritmů používaných v genetickém programování
3. Návrh, tvorba, ověření programu pro genetické programování

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

WEISE, Thomas. Global Optimization Algorithms: Theory and Application [online]. 2008 [cit. 2008-05-10]. Dostupný z www: http://www.it-weise.de/projects/book.pdf.

KOZA, John R., et al. Genetic Programming IV: Routine Human-Competitive Machine Intelligence. New York : Springer Science+Business Media, 2003. 606 s. ISBN 0-387-25067-0

KOZA, John R., et al. Genetic Programming III: Darwinian Invention and Problem Solving. San Francisco : Morgan Kaufmann Publishers, 1999. 1154 s. ISBN 1-55860-543-6.

Vedoucí diplomové práce: **doc. Ing. Pavel Petr, Ph.D.**
Ústav systémového inženýrství a informatiky

Datum zadání diplomové práce: **6. října 2008**

Termín odevzdání diplomové práce: **1. května 2009**

L.S.

doc. Ing. Renáta Myšková, Ph.D.
děkanka

doc. Ing. Jiří Křupka, Ph.D.
vedoucí ústavu

V Pardubicích dne 6. října 2008

Prohlašuji:

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

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

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 30. 4. 2009

Petr Sotona

PODĚKOVÁNÍ

Rád bych na tomto místě poděkoval panu doc. Ing. Pavlovi Petrovi, Ph.D., vedoucímu diplomové práce, za odbornou pomoc, vedení, podporu při vypracovávání a cenné připomínky, jak k obsahové a formální stránce diplomové práce, tak i k samotnému programu.

Také bych chtěl poděkovat svým rodičům za jejich celoživotní podporu.

ANOTACE

Diplomová práce se zabývá tvorbou softwarového nástroje na řešení úloh symbolické regrese prostřednictvím techniky genetického programování. Předmětem úloh je na základě vstupní datové matice naleznout funkci v symbolické podobě, která daná data co nejlépe proloží. Softwarový nástroj je implementován prostřednictvím programovacího jazyka C++ s využitím knihovny grafického rozhrani Qt 4.4.

KLÍČOVÁ SLOVA

Genetický algoritmus; genetické programování; symbolická regrese.

TITLE

Design and Implementation of Genetic Programming System.

ANNOTATION

Main objective is to describe genetic programming method and design and implement software tool which is able to solve symbolic regression problem through genetic programming. Software tool is created in C++ programming language with Qt 4.4 GUI library.

KEYWORDS

Genetic algorithm; genetic programming; symbolic regression.

Obsah

Úvod....11
1 Princip genetického algoritmu....13
1.1 Evoluční algoritmy....13
1.2 Výzkum evolučních algoritmů....14
1.3 Základní pojmy....16
1.4 Jednoduchý genetický algoritmus....17
1.4.1 Reprezentace jedinců....19
1.4.2 Funkce vhodnosti....21
1.4.3 Operátor selekce....21
1.4.4 Operátor křížení....27
1.4.5 Operátor mutace....29
1.4.6 Operátor inverze....30
1.5 Věta o schématech....30
1.6 Možnosti využití genetického algoritmu....35
2 Principy genetických algoritmů používaných v genetickém programování....36
2.1 Reprezentace jedinců....36
2.2 Počáteční populace....39
2.3 Funkce vhodnosti....40
2.4 Operátor křížení....42
2.5 Operátor mutace....44
2.6 Operátor permutace....45
2.7 Operátor editace....47
2.8 Operátor zapouzdření....48
2.9 Operátor decimace....49
2.10 Ukončovací kritérium....50
2.11 Doporučené hodnoty parametrů, operátory a metody....50
2.12 Možnosti využití genetického programování....52
2.13 Příklad umělého mravence....53
3 Návrh, tvorba, ověření programu pro genetické programování....58
3.1 Stanovený cíl a nezbytné požadavky....58

3.2 Návrh technologického řešení.....59
3.3 Návrh algoritmu genetického programování pro symbolickou regresi.....59
3.4 Návrh rozhraní.....65
3.5 Návrh objektového modelu.....66
3.6 Implementace.....69
3.7 Testování programu na vzorových příkladech symbolické regrese.....70
3.7.1 První testovací příklad.....71
3.7.2 Druhý testovací příklad.....77
3.7.3 Průběh řešení druhého příkladu.....77
3.7.4 Závěr testování.....79
3.8 Návrhy na rozšíření programu.....80
Závěr.....81
Použitá literatura.....83
Seznam příloh.....86

Seznam obrázků

Obrázek 1 - Operátor křížení. 28
Obrázek 2 - Vícebodové křížení....28
Obrázek 3 - Operátor mutace....29
Obrázek 4 - Operátor inverze....30
Obrázek 5 - Příklad syntaktického stromu....37
Obrázek 6 - Operátor jednobodového křížení: náhodný výběr uzlů křížení u kopií rodičů....43
Obrázek 7 - Operátor jednobodového křížení: výslední potomci....43
Obrázek 8 - Operátor mutace: výběr uzlu....44
Obrázek 9 - Operátor mutace: nahrazení uzlu novou větví....45
Obrázek 10 - Operátor permutace: výběr uzlu....45
Obrázek 11 - Operátor permutace: prohození argumentů funkce....46
Obrázek 12 - Operátor editace: zjednodušování stromu....47
Obrázek 13 - Operátor editace: výsledný zjednodušený strom....47
Obrázek 14 - Operátor zapouzdření: výběr funkce....48
Obrázek 15 - Operátor zapouzdření: funkce je zapouzdřena do terminálu E0....49
Obrázek 16 - Pohyb mravence "prošívače"....55
Obrázek 17 - Syntaktický strom optimálního řešení. 56
Obrázek 18 - Vývojový diagram zachycující základní algoritmus programu....64
Obrázek 19 - Diagram tříd objektového modelu....67
Obrázek 1 - Záložka genetický algoritmus....89
Obrázek 2 - Záložka funkce jedince....93
Obrázek 3 - Záložka strom jedince....94
Obrázek 4 - Záložka záznam průběhu algoritmu....95
Obrázek 5 - Záložka data....96
Obrázek 6: Dialog pro načtení datové matice....97

Seznam obrázků

Graf 1- Rozložení pravděpodobností při selekci přímo úměrné vhodnosti.....24
Graf 2 - Ruletová selekce přímo úměrná pořadí.....26
Graf 3 - Průběh hledání optimálního řešení při použití všech možných funkcí a terminálů.....73
Graf 4 - Průběh hledání nejlepšího řešení při optimálním nastavením parametrů. 75
Graf 5 - Průběh hledání optimálního řešení při měření velikosti proudů.....79

Seznam tabulek

Tabulka 1- Příklad rozdílu standardního a Grayova kódování.....20
Tabulka 2 - Pravděpodobnosti jedinců při selekci přímo úměrně vhodnosti.....24
Tabulka 3 - Pravděpodobnosti jedinců při selekci přímo úměrné pořadí.....26
Tabulka 4 - Vstupní datová matice naměřených hodnot.....71
Tabulka 5 - Výsledná datová matice včetně odhadu R (Rozdíl představuje chybu odhadu).....74
Tabulka 6 - Výsledná datová matice včetně odhadu R při optimálním nastavení parametrů.....76
Tabulka 7 - Vstupní datová matice naměřených hodnot druhého příkladu.....77

Úvod

Genetické programování je poměrně novou metodou umělé inteligence, kterou poprvé zveřejnil John Koza v 90. letech 20. století a díky své obecnosti a širokým možnostem svého využití se velice rychle stala oblíbenou metodou nejen na akademické půdě. Genetické programování je uplatňováno zejména při řešení optimalizačních problémů a automatickém programování. Genetické programování rozšiřuje možnosti genetického algoritmu, který byl navržen v 60. letech 20. století Johnem Hollandem na univerzitě v Michiganu. Genetický algoritmus je evolučním algoritmem, který na základě počátečních potenciálních řešení a aplikací genetických operátorů generuje řešení nová, lepší a postupně se přibližuje k řešení optimálnímu.

Genetické programování na rozdíl od genetického algoritmu potenciální řešení reprezentuje rozdílnou strukturou, čímž je reprezentace složitých potenciálních řešení mnohem snazší a přímočařejší a také díky své dynamické struktuře umožňuje řešit problémy, které jsou jinak jednoduchým genetickým algoritmem řešitelné velice obtížně.

Cílem diplomové práce je přiblížit problematiku genetického programování a vytvořit vhodný programový nástroj, který je schopen prostřednictvím genetického programování řešit složité úlohy symbolické regrese.

Symbolická regrese patří mezi základní aplikace genetického programování, kdy na základě zjištěných dat se pokoušíme nalézt funkci, která by nám daná data co nejlépe proložila, přičemž nemáme žádnou představu o tom, jakého tvaru by měla funkce nabývat a jakých hodnot by měly nabývat její numerické parametry.

Před samotným návrhem a implementací je v první kapitole přiblížen Hollandův jednoduchý genetický algoritmus. Genetické programování vychází z genetického algoritmu a je jeho rozšířením. Bez pochopení principu, jakým způsobem je genetický algoritmus realizován, jak jsou reprezentována potenciální řešení, jakým způsobem jsou realizovány

genetické operátory a jak je možné, že genetický algoritmus postupně nalézá lepší potenciální řešení daného problému, by bylo pochopení metody genetického programování velice obtížné.

V druhé části práce je přiblížena samotná metoda genetického programování ve srovnání s genetickým algoritmem. Je uvedeno, v čem se metody odlišují, jaké má genetické programování před genetickým algoritmem výhody, s kterými nedostatky se musí vypořádat a jak je možné tyto nedostatky potlačit. Genetické programování je v závěru demonstrováno na problému tzv. umělého mravence, kdy je nutné najít co nejlepší algoritmus, který by mravence navigoval prostorem, přičemž mravenec musí během omezeného času nalézt v prostoru co nejvíce návnad.

Třetí část práce se věnuje samotnému návrhu programu. Je nejprve navržen samotný genetický algoritmus, jeho objektový model, rozhraní a posléze je implementován v jazyce C++ s využitím knihovny Qt. V závěru části je výsledný program testován na příkladech symbolické regrese.

Přílohy práce obsahují uživatelskou příručku a stručnou dokumentaci k programové části, která bude moct být využita při případném rozšíření daného programu. Na přiloženém CD se pak nachází okomentovaný zdrojový kód programu včetně zkompilovaného programu pro platformu MS Windows a Linux/X.11

1 Princip genetického algoritmu

V 60. letech 20. století se řada vědců snažila sestavit evoluční algoritmus, který by byl natolik univerzální, že by prostřednictvím něho bylo možné řešit řadu složitých optimalizačních úloh. Dílčích úspěchů dosáhla řada vědců, ale dostatečně univerzální algoritmus sestavil až **John Holland**[1] se svým genetickým algoritmem**.** Ačkoliv je genetický algoritmus dnes často používán v různých modifikacích, aby bylo dosaženo jeho větší efektivnosti, princip algoritmu zůstává stejný.

Tato kapitola stručně přibližuje problematiku evolučních algoritmů a popisuje princip tzv. jednoduchého genetického algoritmu, genetického algoritmu v podobě v jaké ho navrhl John Holland.

1.1 Evoluční algoritmy

Evoluční algoritmy jsou založeny na jedné myšlence - na základě skupiny možných řešení daného problému postupně aplikací určitých operátorů získávat stále lepší a lepší řešení. Inspirací pro evoluční algoritmy je přirozený proces evoluce v přírodě, který popsal Charles Darwin. Evoluční algoritmy napodobují model evolučních procesů v přírodě a jeho napodobením ze stávajících řešení získávají postupně řešení nová, lepší.

Evoluční algoritmy mají několik základních rysů [3]:

- pracují zároveň s celou skupinou možných řešení zadaného problému,
- na počátku algoritmu vygenerují potenciální řešení,
- vygenerovaná řešení kombinují, náhodně mění a tímto způsobem generují řešení nová,
- vyřazují nevhodná řešení a opakují cyklus, dokud nenaleznou dostatečně dobré řešení.

Pod obecný pojem evoluční algoritmy se zejména zařazují genetické algoritmy, evoluční strategie, evoluční programování a genetické programování. Dnes je mnohdy obtížné rozhodnout, protože tyto oblasti se v současnosti sbližují a vzájemně prolínají. [3]

1 **John Holland** – nar. 1929 ve Fort Wayne, Indiana. V současnosti profesorem na univerzitě v Michiganu. [12]

1.2 Výzkum evolučních algoritmů

V 50. a 60. letech 20. století se několik vědců snažilo nezávisle na sobě přijít na co nejlepší techniku evolučního algoritmu, která by mohla být použita jako nástroj na řešení složitých optimalizačních problémů, při nichž jiné známé matematické metody selhávaly nebo žádné metody pro daný problém nebyly známy.

První úspěch zaznamenal v 60. letech **Ingo Rechenberg**[2], kdy zveřejnil svou metodu e**volční strategie** (Evolution strategies), metodu, kterou například úspěšně použil při hledání optimálního profilu leteckého křídla. Metoda je využitelná pro optimalizaci v případech, kdy potenciální řešení je možné numericky popsat. Metoda byla dále intenzivně rozvíjena zejména **Hans Paul Schwefelem**[3], Rechenbergovým kolegou z Technické univerzity v Berlíně. Postupně se ale ukázalo, že evoluční strategie dokáže řešit pouze určitý typ úloh, kde nahrazuje gradientovou metodu, která v případech velkého počtu suboptimálních řešení selhává. [7]

V roce 1966 **Lawrence J. Fogel**[4] rozvinul evoluční strategii do tzv. **evolučního programování** (Evolutionary programming), do techniky, v které možná řešení mohou nabývat konečných stavů (mají omezený stavový prostor) a na základě mutace a hodnotící funkce, je postupně hledáno nejlepší řešení. [7]

Postupně se evoluční strategie a evoluční programování vzdalovalo původní myšlence co nejbližšího napodobení procesu evoluce v přírodě a tak v 60. letech 20. stol. vznikly dvě komunity vědců. Jedna komunita dále rozvíjela metodu evoluční strategie, zatímco druhá se snažila přijít na lepší způsob simulace evoluce. Skupiny pak pracovaly na svých výzkumech paralelně. [3]

Druhou skupinou vědců byly později vyvinuty **genetické algoritmy**. Genetické algoritmy objevil John Holland na univerzitě v Michiganu. Původním cílem Hollandova výzkumu bylo studovat proces adaptace živočišných druhů v přírodě a najít způsob, jak tento

2 **Ingo Rechenberg** - nar.1934 v Berlíně. Od roku 1972 profesorem na Technické Univerzitě v Berlíně jako vedoucí katedry bioniky a evolučních technik. [20]

3 **Hans Paul Schwefel** – nar. 1940 v Berlíně. Od roku 1985 profesorem na Univerzitě Dortmund. [16]

4 **Lawrence J. Fogel** – 1928 - 2007 nar. v New Yorku. Jeho disertační práce z roku 1965 dala základ technice evolučního programování. Techniku uplatňoval v praxi. Členem několika vědeckých institucí. [13]

proces může být simulován na výpočetní technice. Neměl v úmyslu vyvinout metodu pro řešení optimalizačních problémů. Jeho algoritmus byl ale natolik obecný, že se ho jeho kolegové pokusili použít k řešení složitých optimalizačních problémů. Na univerzitě pak byla zformována skupina vědců z jeho kolegů a studentů a ti algoritmus dále úspěšně testovali a rozvíjeli. [3]

V roce 1975 John Holland zveřejnil svou knihu *Adaptation in Natural and Artificial Systems*, v které shrnul navržený genetický algoritmus, pomocí něhož je možné simulovat proces adaptace v přírodě. Publikací dal teoretický základ pro výpočetní simulaci adaptace.

Hollandův genetický algoritmus byl založen na přímé analogii s procesy v přírodě, které odhalil Charles Darwin. Algoritmus pracuje s populací, kde každý jedinec představuje jedno možné řešení. Každý jedinec je reprezentován prostřednictvím genotypu, který je složen z genů. Každý gen reprezentuje určitou vlastnost a nabývá omezených stavů tzv. alel. Na základě hodnotící funkce je možné jedince porovnávat a určovat, který lépe splňuje požadovaná kritéria. Pak je simulován proces evoluce, kdy se na základě hodnotící funkce vyberou jedinci, na které jsou aplikovány operátory křížení, mutace a inverze, a tím jsou vygenerováni jedinci noví, kteří mají vlastnosti svých předků. Nově vzniklí jedinci jsou pak ohodnoceni, nakolik splňují požadovaná kritéria a opět jsou na základě selekce z nich vybráni lepší jedinci a na ně uplatněny operátory, které generují další řešení. Cyklus se opakuje tak dlouho, dokud není dosaženo dostatečně dobrého řešení, či není splněna jiná ukončovací podmínka. Lepší jedinec má větší šanci, že bude vybrán, a tak každá následující generace se skládá z jedinců, kteří vznikli kombinací vlastností těch lepších jedinců z generace předcházející. Během realizace cyklu se tak populace zlepšuje a jsou generována postupně lepší řešení. [10]

Výzkum a aplikace genetických algoritmů zpočátku ležel mimo větší zájem vědecké komunity, protože optimalizace prostřednictvím genetických algoritmů byla příliš náročná na výpočetní zdroje a tehdejší výpočetní technika nebyla dostatečná. Velký rozvoj v této oblasti přišel až v 90. letech 20. století a to díky nové výpočetní technice. [3]

V 90. letech 20. století rozšířil **John Koza**[5] techniku genetických algoritmů do podoby tzv. **genetického programování**, kdy navrhl reprezentovat potenciální řešení jako programové struktury v podobě syntaktického stromu a této reprezentaci také přizpůsobil jednotlivé genetické operátory. Díky tomu genetické programování dokáže řešit některé složité úlohy mnohem jasněji, protože odpadá starost s někdy obtížným převodem reprezentace potenciálního řešení do podoby řetězce, kterým je genotyp reprezentován v jednoduchém genetickém algoritmu. [2]

1.3 Základní pojmy

Před přiblížením problematiky genetického algoritmu je vhodné ujasnit pojmy, které jsou v práci použity. Některé pojmy jsou v práci ještě dále přiblíženy v následujících kapitolách.

Alela - hodnota genu. [10]

Chromozom - posloupnost genů. Zpravidla realizován jako posloupnost bitů nebo vektor reálných čísel. Může být definován s pevně danou délkou nebo jeho délka může být variabilní. [10]

Fenotyp - potenciální řešení. [10]

Funkce vhodnosti - funkce, která ohodnotí každé individuum v populaci a přiřadí mu jeho vhodnost. [10]

Gen – rozlišitelná jednotka genotypu, která reprezentuje některou vlastnost fenotypu. [10]

Genetické operátory – množina operací, které jsou uplatněny na jedince v určitém pořadí. Smyslem genetických operátorů je prohledávat prostor potenciálních řešení. [10]

Genotyp - reprezentace fenotypu ve vhodné podobě. [10]

Jedinec - potenciální řešení zadaného problému. Je reprezentován svým genotypem. [10]

Křížení – genetický operátor, který produkuje nové genotypy jedinců, které jsou složeny z částí genotypů rodičů. [3]

5 **John R. Koza –** v současnosti profesor na Univerzitě ve Stanfordu. V 60. letech studoval na univerzitě v Michiganu informatiku. Studium tam dovršil v roce 1972 získáním titulu PhD. Autor řady publikací. [16]

Mutace – genetický operátor, který náhodně mění genotyp. [3]

Populace - množina jedinců. [10]

Potomek - jedinec, jenž je výsledkem křížení. [3]

Prostor potenciálních řešení – množina všech genotypů. [10]

Reprodukce – výběr jednoho jedince ze stávající generace na základě selekce a jeho vložení do generace následující, přičemž genotyp jedince zůstává nezměněn. [3]

Rodič - jedinec, který byl vybrán selekcí a jsou z něho na základě operátoru křížení odvozena řešení nová. [3]

Selekce - výběr jedince úměrně jeho vhodnosti. [3]

Vhodnost – vyjadřuje, jak moc jedinec splňuje na něho kladená kritéria. [10]

1.4 Jednoduchý genetický algoritmus

V současnosti neexistuje přesná definice genetického algoritmu, která by byla akceptovatelná celou vědeckou komunitou. Neexistuje tak přesná hranice mezi tím, co je možné považovat za genetický algoritmus a co už za genetický algoritmus považovat nelze. Obecně lze říci, že genetický algoritmus je evolučním algoritmem, který prostřednictvím operátorů selekce, křížení a mutace získává další potenciální řešení. [3]

Genetický algoritmus napodobuje proces evoluce v přírodě, kdy většina jedinců každé následující generace má vlastnosti, které jsou kombinací vlastností lepších jedinců generace předchozí. Populace jedinců jednoho druhu se v přírodě každou generací vyvíjí a každou generací se zlepšuje. Vývoj je zajištěn tím způsobem, že nevhodní jedinci, kteří se nedovedou přizpůsobit okolním podmínkám, vymírají, aniž by zanechali potomky, kteří by dále šířili jejich genetický materiál. Při páření jsou také upřednostňováni lepší jedinci, a tak lepší jedinci mají větší šanci, že předají svůj lepší genetický materiál více potomkům než jedinci slabší, kteří možná ani nedostanou šanci se páření účastnit. Jedinci následující generace tak budou mít ve výsledku lepší vlastnosti než jedinci generace předchozí. Každou další generací se populace adaptuje na okolní podmínky a je tak blíže k tomu, aby jedinci měli vlastnosti optimální.

Pro praktické účely simulace Holland navrhl reprezentovat vlastnosti jedince prostřednictvím chromozomů a samotný proces postupné adaptace simuloval prostřednictvím 4 operátorů, které jsou použity pro tvorbu jedinců každé následující generace.

Jedná se o následující operátory [10]:

- operátor selekce;
- operátor křížení;
- operátor mutace;
- operátor inverze.

Čtvrtý Hollandův operátor inverze se v současnosti uplatňuje v genetickém algoritmu již zřídka, protože se ukázalo, že na kvalitu výsledné populace a průběh algoritmu nemá výrazný vliv. [2]

Algoritmus je v současnosti používán v různých modifikacích a dle charakteru řešeného problému jsou do algoritmu implementovány i další genetické operátory, které jsou definovány za účelem zvýšení efektivnosti algoritmu. Základní tvar algoritmu ale zůstává stejný. Je založen na evolučním principu, kdy na počátku algoritmu je vygenerována počáteční generace a postupnou aplikací genetických operátorů v každé generaci vznikají jedinci noví, přičemž do každé následující generace postupují na základě operátoru selekce lepší jedinci, kteří předávají své lepší geny dále a tak je populace každou generací zlepšována. Genetický algoritmus, který obsahuje pouze genetické operátory, které navrhl Holland, se dnes označuje jako jednoduchý genetický algoritmus.

Samotný genetický algoritmus se skládá z 6 základních kroků, v kterých jsou genetické operátory uplatněny [2]:

1. *Vygeneruj počáteční populaci o N jedincích.*
2. *Vypočti vhodnost každého jedince.*
3. *Opakuj, dokud nevytvoříš N nových jedinců:*
   1) *Použij operátor selekce a vyber dva jedince.*
   2) *S určitou pravděpodobností p_c aplikuj na vybraný pár operátor křížení a získej tak nové dva jedince. Pokud operátor není aplikován, vytvoř kopie jedinců a ty považuj za nové dva jedince.*
   3) *S určitou pravděpodobností p_m aplikuj na každého jedince z dvojice nových jedinců operátor mutace.*
   4) *Výsledné dva jedince vlož do nové populace.*
4. *Současnou populaci nahraď nově získanými jedinci (novou generací).*
5. *Vypočti vhodnost každého jedince v populaci.*
6. *Pokud není splněna ukončovací podmínka, vrať se ke kroku 3. Pokud podmínka je splněna zjisti nejlepšího jedince a oznam konec.*

Bližší průběh algoritmu bude upřesněn v následujících kapitolách v souvislosti s popisem jednotlivých prvků algoritmu.

1.4.1 Reprezentace jedinců

Holland navrhl reprezentovat jedince v genetickém algoritmu prostřednictvím jeho vlastností – tzv. genů. Každý **gen** vyjadřuje určitou vlastnost jedince. Vlastnosti nabývají omezeného počtu stavů, hodnot – tzv. **alel**. Geny jedince jsou uspořádány do posloupnosti, kde každý gen má přesně určené jednoznačné pořadí v rámci posloupnosti. Tuto uspořádanou množinu genů jednoho jedince nazýváme **chromozom**. Chromozom pak může zastupovat **genotyp** jedince (v případě, že genotyp je reprezentován jedním chromozomem). Genotyp pak reprezentuje **fenotyp** jedince, což je množina všech výsledných vlastností jedince. [2]

Chromozom je nejčastěji reprezentován binárním řetězcem, kde alela í-tého genu je určena í-tou hodnotou v binárním řetězci. Pokud gen reprezentuje vlastnost, která se dá numericky vyjádřit, je vhodné numerickou hodnotu převést do její binární podoby.

V souvislosti s binárním kódováním numerických hodnot se objevuje jedna závažná skutečnost, která negativně ovlivňuje chování genetického algoritmu. Předpokladem úspěšnosti genetických operátorů je, že nepatrná změna v chromozomu se projeví jenom nepatrnou změnou výsledných vlastností. Předpoklad je ale v případě kódování čísel běžným binárním kódem narušen.

Např. dle [2] je zcela zřetelný rozdíl patrný v případě dvou reálných čísel x_1= -0,000048 a x_2 = 0,000048, která jsou běžně binárně kódována v následující podobě:

v_1 = (0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1);

v_2 = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

kde v_1 je binární kód numerické hodnoty x_1 a v_2 je binární kód hodnoty x_2.

Rozdíl mezi čísly v hodnotě je nepatrný, ale v případě kódování se jejich chromozomy liší kromě prvního ve všech zbývajících bitech. Řešením tohoto problému je použití Grayova kódu, jehož základní vlastností je, že dvě sousední hodnoty jsou zakódovány tak, že se liší právě v jednom bitu. Rozdíl mezi běžným kódováním čísel a Grayovým kódováním je patrný z tabulky č. 1.

Tabulka 1- Příklad rozdílu standardního a Grayova kódování. [2]

Číslo	Binární kód	Grayův kód
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101

Volba vhodné reprezentace jedinců má výrazný vliv na úspěch genetického algoritmu. Způsobu reprezentace musí odpovídat operátory křížení a mutace, které pokud nejsou pro danou reprezentaci jedinců vhodně uzpůsobeny, nemusí genetický algoritmus nikdy dojít

k nalezení optimálního řešení, protože operátory křížení a mutace mohou mít destrukční charakter. V takovém případě pak místo toho, aby jedinec, který vznikl jako výsledek operátoru křížení, měl kombinaci vlastností svých rodičů, může získat vlastnosti zcela nečekané. Tím nebude zajištěno, že výsledkem křížení dobrých rodičů budou většinou dobří potomci.

1.4.2 Funkce vhodnosti

Hodnota funkce vhodnosti jedince vyjadřuje, jak moc dobře jedinec splňuje kritéria, které musí splňovat hledané optimální řešení. Sestavení funkce je úzce vázáno na řešenou problematiku. Funkce vhodnosti musí být sestavena takovým způsobem, aby bylo zajištěno, že pro jedince, který reprezentuje nejlepší řešení, nabyla nejlepší hodnoty, pro jedince reprezentující nejhorší řešení pak zase hodnoty nejhorší. Zpravidla je funkce sestavena takovým způsobem, aby nabývala hodnoty v intervalu <0,1>, kdy hodnota funkce vhodnosti pro nejlepšího jedince nabude hodnoty 1 a pro nejhoršího nabude hodnoty 0.

Někdy je velice obtížné stanovit funkci vhodnosti v případech, kdy o vhodnosti řešení má informace pouze jeden člověk, který je není schopen sdělit vhodným způsobem nebo není možné funkci vhodnosti sestavit z důvodu, že se hledá řešení problému, které je založeno citovém vjemu člověka. V tom případě algoritmus musí být v interakci s daným člověkem, který po každém cyklu algoritmu musí každého jedince sám ohodnotit. Pochopitelně takto realizovaný algoritmus je pomalý, nicméně i tímto způsobem je možné při omezeném počtu jedinců a generací dojít k výsledku. Příkladem takové aplikace je v USA v praxi využívaný systém Faceprints sloužící k identifikaci pachatelů trestné činnosti. Svědek trestné činnosti je vyzván k určení podobnosti jednotlivých obličejů s hledanou osobou, k čemuž využívá desetibodové stupnice. Na základě takto získaného ohodnocení genetický algoritmus provede selekci a pomocí genetických operátorů se vytvoří nová sada obličejů. [2]

1.4.3 Operátor selekce

Operátor selekce vybírá jedince, na kterých budou následně aplikovány genetické operátory, přičemž při výběru musí upřednostňovat lepší jedince. Operátor napodobuje proces přirozeného výběru vhodných jedinců v přírodě, kdy lepší jedinci mají větší šanci na přežití

a jsou v případném páření také svými protějšky při páření upřednostňováni před méně úspěšnými samci, kteří pravděpodobně opustí svět bez potomků a nebudou moct své nekvalitní geny šířit dále.

Existuje několik způsobů, jak může být operátor selekce realizován. Vždy však platí, že pravděpodobnost výběru jedince musí být úměrná jeho vhodnosti. Pravděpodobnost, že bude vybrán vhodnější jedinec, musí být vždy vyšší, než že bude vybrán jedinec horší.

Při návrhu selekčního mechanismu je nutné mít na zřeteli dvě věci. Pokud selekční mechanismus příliš upřednostňuje nejlepší jedince, dojde rychle k tomu, že se ztratí rozmanitost populace a následná evoluce bude silně omezená, protože jedinci brzy budou složeni ze stejných vlastností, které budou moct být změněny jenom velice obtížně, a hrozí, genetický algoritmus uvízne v blízkosti suboptimálního řešení. Na druhou stranu pokud však selekční mechanismus je příliš benevolentní, že příliš toleruje méně vhodné jedince, algoritmus se bude k optimálnímu řešení přibližovat velice pomalu. [1]

Nejčastější způsob selekce představuje selekce založená na přímé úměře k vhodnosti jedince, selekce založená na přímé úměře k pořadí jedince a turnajová selekce. Každá ze selekcí má své výhody i nedostatky. Často jsou selekce doplněny některými mechanismy, které zvyšují efektivnost algoritmu, tím že tyto nedostatky potlačují. Příkladem takových mechanismů může být škálování nebo elitismus.

Ruletová selekce přímo úměrná vhodnosti jedince

Ruletový mechanismus je zřejmě nejčastější formou selekčního mechanismu. Selekční mechanismus si lze v tomto případě představit jako ruletové kolo. V případě ruletového kola, je kolo rozděleno na stejně velké výseče a každá výseč reprezentuje určité číslo. V případě selekce založené na ruletovém mechanismu každá výseč reprezentuje jednoho jedince. Jednotlivé výseče ale nejsou stejně velké a jsou úměrné vhodnosti jedince, kterého reprezentují. Tím je zaručeno, že lepší jedinec bude mít na ruletě přidělenou větší výseč a selekčním mechanismem tak bude vybrán s větší pravděpodobností. Pak je generováno náhodné číslo, které vyjadřuje, kde se kulička na roztočené ruletě zastaví. Jedinec, do jehož přidělené výseče kulička padla, pak bude selekčním mechanismem vybrán. Budou na něho uplatněny genetické operátory a předá své geny následující populaci. [2]

Existuje několik způsobů, jak stanovit velikosti výseče pro každého jedince. Nejjednodušší způsob je, když velikost výseče jedince je přímo úměrná velikosti jeho vhodnosti. Toho lze docílit prostým sečtením vhodností všech jedinců v populaci a jedinci je pak přiřazena výseč, která proporcionálně odpovídá jeho vhodnosti v součtu všech vhodností. Předpokladem použití tohoto způsobu je, že funkce vhodnosti nabývá nezáporných hodnot a je maximalistická – nejlepší jedinec je ohodnocen funkcí vhodnosti nejvyšší možnou hodnotou.

Matematicky je možno tento vztah vyjádřit následujícím způsobem. Nechť máme populaci o velikosti N a nechť je ohodnocení každého jedince nezáporné, potom pravděpodobnost s jakou bude jedinec vybrán (a odpovídající velikost kruhové výseče), je dána následujícím vztahem [2]:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}, \quad i \in \{1,..,N\}, \tag{1}$$

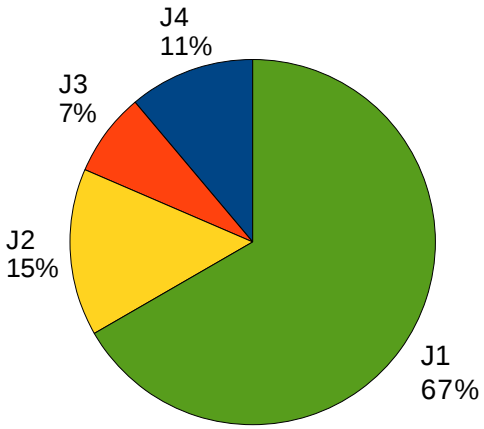
kde f_i představuje vhodnost *í*-tého jedince, $\sum_{j=1}^{N} f_j$ součet vhodností všech jedinců a p_i je pravděpodobnost výběru *í*-tého jedince.

Ruletový selekční mechanismus založený na přímé úměře však trpí nedostatkem. V případě, že v populaci se vyskytne jedinec, který bude oproti zbývajícím jedincům velice dobře ohodnocen a zbývající jedinci, budou naopak ohodnoceni nízko (tabulka č. 2), pak vysoce ohodnocenému jedinci bude přiřazena výseč pokrývající velkou část plochy ruletového kola (graf č. 1). Jedinec pak bude v selekčním mechanismu natolik dominantní, že následující generace získá velkou část genů pouze od něho. V populaci tak budou převažovat jeho geny a ztratí se variabilita populace.

Další výrazné negativum selekčního ruletového mechanismu založeného na přímé úměře vhodnosti jedince nastává tehdy, když rozdíly mezi jedinci jsou příliš malé. Potom každému jedinci je přiřazena výseč s přibližně stejně velkou plochou. Situace nastává v případech, kdy se algoritmus pohybuje v blízkosti optimálního řešení, rozdíly mezi vhodnostmi jedinců jsou malé a lepší jedinci pak nejsou před ostatními dostatečně zvýhodněni. Jejich lepší geny se do následující generace obtížně prosazují a algoritmus se náhodně pohybuje v okolí optima.

Tabulka 2 - Pravděpodobnosti jedinců při selekci přímo úměrně vhodnosti. [Zdroj vlastní]

Jedinec	J1	J2	J3	J4
Vhodnost	18	4	2	3
Pravděpodobnost	0,67	0,15	0,07	0,11



Graf 1- Rozložení pravděpodobností při selekci přímo úměrné vhodnosti. [Zdroj vlastní]

Nedostatek selekčního tlaku při prohledávání okolí optima je možné odstranit použitím škálování. **Škálování** představuje mechanismus, kdy vhodnosti jedinců jsou přepočteny takovým způsobem, že rozdíly mezi nejhoršími a nejlepšími jedinci jsou výrazně zvýšeny, přitom jsou ale zachovány původní porovnávací vztahy. [7]

Nejrozšířenější způsob škálování představuje lineární škálování vhodnosti, kdy vhodnost je přepočtena dle vztahu [7]:

$$\bar{f}_i = a + f_i . b\,, \quad i \in \{1,..,\ N\}\,, \tag{2}$$

kde *a*, *b* jsou konstanty, f_i vhodnost *í*-tého jedince, $\bar{f}_i$ škálovaná vhodnost *í*-tého jedince a *N* je počet jedinců v populaci.

Parametry *a*, *b* musí být voleny takovým způsobem, aby průměry ze škálované a původní vhodnosti všech jedinců v populaci byly shodné. Tím se zachová pravděpodobnost vylosování průměrného jedince a zachová se počet vybraných průměrných jedinců.

Musí tedy platit vztah [7]:

$$\frac{\sum_{i=1}^{N} \bar{f}_i}{N} = \frac{\sum_{i=1}^{N} f_i}{N}, \qquad (3)$$

kde $\bar{f}_i$ je škálovaná vhodnost *í*-tého jedince, f_i je neškálovaná vhodnost *í*-tého jedince a N je počet jedinců v populaci.

Ruletová selekce přímo úměrná pořadí jedince

Tato metoda předpokládá, že se jedinci seřadí vzestupně podle jejich vhodnosti a každému jedinci je pak přiřazeno pořadí.

Velikost výseče i-tého jedince na ruletě se pak spočítá dle vztahu [2]:

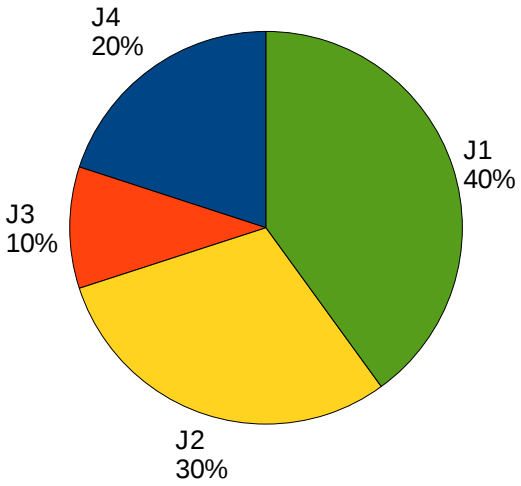
$$p_i = \frac{i}{\sum_{j=1}^{N} j} = \frac{2.i}{N.(N+1)}, \quad i \in \{1,..,N\}, \qquad (4)$$

kde N je velikost populace, $\sum_{j=1}^{N} j$ součet pořadí všech jedinců v uspořádané populaci a p_i pravděpodobnost výběru *í*-tého jedince. Index jedince i zároveň určuje jeho pořadí v uspořádané populaci.

Výhodou selekčního mechanismu založeného na přímé úměře k pořadí jedince v populaci je, že díky způsobu výpočtu velikosti výseče odpadá požadavek na funkci vhodnosti, že vhodnost musí být reprezentována nezápornou hodnotou. Selekce založená na pořadí také potlačuje roli nadprůměrně ohodnocených jedinců, kteří negativně mohou ovlivnit genetický algoritmus tím způsobem, že algoritmus bude často konvergovat pouze k suboptimálnímu řešení a bude prohledávat okolí pouze tohoto nadprůměrného jedince. Příliš velký rozdíl mezi nadprůměrným jedincem a ostatními je potlačen (tabulka č. 3 a graf č. 2). Také je zajištěn selekční tlak ke konci algoritmu, kdy populace je již složena z dostatečně dobrých jedinců, mezi kterými nejsou výrazné rozdíly a jejich výseče na ruletě, které by byly přímo úměrné funkci vhodnosti by byly jinak téměř stejné. Každý jedinec, by měl téměř stejnou pravděpodobnost výběru a nejlepší jedinec by se tak prosazoval velice obtížně a pomalu.

Tabulka 3 - Pravděpodobnosti jedinců při selekci přímo úměrné pořadí. [Zdroj vlastní]

Jedinec	J1	J2	J3	J4
Vhodnost	18	4	2	3
Pořadí	1	2	4	3
Pravděpodobnost	0,4	0,3	0,2	0,1



Graf 2 - Ruletová selekce přímo úměrná pořadí.
[Zdroj vlastní]

Nevýhodou selekce založené na pořadí jedinců je, že pokud se v populaci vyskytnou nadprůměrní jedinci, jejich nadprůměrnost nebude dostatečně zvýhodněna. Také pokud rozdíl vhodnosti nejhoršího a nejlepšího jedince je nepatrný, pak tento malý rozdíl bude pořadím zveličen a to i několikanásobně.

Nedostatky je možno potlačit použitím jiné než přímé lineární úměry pravděpodobnosti výběru k pořadí. Přesto ale negativa nemohou být plně potlačena. [5]

Selekce na pořadí se jeví pro použití jako výhodnější než selekce založená na úměře k vhodnosti jedince. Algoritmy založené na selekci přímo úměrné k pořadí jsou však pomalejší, protože selekční tlak v počáteční a střední fázi algoritmu není úměrný vhodnosti jedince a nadprůměrní jedinci se tak obtížněji prosazují. Na druhou stranu populace si po celou dobu zachovává svou variabilitu a nehrozí tolik, že by algoritmus uvízl v suboptimálním řešení. [2]

Turnajová selekce

Poslední nejčastěji používaný selekční mechanismus je založený na procesech v přírodě. V přírodě se jedinec musí při páření utkávat se svými konkurenty. Ten, kdo z nich vyhraje, získává právo předat své geny generaci následující. Vzhledem k tomu, že jedinci jsou ale geograficky odděleni, nemůže se jedinec utkat se všemi svými konkurenty ale pouze jenom s několika z nich. Turnajová selekce tento proces napodobuje. Vybere náhodně k jedinců ($k \geq 2$) a z nich pak vybere jedince s nejlepší vhodností. Mechanismus turnajové selekce se také často používá v modifikaci, kdy nejlepší jedinec vyhrává turnaj s určitou pravděpodobností. [2]

Turnajová selekce je oblíbená z důvodu, že je snadno implementovatelná. Na funkci vhodnosti nejsou kladeny žádné nároky a umožňuje pracovat jak s maximalistickou, tak i s minimalistickou funkcí. Díky parametru k je možno také snadno ovlivňovat selekční tlak během chodu algoritmu, tím zabránit předčasné konvergenci a udržovat dostatečnou variabilitu jedinců v populaci během celého algoritmu.

Elitismus

Problémem selekčního mechanismu založeného na náhodném výběru je, že nejlepší jedinec se nemusí vždy dostat do populace následující a tak dosud nejlepší nalezené řešení může být ztraceno. Z toho důvodu je do selekčního mechanismu implementován elitismus, který zajišťuje, že nejlepší jedinec ze současné generace je vždy vybrán do generace následující. Do následující generace nemusí být vybírán pouze jenom jeden jedinec, ale nejlepších jedinců může být vybráno hned několik.

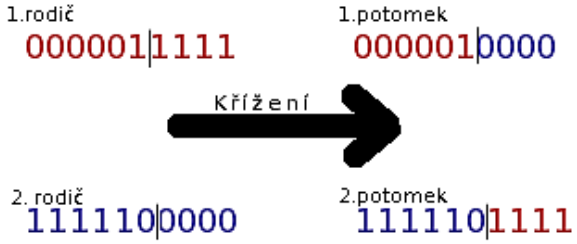
Použití elitismu k zabránění ztráty nejlepšího jedince výrazně zvyšuje účinnost genetického algoritmu. [5]

1.4.4 Operátor křížení

Operátor křížení napodobuje samotné páření, kdy pářením vznikají jedinci noví, jejichž vlastnosti jsou kombinací vlastností jejich rodičů. Dva jedinci, kteří byli vybráni

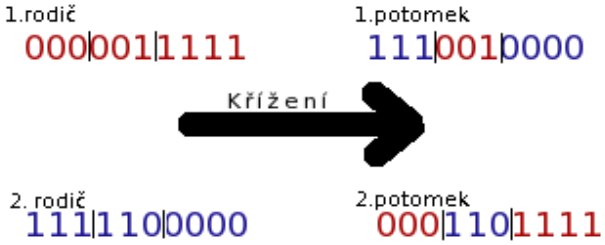
operátorem selekce, si prohodí část genů, čímž vzniknou dva noví jedinci. Genotypy obou nových jedinců obsahují dvě části genů (v případě jednobodového křížení), přičemž každou část genů získaly od jiného chromozomu z dvojice, která byla vybrána.

V případě že máme dva jedince, kteří jsou reprezentováni chromozomem jako binárním řetězcem o pevné délce, pak můžeme operátor křížení demonstrovat následujícím způsobem (obrázek č. 1). Nejprve je zjištěna délka chromozomu jedinců d a poté je generováno náhodné číslo j, $j \in <1, d>$, které určí polohu, v které se chromozomy rozloží na dvě části. Pak každý ze dvou potomků, získá právě jednu část od jednoho rodiče a druhou od druhého. Přitom musí být zachováno pořadí genů v chromozomu takovým způsobem, že i-tý gen rodiče, musí být umístěn na i-té místo v chromozomu potomka.



Obrázek 1 - Operátor křížení.
Převzato a upraveno z [10].

Příklad výše demonstroval tzv. jednobodové křížení, kdy se chromozomy rodičů dělí v jednom bodě. To ale ne zcela odráží křížení, které je realizováno v přírodě. Během páření si jedinci nezaměňují pouze jednu část, ale zaměňovaných částí chromozomu může být několik. Pak hovoříme o křížení vícebodovém (obrázek č. 2), kdy chromozom rodiče se dělí ve více bodech.



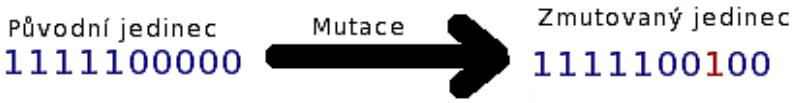
Obrázek 2 - Vícebodové křížení.
Převzato a upraveno z [10].

Operátor křížení je aplikován na vybraný pár, který byl vybrán operátorem selekce, s určitou pravděpodobností. Je doporučováno volit vysokou pravděpodobnost

v rozmezí 85-95%. Pokud na vybraný pár není aplikován operátor křížení, vytvoří se kopie rodičů a tyto kopie jsou pak považovány za nové potomky a dále podstupují operátor mutace či jsou vloženi rovnou do nové generace jedinců. [2]

1.4.5 Operátor mutace

Genetický operátor mutace náhodně vybere gen u chromozomu, u kterého zamění jeho alelu (obrázek č. 3). Operátor mutace napodobuje mutaci v přírodě, kdy u potomka může dojít vzácně k mutaci, která zcela nečekaně změní některou jeho vlastnost. Při kódování pomocí binárních řetězců se mutace realizuje záměnou hodnoty bitu na náhodně zvolené pozici.



Obrázek 3 - Operátor mutace.
Převzato a upraveno z [10].

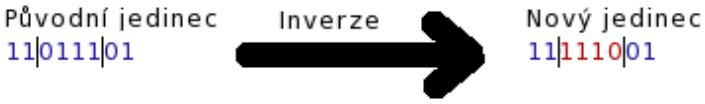
Mutace má pozitivní vliv na průběh algoritmu tím, že vytváří u jedinců nové vlastnosti. V případech, kdy se algoritmus nachází v blízkosti suboptimálního řešení a křížením algoritmus prohledává pouze jeho okolí, mutace udržuje variabilitu populace tím, že náhodně mění vlastnosti jedinců a prohledává se tak celý prostor potenciálních řešení nikoli pouze okolí. [10]

Mutace obohacuje prohledávaný prostor o řešení, kterých není možno dosáhnout křížením, protože určitá vlastnost nemusí být obsažena u jedinců počáteční populace. Mutace však narušuje genetickou informaci jedinců a příliš velké uplatňování operátoru mutace má za následek, že algoritmus se v prohledávaném prostoru bude pohybovat více náhodně a k optimálnímu řešení bude konvergovat velice pomalu.

Stejně jako operátor křížení tak i operátor mutace se uplatňuje s určitou pravděpodobností. Na rozdíl od křížení se doporučuje uplatňovat mutaci s malou pravděpodobností v rozmezí 3-5%. V případě, že algoritmus se pohybuje v blízkosti suboptimálního řešení a není schopen nalézt řešení optimální, je vhodné pravděpodobnost mutace zvýšit. [2]

1.4.6 Operátor inverze

Operátor inverze přehodí pořadí souvislé oblasti genů (obrázek č. 4). Jedná se o čtvrtý Hollandův operátor, který je dnes již do algoritmu implementován vzácně a od jeho používání se upouští. Ukázalo se, že nezvyšuje účinnost genetického algoritmu a na nalezení optimálního řešení má zanedbatelný vliv. [10]



Obrázek 4 - Operátor inverze.
Převzato a upraveno z [10].

1.5 Věta o schématech

Holland položil teoretický základ genetického algoritmu a dokázal, že genetický algoritmus funguje a s rostoucím počtem generací se populace jedinců, kteří reprezentují potenciální řešení, zlepšuje. Prostřednictvím pravděpodobnosti přežití lepších schémat definoval tzv. větu o schématech, která tento teoretický důkaz poskytuje. V kapitole bude demonstrováno odvození této věty. Cílem odvození je najít vztah pro určení pravděpodobnosti přežití lepších schémat.

Pro potřeby odvození důkazu je nejprve nutné definovat pojem **schéma**. Předpokládejme, že chromozomy jsou reprezentovány posloupností bitů. Schéma pak definujeme jako řetězec délky l nad množinou {0, 1, *}, kde znak * představuje zástupný symbol. Schéma si je možné představit jako šablonu popisující určitou podmnožinu délky l. [2]

Pokud máme definované schéma S o délce l = 5 jako S = (01**1), pak toto schéma zastupuje chromozom, který má shodné hodnoty na pozicích bitů, které jsou ve schématu definovány. Schéma S tedy například zastupuje chromozom CH_1 = (01111). Chromozom CH_2 = (11001) však již schéma S nezastupuje, protože se od schématu S liší na pozici prvního bitu.

Celkový počet chromozomů, které schéma zastupuje, je závislý na počtu volných pozic ve schématu. Schéma neobsahující žádnou volnou pozici má přirozeně pouze jednu možnou instanci, kterou schéma může zastupovat. Pokud schéma má *k* volných pozic, pak schéma zastupuje 2^k možných instancí řetězce příslušné délky.

Řád schématu *o(S)* je definován jako počet určených pozic schématu *S*, to jest pozic, které jsou ve schématu obsazené hodnotou 1 nebo 0. Definiční délkou schématu *d(S)* rozumíme vzdálenost mezi první a poslední určenou pozicí schématu. Pokud máme například konkrétní schéma *S* = (*01*0***), potom *o(S)* = 3 a *d(S)* = 5 – 2 = 3. [2]

Při odvození věty o schématech předpokládejme binární kódování chromozomů, selekci přímo úměrnou vhodnosti jedince, operátor jednobodového křížení a jednobodový operátor mutace. Pak můžeme definovat následující vztahy.

Nechť S je libovolné schéma, které má alespoň jednu instanci v populaci *P(t)*. Nechť *m(S,t)* je počet instancí schématu *S* v generaci *t* a nechť *u(S,t)* je střední ohodnocení schématu *S* v generaci *t*, dané vztahem. Pak střední ohodnocení schématu *S* v generaci *t* spočítáme [2]:

$$u(S,t) = \frac{1}{m(S,t)} \sum_{j=1}^{m(S,t)} f_{jt}, \qquad (5)$$

kde f_{jt} je vhodnost *j*-té instance schématu *S* v generaci *t*.

Při použití selekce s pravděpodobností výběru, která je přímo úměrná jeho vhodnosti, je očekávaný počet potomků m_x řetězce *x* roven podílu [2]:

$$m_x = \frac{f(x)}{f(t)}, \qquad (6)$$

kde *f(x)* je ohodnocení řetězce *x* a *f(t)* je průměrné ohodnocení jedinců populace *P(t)* v generaci *t*.

V případě, že *x* je instancí schématu *S*, dostáváme dosazením vztahu (5) do vztahu (6) zjednodušený odhad pro očekávaný počet výskytů instancí schématu *S* v další generaci ve tvaru [2]:

$$E(m(S,t+1)) = \frac{u(S,t)}{f(t)} m(S,t), \qquad (7)$$

kde *E(m(S,t+1))* je očekávaný počet instancí schématu *S* v generaci *t+1*, *u(S,t)* střední ohodnocení schématu *S* v generaci *t*, *f(t)* průměrné ohodnocení populace *P(t)* a *m(S,t)* je počet instancí schématu *S* v generaci *t*.

Je zřejmé, že křížení i mutace mohou způsobit poškození konkrétní instance schématu *S*, čímž se sníží počet instancí tohoto schématu v populaci. Proto pravděpodobnost přežití lepších schémat musí být o tuto skutečnost ošetřena.

Pravděpodobnost destrukce schématu $p_d(S)$ při křížení závisí na definiční délce schématu. Protože existuje celkem l - 1 míst, na kterých může dojít ke křížení, a z toho na $d(S)$ místech může dojít ke zničení schématu, pravděpodobnost destrukce schématu p_d můžeme spočítat dle vztahu [2]:

$$p_d(S) \leq p_c \frac{d(S)}{(l-1)}, \qquad (8)$$

kde $p_d(S)$ je pravděpodobnost destrukce schématu *S*, $d(S)$ definiční délka schématu *S*, l délka schématu *S* a p_c je pravděpodobnost jednobodového křížení.

Nerovnost ve vztahu je použita z toho důvodu, že při křížení nemusí dojít k tomu, že na $d(S)$ místech ztratí schéma svou instanci, neboť křížením může být instance opět obnovena. Například když jsou kříženi dva jedinci, kteří jsou oba zastoupeny schématem *S*, pak oba výslední jedinci budou opět zastoupeni schématem *S*.

Doplňková pravděpodobnost p_{s_c} odvozená ze vztahu (8) udává, jaká je pravděpodobnost, že po křížení dvou chromozomů, z nichž jeden je instancí schématu *S*, vznikne alespoň jeden řetězec, který se schématem S shoduje. Pravděpodobnost přežití schématu po aplikaci operátoru křížení spočítáme [2]:

$$p_{s_c} \geq 1 - p_c \frac{d(S)}{(l-1)}, \tag{9}$$

kde p_{s_c} je pravděpodobnost přežití schématu *S* po aplikaci jednobodového operátoru křížení, p_c pravděpodobnost křížení, *d(S)* definiční délka schématu *S* a *l* je délka schématu *S*.

Ze vztahu (9) logicky vyplývá, že čím je definiční délka schématu menší, tím je větší pravděpodobnost zachování schématu i po aplikaci operátoru jednobodového křížení.

Schéma může být v chromozomu narušeno také operátorem mutace. Pravděpodobnost, že nedojde ke změně schématu, je pak určena pravděpodobností, že nebude mutace uplatněna na dané schéma *S*, a řádem schématu [2]:

$$p_{s_m}(S) = (1 - p_m)^{o(S)}, \tag{10}$$

kde $p_{s_m}(S)$ je pravděpodobnost přežití schématu *S* po aplikaci operátoru mutace, p_m pravděpodobnost uplatnění mutace na dané pozici v řetězci a *o(S)* je řád schématu *S*.

Dle vztahu (10) pak logicky odvodíme, že vliv mutace na možnou ztrátu instance schématu *S* je tím menší, čím menší je řád tohoto schématu.

Spojením vlivu selekce (7), operátoru křížení (9) a mutace (10) do jednoho vztahu, potom získáme vztah pro očekávaný počet výskytů schématu *S* v následující generaci. Tento vztah je nejdůležitějším vztahem v oblasti genetických algoritmů a Holland ho zformuloval do tzv. **věty o schématech** [2]:

Nechť pro populaci P(t) platí, že m(S,t) je počet instancí schématu S v populaci v čase t, u(S,t) je střední hodnota ohodnocení schématu S a f(t) je průměrné ohodnocení populace čase v t. Potom očekávaný počet instancí schématu S v populaci P(t+1) je zdola ohraničený nerovností:

$$E(m(S,t+1)) \geq \frac{u(S,t)}{f(t)} m(S,t)(1-p_c \frac{d(S)}{l-1})(1-p_m{}^{o(S)}), \qquad (11)$$

kde p_c je pravděpodobnost křížení, p_m pravděpodobnost uplatnění mutace na dané pozici v řetězci, *d(S)* definiční délka schématu *S*, *o(S)* řád schématu *S* a *i* je délka schématu *S*.

Ze vztahu (11) je zřejmé, že pokud střední ohodnocení schématu *S* je vyšší než průměrné ohodnocení populace *u(S,t) > f(t)*, potom se výskyt schématu S v následující populaci zvyšuje s rostoucí pravděpodobností (analogicky výskyt horšího schématu klesá s rostoucí pravděpodobností). Dle vztahu (9) jsou také vůči destrukci při uplatnění operátoru křížení odolnější schémata s malou definiční délkou *d(S)* a proti operátoru mutace jsou odolnější schémata s nižším řádem *o(S)*. [2]

Z těchto důvodů krátká schémata nízkého řádu, která mají průměrné ohodnocení větší než průměrné ohodnocení celé populace, mají exponenciálně rostoucí počet instancí v následujících generací. Někteří autoři tato schémata nazývají **stavebními bloky** a jejich nalezením může být algoritmus výrazně urychlen. [1]

Věta o schématech dokazuje, že s rostoucím počtem generací, se populace zlepšuje. Nevhodná schémata opouští populaci a vzniklá nová schémata přežívají dokud jsou dostatečně dobrá oproti schématům ostatním.

Formulovaná věta o schématech, ve tvaru v jakém ji odvodil Holland, je uplatněna v případě jednoduchého genetického algoritmu. Pro řešení řady specifických složitých problémů, ale není možné jednoduchý Hollandův genetický algoritmus použít, aniž by byla zvolena jiná forma genetických operátorů a odlišná reprezentace jedinců. Nevhodnou reprezentací potenciálních řešení, nevhodnou formou genetického operátoru křížení, mutace,

nevhodným způsobem selekce, nevhodně nastavenými parametry algoritmu, špatně stanovenou funkcí vhodnosti, můžeme docílit toho, že věta o schématech nemůže být uplatněna, protože námi navržený algoritmus se chová zcela jinak, než jak jsme při jeho návrhu předpokládali. Z toho důvodu každá modifikace genetického algoritmu, odlišná forma reprezentace potenciálních řešení a každá byť i jen drobná změna genetického operátoru by měla být důsledně otestována, zda místo zamýšleného zvýšení efektivnosti algoritmu nemá spíše destrukční charakter a ve výsledku genetický algoritmus prostor prohledává zcela náhodně.

1.6 Možnosti využití genetického algoritmu

V praxi se pomocí genetických algoritmů řeší úlohy optimalizace, využívají se k vyhledání nejlepší topologie, technologii, výrobě, a v průmyslové automatizaci, a jako alternativní metody učení neuronových sítí. Na rozdíl od gradientních metod, které se při učení neuronových sítí jinak běžně používají, představují genetické algoritmy jiný přístup, kdy je prostřednictvím genetických operátorů prohledáván celý prostor potenciálních řešení. Prostor je prohledáván paralelně prostřednictvím populace jedinců a paralelním prohledáváním je dosaženo toho, že genetický algoritmus s větší pravděpodobností nalezne optimální řešení rychleji než gradientní metoda. [2]

Genetický algoritmus je natolik obecný, že je ho možné využívat jako optimalizační metodu v mnoha oblastech a na rozdíl od ostatních metod je navíc odolný od uvíznutí v suboptimálním řešení a dokáže se vypořádat s velkou množinou přípustných řešení. Vždy však závisí na způsobu, jakým je algoritmus implementován a nastavení jeho parametrů. [3]

Genetický algoritmus při řešení problémů s velkou množinou potenciálních složitých řešení, může být výpočetně velice náročný. Sice nachází dostatečně dobré řešení, optimální řešení ale nemusí být nalezeno vždy. V případě, že máme k dispozici znalosti o dané problematice a máme například určitou analytickou metodu, která dokáže zlepšit nalezené řešení například výpočtem optimálních numerických parametrů nalezeného řešení, je vhodné algoritmus modifikovat a tuto metodu do algoritmu vložit v podobě nového genetického operátoru. To povede k tomu, že prohledávání prostoru se výrazně urychlí a bude mnohem cílenější. [2]

2 Principy genetických algoritmů používaných v genetickém programování

Genetické programování lze považovat za rozšíření genetického algoritmu v tom smyslu, že potenciální řešení nejsou reprezentována řetězci pevné délky, ale jsou reprezentována hierarchickou programovou strukturou. John Koza, profesor na Stanfordské univerzitě vytvořil metodologii genetického programování, kdy se rozhodl reprezentovat potenciální řešení rozdílnou strukturou než řetězcem a této reprezentaci přizpůsobil jednotlivé operátory genetického algoritmu včetně rozšiřujících metod, které zvyšují efektivnost algoritmu. [10]

Genetické programování umožňuje reprezentovat potenciální řešení přímočařejší strukturou a díky tomu má širší možnosti uplatnění než jednoduchý genetický algoritmus, který by sice mohl být také k řešení daných úloh použit, ale reprezentace řešení v podobě řetězce by byla příliš složitá. Kromě řešení složitých úloh optimalizace se často také používá k automatickému programování, kdy prostřednictvím genetického programování se na základě definovaných vstupů a požadovaných výstupů automaticky “pěstují” programy. [3]

Kromě změny reprezentace potenciálních řešení zůstává princip a průběh genetického algoritmu stejný. Z počátečních náhodně vygenerovaných potenciálních řešení se aplikací operátorů selekce, křížení a mutace získávají nová řešení, která pak postupují do další generace. Cyklus se opakuje tolikrát, dokud není splněna ukončovací podmínka.

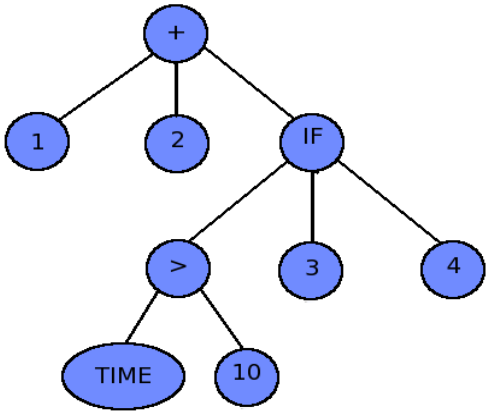
Protože genetické programování je rozšířením genetického algoritmu, většinu prvků mají oba algoritmy společné, v kapitole tak budou popsány pouze ty prvky algoritmu genetického programování, v kterých se algoritmy liší.

2.1 Reprezentace jedinců

Způsob reprezentace potenciálních řešení je v případě genetického programování klíčová. Jednak určuje způsob, jakým genetické operátory musí s jedincem manipulovat, a jednak způsob reprezentace omezuje prohledávaný prostor. Genetické algoritmy,

které reprezentují jedince v podobě chromozomu s pevnou délkou, se ukázaly jako dostatečné k řešení mnoha problémů. Nicméně v některých případech, se jako vhodnější forma reprezentace jeví reprezentace v podobě hierarchické struktury. [3]

John Koza navrhl reprezentovat řešení v podobě hierarchické stromové struktury ve formě tzv. **syntaktických stromů**. Strom je pak složen z **terminálů** (funkcí) a **neterminálů** (proměnných a konstant). Tato reprezentace programového kódu odpovídá reprezentaci S - výrazů programovacího jazyka LISP a byl to také jeden z důvodů, proč se Koza rozhodl tento programovací jazyk pro své experimenty použít. Je třeba upozornit, že Koza ve své knize [3] zdůrazňuje, že to nebyl jediný důvod, proč se rozhodl pro jazyk LISP, ale důvodů pro tuto volbu bylo několik. Jazyk LISP není jediným jazykem, prostřednictvím něhož je možné realizovat genetické programování, je možné použít i jiné programovací jazyky, vždy je ale nutné generované výrazy reprezentovat ve vhodné formě syntaktického stromu.



Obrázek 5 - Příklad syntaktického stromu. Upraveno a převzato z [3].

Příklad na obrázku č. 5 reprezentuje výraz jazyku LISP *(+1 2 (IF (>TIME 10) 3 4))* ve stromové struktuře. Díky stromové struktuře je výraz snadno modifikovatelný genetickými operátory. Jakýkoliv uzel stromu může být nahrazen jiným uzlem. Místo zvoleného uzlu může být generována nová větev stromu či samotný strom se může stát součástí složitějšího výrazu. Výhodou této reprezentace je také skutečnost, že strom není omezen a prohledávaný prostor možných řešení je nekonečný.

Množina možných struktur v genetickém programování je množinou všech možných spojeních funkcí, které mohou být rekurzivně složeny z množiny funkcí $F = \{ f1, f2 , .., f_{Nunc}\}$

a množiny terminálů $T = \{a_1, a_2, .., a_{Nterm}\}$. Každá funkce f_i z množiny funkcí F má určen počet argumentů, který určuje tzv. **aritu funkce**. Například funkce sčítání je dvouaritní, protože vyžaduje dva argumenty, zatímco funkce sinus je jednoaritní, protože vyžaduje pouze jeden argument. [3]

Množina F může zahrnovat následující funkce [3]:

- aritmetické operátory (+, -, *, /, atd.),
- matematické funkce (sin, cos, exp, log, atd.),
- booleovské operace (AND, OR, NOT, atd.),
- podmíněné operátory (If – Then – Else, atd.),
- iterační výrazy (Do-Until, atd.),
- rekurzivní výrazy,
- specifické funkce vázající se k řešení dané problematiky.

Terminály jsou typicky tvořeny proměnnými (reprezentující zpravidla vstupy, senzory, detektory a stavové proměnné systému) nebo konstantami (jako např. konstanta 3 nebo booleovská hodnota FALSE). V některých případech může být terminálem funkce nevyžadující žádný explicitní argument. Tato funkce zpravidla vrací určitou hodnotu reprezentující stav systému. [3]

Je zřejmé, že aby měl daný výraz smysl, musí množině použitých funkcí odpovídat množina požadovaných terminálů. Pokud např. chceme, aby program obsahoval cyklus, musí množina terminálů obsahovat konstantu, která daný cyklus zastaví. Z tohoto důvodu musí být množina funkcí a terminálů definována takovým způsobem, že musí splňovat **požadavky uzavřenosti** (closure) a **postačitelnosti** (sufficiency).

Uzavřeností se rozumí, že každá z funkcí obsažených v množině F může přijmout jako svůj argument libovolnou hodnotu či datový typ, jenž může být výsledkem použití kterékoliv funkce z množiny F, nebo kteroukoliv hodnotu libovolného terminálu z množiny T. [2]

Často z důvodu požadavku uzavřenosti musí některé funkce být v programu implementovány v podobě svých “chráněných verzí”. Pokud v množině funkcí F je obsažena funkce dělení, musí zpravidla být implementována ve své chráněné verzi, protože jinak hrozí,

že nastane případ dělení nulou, pro který tato funkce není definována, a celý algoritmus bude ukončen z důvodu, že nastal nečekaný stav, s kterým se není schopen program vypořádat. Chráněná verze funkce musí ošetřit situace, při nichž by nemohla být jinak podmínka uzavřenosti naplněna. [2]

Požadavek postačitelnosti znamená, že množiny *F* a *T* musí obsahovat takové funkce a terminály, že pomocí těchto prvků bude možné vyjádřit řešení daného problému. [2]

V závislosti na konkrétním problému to může být někdy snadné, ale často tento úkol vyžaduje veliké úsilí a detailní porozumění problému.

Dobře definovaná množina funkcí a terminálů významně ovlivňuje efektivnost celého algoritmu. Pokud v množině chybí některá klíčová funkce nebo terminál je pravděpodobné, že algoritmus nebude moct nalézt optimální řešení, doba vykonávání algoritmu se výrazně prodlouží či výsledný výraz bude pro reálné použití příliš složitý.

2.2 Počáteční populace

Pro každého jedince v populaci je na počátku algoritmu genetického programování vygenerován syntaktický strom obsahující náhodný výraz, který jedinec reprezentuje jako potenciální řešení. Strom je generován tím způsobem, že je náhodně zvolena funkce f_i z množiny *F*, která je zvolena za kořen stromu. Následně pro každý argument dané funkce f_i je generován uzel, který může být terminálem nebo funkcí. V případě, že by vygenerovaným uzlem pro argument byla vygenerována opět funkce, musí být i pro tuto funkci vygenerovány argumenty. Generování pokračuje tak dlouho, dokud nejsou všechny větve stromu uzavřeny terminály a strom je tak kompletní.

Z důvodu, že generování probíhá zcela náhodně, velice často nastávají situace, kdy nemůže být strom dokončen, protože obsahuje vysoký počet neuzavřených funkcí, na které nenavazuje žádný terminál a operační paměť počítače je již vyčerpána. K zabránění této situace je při generování jedince definován parametr h_{max}, který představuje maximální hloubku stromu. Jakmile při generování jedince větev stromu dosáhne ve stromě hloubky

h_{max}- 1, pak jako následující uzel pro tuto větev musí být generován terminál, díky čemuž bude větev uzavřena a nebude hrozit, že generování stromu se nikdy nezastaví (bylo by zastaveno až vyčerpáním paměti počítače).

Existují dvě metody generování syntaktických stromů: úplná (full method) a růstová (grow method). Při **generování stromu úplnou metodou** všechny větve stromu musí mít právě hloubku parametru h_{max}. Při **generování stromu metodou růstovou** hloubka větve stromu nesmí překročit maximální hloubku stromu h_{max}. [10]

Růstová metoda umožňuje generování mnohem rozmanitějších stromů. Při růstové metodě je možné také prostřednictvím nastavení pravděpodobnosti generování jednotlivých terminálů a funkcí ovlivňovat strukturu stromu.

Koza doporučuje ke generování stromu používat metodu, kterou nazval výrazem **"ramped half and half"**, kdy polovina jedinců v populaci je generována růstovou metodou a polovina metodou úplnou. Také doporučuje měnit maximální hloubku h_{max} při generování jednotlivých stromů v rozmezí od 2 do specifikované maximální hloubky takovým způsobem, že například při generování počáteční populace, kdy byla jako maximální hloubka stromu stanovena hodnota h_{max}= 6, pro 20% jedinců populace by byla při generování pro maximální hloubku stromu použita hodnota h_{max} = 2, pro dalších 20% jedinců populace h_{max}= 3, pro dalších 20% jedinců bude mít hloubku h_{max} = 4 atd. [3]

Do počáteční populace může být také vložen jedinec, který reprezentuje výraz, o němž jsme přesvědčeni, že by mohl být hledaným výrazem. V tom případě ale Koza doporučuje, aby počáteční populace byla složena z jedinců, kteří mají podobnou vhodnost jako takto vkládaný jedinec, jinak hrozí, že se tento vložený jedinec stane dominantním a hned po první generaci v populaci převládnou jedinci se stejným genetickým složením. [3]

2.3 Funkce vhodnosti

Funkce vhodnosti může být sestavena obdobně jako u jednoduchého genetického algoritmu mnoha způsoby. Vždy ale musí odrážet kvalitu daného potenciálního řešení v rámci dané problematiky. Většinou je funkce sestavena takovým způsobem, že nabývá hodnot v intervalu <0,1>, kdy nejlepšímu řešení je přiřazena hodnota 1, nejhoršímu hodnota 0.

Nejčastěji používanou funkcí vhodnosti v případě genetického programování je funkce, která reprezentuje chybu mezi požadovaným výstupem a výstupem, který program navrací (v případě automatického programování). Funkci je možno sestavit následujícím způsobem [3]:

$$e_i = \sum_{j=1}^{N} |y_{i,j} - c_j|, \qquad (12)$$

kde e_i představuje funkci vhodnosti *í*-tého jedince, $y_{i,j}$ výsledek daného řešení pro *j*-tý případ, c_j hodnotu požadovaného výrazu pro *j*-tý případ a *N* počet případů.

Výsledkem funkce bude vhodnost nabývající hodnoty v rozmezí <0,∞), kdy čím nižší hodnoty vhodnost bude nabývat, tím se bude jednat o lepší řešení. Pokud bude nalezeno řešení optimální, bude vhodnost rovna hodnotě 0.

Pokud hledaným výrazem je výraz vracející booleovskou hodnotu nebo symbolickou hodnotu, pak funkce vhodnosti může navracet počet, v kolika případech se výstup výrazu neshoduje s požadovanou hodnotou. [3]

Reprezentace jedinců v genetickém programování není omezena na rozdíl od genetických algoritmů, které reprezentují potenciální řešení nejběžněji binárním řetězcem s pevnou délkou. Genetické programování tak prohledává neomezený prostor potenciálních řešení včetně příliš složitých výrazů, které jsou natolik složité, že se stávají nežádoucími a raději se uspokojíme s jednodušším výrazem i za tu cenu, že složitější výraz by byl přesnější. Ačkoliv při generování počáteční generace jsou stromy omezeny maximální možnou hloubkou, aplikací genetických operátorů mohou vznikat složité stromové struktury. Navíc složité struktury jsou také nežádoucí z toho důvodu, že výpočet vhodnosti je mnohem náročnější a jak jsou geny složitého řešení předávány dále, algoritmus je postupně zpomalován.

Penalizace jedince za nadměrný počet uzlů ve stromu představuje dobrý způsob, jak počet složitých jedinců v populaci omezit. Pokud strom jedince dosáhne určitého počtu uzlů, pak za každý další uzel, který přesahuje tento počet, bude jedinec penalizován tím způsobem, že mu bude vhodnost zhoršena úměrně k počtu uzlů. Dle způsobu úměry k počtu přesahujících uzlů pak můžeme rozlišovat např. exponenciální, lineární nebo logaritmickou penalizaci. [10]

Ve většině literatury je používána normovaná vhodnost, která nabývá hodnot v rozmezí <0,1>. Pro potřeby srovnání výsledků je proto vhodné výše navrženou funkci vhodnosti (12) po aplikaci penalizace do tohoto intervalu převést prostřednictvím vztahu [3]:

$$f_i = \frac{1}{(1+e_i)}, \qquad (13)$$

kde f_i je přepočtená vhodnost *í*-tého jedince, $f_i \in (0,1>$, a e_i je původní vhodnost jedince, $e_i \in <0,\infty)$.

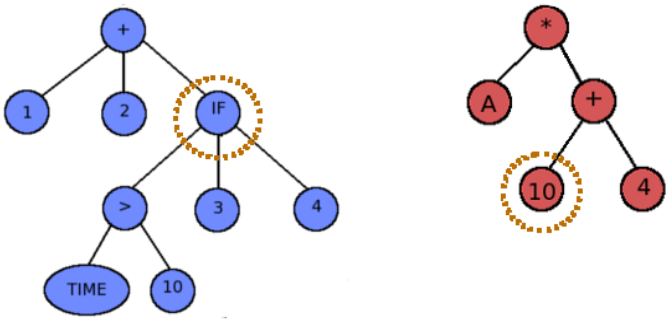
Optimální řešení pak bude reprezentováno hodnotou 1 a čím nižší hodnota bude vhodnosti přiřazena, tím bude dané řešení horší.

Vzhledem ke způsobu, jakým je generována počáteční populace a jsou realizovány genetické operátory, může nastat situace, kdy vygenerované řešení bude nepřípustné, nebude smysluplné. V takovém případě by měla funkce nabýt nejhorší možné hodnoty, čímž bude zajištěno, že jedinec reprezentující dané řešení bude vybrán operátorem selekce s co nejmenší možnou pravděpodobností a nebude tak šířit své nekvalitní geny dále. V případě vhodnosti, která byla vztahem (13) převedena do intervalu (0,1> takovému jedinci bude přiřazena vhodnost s hodnotou 0. [2]

2.4 Operátor křížení

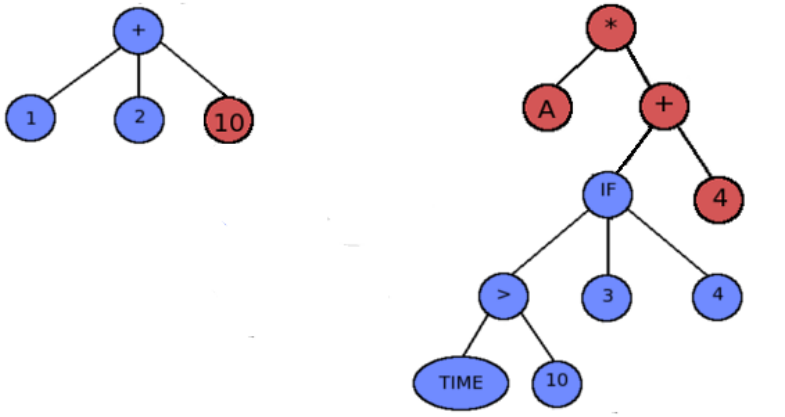
Operátor křížení musí být přizpůsoben odlišné reprezentaci jedinců, kdy v případě genetického programování nejsou jedinci reprezentováni řetězcem bitů ale stromovou strukturou. Jedinci si tak nebudou vyměňovat část řetězce, ale budou si vyměňovat část své stromové struktury.

Na počátku operátor křížení vytvoří kopie rodičů a vybere u každého zkopírovaného rodiče uzel v jeho stromu (obrázek č. 6).



Obrázek 6 - Operátor jednobodového křížení: náhodný výběr uzlů křížení u kopií rodičů. Vytvořeno dle [3].

Vybrané uzly se stanou kořenem větve stromu, která je pak z každého zkopírovaného rodiče vyjmuta a vložena do druhého na místo, které se uvolnilo odtržením větve stromu (obrázek č. 7). Jedinci si tak vlastně vymění některou ze svých větví stromu mezi sebou.



Obrázek 7 - Operátor jednobodového křížení: výslední potomci. Vytvořeno dle [3].

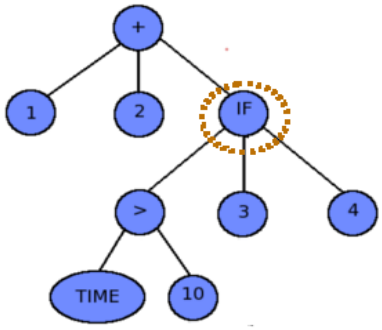
Koza doporučuje pro křížení stanovit maximální povolenou hloubku nově vzniklých jedinců, aby tak bylo zabráněno tvorbě složitých řešení. Pokud je tato hloubka překročena u některého z nově vytvořených jedinců, jedinec je vyřazen a místo něho bude do nové populace vložen jeden z jeho rodičů. [3]

Koza na svých příkladech omezil maximální možnou hloubku jedince na hodnotu 17. Tato hodnota se ukázala jako dostatečná. Pro představu při tomto nastavení může křížením jako největší program, který využívá funkce s nejvyšším počtem argumentů 2, vzniknout program obsahující 2^{17} = 131 072 uzlů funkcí a terminálů. Průměrný řádek programu napsaný v jazyce LISP je složen z kombinace 4 terminálů nebo funkcí. Při tomto nastavení tak může vzniknout program obsahující 33 000 řádků. Koza na svých příkladech nacházel optimální řešení, které se zpravidla skládalo z 500 funkcí a terminálů. [3]

Jinou možností jakou zabránit tomu, aby nedocházelo ke generování složitých řešení je použít metodu penalizace. Její nevýhodou však je, že složití jedinci nebudou ihned zaměněni svými rodiči, ale vstoupí do následující populace, přičemž dojde k náročnému výpočtu jejich vhodnosti a to celý algoritmus zpomalí. Na druhou stranu penalizace dává alespoň částečnou možnost složitým jedinců se prosadit, pokud tito jedinci budou obsahovat dostatečně dobré řešení. Určitým kompromisem se jeví tyto dvě metody uplatňovat společně. Samozřejmostí je, že maximální povolená hloubka jedince pro penalizaci musí být nižší než maximální povolená hloubka jedince po aplikaci operátoru křížení. [1]

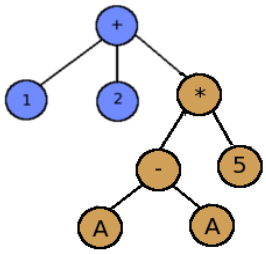
2.5 Operátor mutace

Operátor mutace je v genetickém programování realizován následujícím způsobem. Nejprve je náhodný způsobem v jedinci stromu určen uzel (obrázek č. 8), na kterém bude mutace uplatněna.



Obrázek 8 - Operátor mutace: výběr uzlu.
Vytvořeno dle [3].

Uzel je včetně jeho případného podstromu odstraněn a na jeho uvolněné místo ve stromu je vygenerována nová větev (obrázek č. 9). Přitom je nově generovaná větev stromu opět omezena svou max. hloubkou.

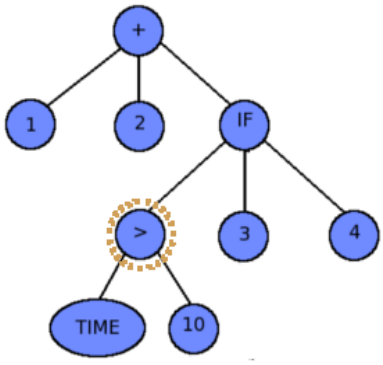


Obrázek 9 - Operátor mutace: nahrazení uzlu novou větví. Vytvořeno dle [3].

Koza vyjádřil pochybnosti o tom, zda vůbec má být operátor mutace v genetickém programování nasazen a nepovažuje ho za natolik potřebný, aby měl na kvalitu výsledného řešení vliv. Nicméně řada dalších autorů operátor mutace do svých algoritmů implementuje a za nepotřebný ho nepovažuje. [3][5]

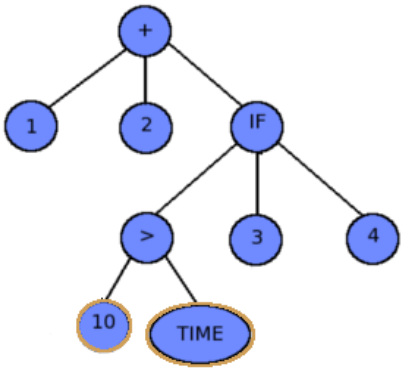
2.6 Operátor permutace

V případě jednoduchého genetického algoritmu operátor inverze přehodil hodnoty genů v určité části chromozomu. Pokud byl tento operátor aplikován na chromozom jedince s vysokou vhodností, mohl napomoct genetickému algoritmu získat chromozom, který by lépe vyhovoval zadanému řešení. [5]



Obrázek 10 - Operátor permutace: výběr uzlu. Vytvořeno dle [3].

Permutace v genetickém programování právě odpovídá inverzi v jednoduchém genetickém algoritmu. Permutace spočívá v tom, že je náhodně vybrán uzel funkce ve stromu (obrázek č. 10), kterému jsou pak argumenty náhodně přehozeny (obrázek č. 11). [3]



Obrázek 11 - Operátor permutace: prohození argumentů funkce. Vytvořeno dle [3].

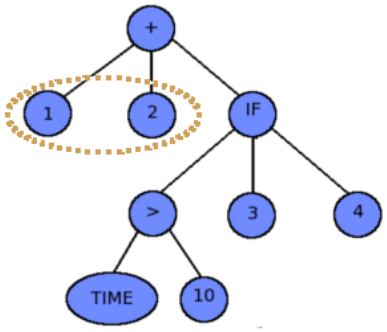
Od inverze se permutace liší v tom, že umožňuje náhodně změnit potenciální řešení více způsoby, zatímco inverze v jednoduchém genetickém algoritmu mění strukturu jedince pouze jediným způsobem a to tím, že zamění strukturu části chromozomu v obráceném pořadí.

Stejně jako v případě inverze u jednoduchého genetického algoritmu tak i v případě permutace u genetického programování se ukázalo, že permutace nemá žádný výrazný vliv na průběh algoritmu. [3]

2.7 Operátor editace

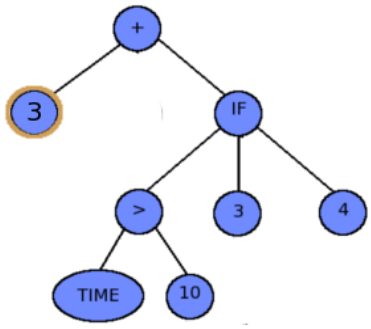
Operátor editace nemá k sobě protějšek v jednoduchém genetickém algoritmu. Smyslem operátoru je zjednodušit stromovou strukturu jedince. Je složen ze sady pravidel, které jsou na jedince uplatněny s cílem zjednodušit řešení nebo odstranit nadbytečné větve stromu, které nemají na řešení žádný vliv. [2]

Průběh operátoru editace zachycují obrázky č. 12 a č. 13.



Obrázek 12 - Operátor editace: zjednodušování stromu. Vytvořeno dle [3].

Odstranění nadbytečných větví má za následek úsporu výpočetního času a zjednodušení stromu je vhodné pokud má být výraz představující řešení dobře čitelný pro uživatele.



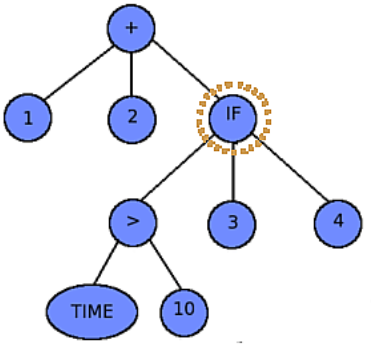
Obrázek 13 - Operátor editace: výsledný zjednodušený strom. Vytvořeno dle [3].

Operátor editace musí být použit na všechny jedince v populaci a jeho použití je řízeno parametrem určující frekvenci jeho uplatnění. Pokud je frekvence rovna 5, pak operátor editace bude uplatňován v každé páté generaci. [3]

Samotný operátor editace má na výsledek algoritmu sporadický vliv. Kozovy pokusy sice ukázaly, že v průběhu algoritmu jedinci, pokud je použit operátor editace, dosahují nepatrně lepších výsledků, nicméně v závěrečné fázi algoritmu byla kvalita jedinců stejná, jak při použití operátoru editace tak i bez něho. [3]

2.8 Operátor zapouzdření

Operátor zapouzdření zapouzdřuje náhodně vybrané větve do podoby jednoho terminálu a tím umožňuje, aby v následných genetických operacích vystupovaly jako jeden celek a nebyly tak genetickými operátory zničeny.

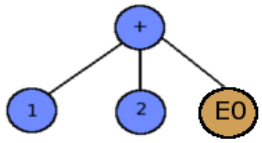


Obrázek 14 - Operátor zapouzdření: výběr funkce. Vytvořeno dle [3]

Operátor zapouzdření náhodně zvolí ve stromu jedince uzel funkce (obrázek č. 14), kterou včetně jejího podstromu nahradí terminálním uzlem (obrázek č. 15). Podstrom nebude smazán, ale bude uchováván spolu s jedincem a terminální uzel na tento podstrom bude odkazovat. Do množiny terminálů definovaných pro daný algoritmus bude také přiřazen tento nový terminál a tak případná mutace může tento terminál vygenerovat a šířit v populaci dále. [3]

Smyslem operátoru zapouzdření je nalézt v populaci “stavební kameny” možného řešení. V programovém kódu, který jedinec reprezentuje, dojde k zapouzdření určité části kódu. Jedná se o pokus nalézt subrutinu a tím dekomponovat složitý programový kód do

několika částí. Následkem zapouzdření se subrutinou musí zacházet jako s celkem. Protože je reprezentována jedním terminálním uzlem, nemůže dojít k tomu, že by byla prostřednictvím genetických operátorů rozdělena. Pokud se subrutina stane dobrým "stavebním kamenem" bude v populaci šířena dále. Koza takovou subrutinu označuje termínem **automaticky definovaná funkce** (automatically defined function). [10]



Obrázek 15 - Operátor zapouzdření: funkce je zapouzdřena do terminálu E0. Vytvořeno dle [3].

2.9 Operátor decimace

Při řešení složitých komplexních problémů často nastává situace, kdy většina jedinců v počáteční populaci má nízkou hodnotu vhodnosti. V důsledku toho velká část výpočetní kapacity je v počáteční fázi algoritmu plýtvána na výpočet vhodnosti jedinců, kteří jsou stejně potenciálnímu řešení příliš vzdáleni, a případné šíření jejich nekvalitních genů v generaci následující bude algoritmus dále zpomalovat.

Cestu k tomu, jak tomuto jevu zabránit, představuje operátor decimace, který přijímá dva argumenty a to číslo generace, v které bude operátor uplatněn, a procento přeživších jedinců. Operátor decimace je uplatněn v určité generaci tím způsobem, že vybere stanovené procento nejlepších jedinců a ti postoupí do generace následující. Ostatní jedinci do následující populace nepostupují a jsou smazáni. Při decimaci je vhodné, aby do následující generace nepostupoval stejný jedinec víckrát, neboli nepostupovali dva jedinci, reprezentující stejné řešení. Ne vždy je ale možné tomu zabránit. [3]

Smyslem operátoru decimace je, že se populace o velkém počtu jedinců zbaví nekvalitních jedinců a bude tak následně prohledávat prostor pouze kolem jedinců, kteří reprezentují dobrá řešení. Počáteční generace by se měla skládat z velkého počtu jedinců, čímž bude nalezen co možná největší počet kvalitních jedinců, kteří po uplatnění operátoru decimace postoupí dále, zatímco nekvalitní jedinci budou smazáni.

Například pokud počáteční generace bude obsahovat 5000 jedinců, pak operátor decimace je možné nastavit takovým způsobem, že hned v první generaci bude do generace následující postupovat pouze 10% z nich a generace následující se bude skládat z 500 jedinců, kteří budou mít výrazně lepší vlastnosti než smazaní jedinci. Algoritmus se pak zabývá prohledáváním prostoru v blízkosti dobrých potenciálních řešení a neplýtvá výpočetní čas prohledáváním řešení, která jsou nevhodná.

2.10 Ukončovací kritérium

Genetické programování napodobuje proces evoluce v přírodě, který je nikdy nekončící proces. Algoritmus však musí být ukončen. Nejčastěji používanou ukončovací podmínkou je, když algoritmus dosáhne určité generace nebo je nalezeno řešení, které splňuje některou na něho kladenou podmínku. Algoritmus dosáhne úspěchu, pokud najde řešení, které splňuje všechny na něho kladené podmínky na 100%. V takovém případě vhodnost počítaná dle vztahu (12) nabude hodnoty 1.

Při řešeních některých velice složitých problémů (složité optimalizační problémy, tvorba matematického modelu na velké množině dat) nepředpokládáme, že by algoritmus nalezl optimální řešení, a proto jako ukončovací podmínku definujeme stav, kdy algoritmus nalezne “pouze” dostatečně dobré řešení. Vhodnost nejlepšího jedince v populaci tedy dosáhne určité požadované hodnoty.

2.11 Doporučené hodnoty parametrů, operátory a metody

Koza navrhl a implementoval do algoritmu genetické operátory, které byly popsány v předchozích kapitolách. Pokud by v algoritmu byly uplatněny všechny Kozou navržené operátory, byl by průběh algoritmu ovlivňován nastavením celkem 19 parametrů. Nalézt správné nastavení tolika parametrů je problém sám o sobě, protože každý z parametrů, pokud je nesprávně nastaven, může mít na průběh algoritmu negativní vliv. Většina počátečních nastavení proto vychází z Kozových pokusů a jeho doporučení hodnot pro jednotlivé parametry.

Níže jsou vypsány Kozou používané hodnoty pro vybrané parametry a použité/nepoužité metody:

- velikost populace, $N = 500$,
- maximální počet generací, $G = 51$ (počáteční populace + 50 následujících),
- pravděpodobnost křížení, $p_c = 0{,}90$,
- pravděpodobnost mutace, $p_m = 0$,
- maximální hloubka stromu po křížení, $h_{max} = 17$,
- maximální hloubka stromu jedince v počáteční generaci, $h_{intial} = 6$,
- metoda generování počáteční populace “ramped half and half”,
- selekce přímo úměrná vhodnosti,
- nepoužíván elitismus.

Je nutno podotknout, že pro každý řešený problém mohou nabývat parametry jiných hodnot. Obzvláště vhodné je měnit některé vybrané parametry během běhu algoritmu a tím reagovat na negativní jevy, které mohou během běhu nastávat jako například uvíznutí v suboptimálním řešení, příliš složitá řešení apod. Mnoho autorů zastává, také rozdílný názor na použití rozličných metod a operátorů.

Někteří autoři publikací používají i zcela jiné genetické operátory. Takovým příkladem je operátor bezhlavého kuřete (beadles chicken operator). Při použití tohoto operátoru je ke křížení použit pouze jeden rodič a druhým rodičem je náhodně vygenerovaný nový jedinec. Potomek je potom zachován jenom v tom případě, když je jeho ohodnocení vyšší nebo stejné než ohodnocení rodiče. Stejně tak mohou být definovány zcela jiné operátory, bez nichž by nebylo možno daný problém ani řešit. [2]

Mnoho autorů udává [2][3], že pokud byly různé metody použity s optimálními parametry, pak všechny metody, které mohly být použity k řešení daného problému, dosáhly podobných výsledků. Neexistuje tedy jednoznačná odpověď na otázku, která metoda je z možných metod na řešení problému nejlepší.

2.12 Možnosti využití genetického programování

Genetické programování jako metoda vycházející z genetického algoritmu dokáže řešit všechny úlohy řešitelné genetickým algoritmem. Některé jsou však genetickým algoritmem řešitelné už velice obtížně a je zapotřebí řešení reprezentovat dlouhým a složitým řetězcem. K přímočařejší reprezentaci řešení je vhodnější, je řešit prostřednictvím genetického programování.

V souhrnu je možné vypsat následující složité problematiky, které je možné genetickým programováním řešit [3]:

- **Optimalizace parametrů** – nalezení optimální hodnoty parametrů ke splnění požadovaných kritérií.
- **Řízení pohybu** – na základě informací přijímaných senzory plánovat optimální cestu prostorem apod.
- **Indukce posloupností** – najít matematické vyjádření posloupnosti na základě zjištěných dat.
- **Symbolická regrese** – nalezení matematického výrazu v jeho symbolické formě, který nám nejlépe proloží daná data. Hledaní vztahu mezi závislou proměnou a nezávislými proměnnými.
- **Automatické programování** – nalezení matematické formule, která na základě vstupu bude vracet požadovaný výstup, přičemž matematickou formuli je možné vidět jako program.
- **Tvorba herní strategie** – nalezení optimální strategie hráče vedoucí k maximalizaci zisku.
- **Empirický výzkum a tvorba předpovědí –** nalezení matematického modelu systému a na základě něho tvořit předpovědi.
- **Nalezení integrálu nebo derivace funkce v symbolické formě** – na základě numerických metod určit derivace (integrace) v bodech a symbolickou regresí je pak vyjádřit v symbolické podobě.
- **Nalezení inverzní funkce** - nalezení matematického výrazu inverzní funkce k dané křivce v symbolické podobě.

• **Nalezení nových matematických vztahů** – hledání nových matematických výrazů a vztahů.

• **Tvorba rozhodovacího stromu** – nalezení matematických vztahů, které na základě popisu objektu ho zařadí do určité kategorie.

• **Řešení krizových situací –** nalezení optimální varianty řešení při omezujících podmínkách, kdy je velký požadavek kladen na rychlost rozhodnutí.

2.13 Příklad umělého mravence

Příklad umělého mravence je klasický demonstrační příklad umělé inteligence, na němž jsou testovány metody na řešení složitých problémů. Cílem řešení je nalézt co nejlepší algoritmus, jak by se měl mravenec pohybovat, aby mohl nalézt všechno jídlo v omezeném prostoru o N x N polí, přičemž ale vidí pouze o jedno pole před sebe. Na tomto příkladě Koza demonstroval genetické programování. Způsob a průběh řešení bude přiblížen v této kapitole. [3]

Prvním krokem k řešení úlohy prostřednictvím genetického programování je definice množiny terminálů a definice množiny funkcí, které budou s těmito terminály pracovat. Musíme také definovat, jakým způsobem budeme hodnotit úspěšnost nalezeného algoritmu.

Na základě spatřeného jídla můžeme ovlivnit rozhodnutí mravence, jakým směrem se vydá. Protože mravenec může vidět jídlo pouze před sebou, definujeme do množiny funkcí jednu funkci, která nám bude vyjadřovat, jakým způsobem se mravenec zachová, pokud před sebou jídlo spatří. Funkce může být vyjádřena následujícím podmíněným výrazem IF-FOOD-AHEAD a bude přijímat dva argumenty, které reprezentují příkazy, jak má mravenec reagovat. První argument bude vyjadřovat, co mravenec má udělat, když před sebou cítí jídlo, a druhý argument bude vyjadřovat, co mravenec má udělat, pokud před sebou jídlo necítí. [3]

Prostřednictvím množiny akcí, které budou reprezentovány množinou terminálů *T*, bude moct mravenec reagovat:

T = {(MOVE), (RIGHT), (LEFT)},

- terminál *MOVE* – mravenec postoupí o jedno pole dopředu,
- terminál *LEFT* – mravenec se otočí doleva,
- terminál *RIGHT* - mravenec se otočí doprava.

Protože možnosti mravence, jak by mohl reagovat na jídlo před sebou, když by mohl vykonat pouze jednu akci, by byly velice omezené, do množiny funkcí budou vloženy ještě dvě funkce: *PRG2* s aritou 2 a *PRG3* s aritou 3. Smyslem těchto funkcí je vytvořit posloupnost příkazů, které mravenec může vykonat.

Funkce vhodnosti bude zcela logicky odrážet počet jídel, která mravenec nalezl při daném algoritmu. Protože může být vygenerován algoritmus, dle kterého by mravenec procházel stále stejným způsobem vymezený prostor stále dokola, musí být doba běhu mravence omezena. Koza omezil dobu výkonu běhu mravence na 400 kroků. Přičemž každá ze tří akcí, která je reprezentována terminály, pro své vykonání spotřebuje jeden celý krok.

Prostor je omezen plochou 32x32 polí a počáteční pozice mravence je na pozici (0, 0). Pokud mravenec opustí omezený prostor, bude do něho vložen na pole u protilehlé stěny prostoru. Pokud tedy opustí prostor z pozice (2, 31), bude mravenec vložen zpět do prostoru na pozici (2, 0).

V počáteční pozici je mravenec natočen čelem k východu. V prostoru je rozmístěno 89 jídel, jejichž pozice je pevně dána. V případě, že by mravenec vykonal všechny příkazy, které mu byly uděleny algoritmem, bude algoritmus vykonán znova. Například když se mravenec bude řídit následujícím algoritmem:

F = (MOVE),

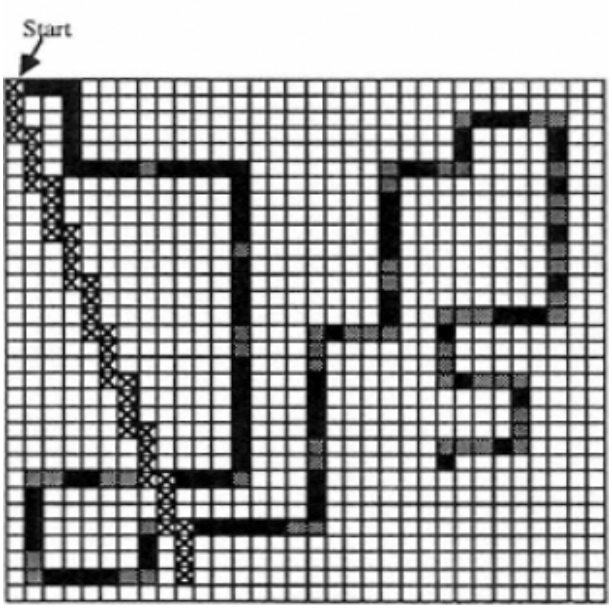
bude se mravenec pohybovat neustále dopředu.

Algoritmus genetického programování pro danou problematiku byl nastaven s parametry, které Koza používal ve většině svých příkladů a bylo uvedeno v předchozí kapitole. (počet jedinců M = 500 a počet generací G=51). [3]

Většina algoritmů, které vznikly v počáteční populaci, nebyla schopna mravence účinně navigovat prostorem. Některé ani nepohnuly mravencem z místa jako například následující algoritmus:

(PROGN2 (RIGHT) (LEFT)).

Mravenec dle tohoto algoritmu se otočil doprava a pak doleva a tuto činnost vykonával neustále dokola.



Obrázek 16 - Pohyb mravence "prošívače". [3]

Jeden algoritmus sice mravence úspěšně navigoval prostorem a nalezl velký počet jídel. Koza tento algoritmus nazval jako “prošívač”, protože způsob, kterým se pohybuje, přirovnával k šití stehů (obrázek č. 16). [3]

Tento algoritmus dělal ale příliš mnoho zbytečných kroků a tak nedokázal během vymezeného počtu kroků nalézt všechna jídla.

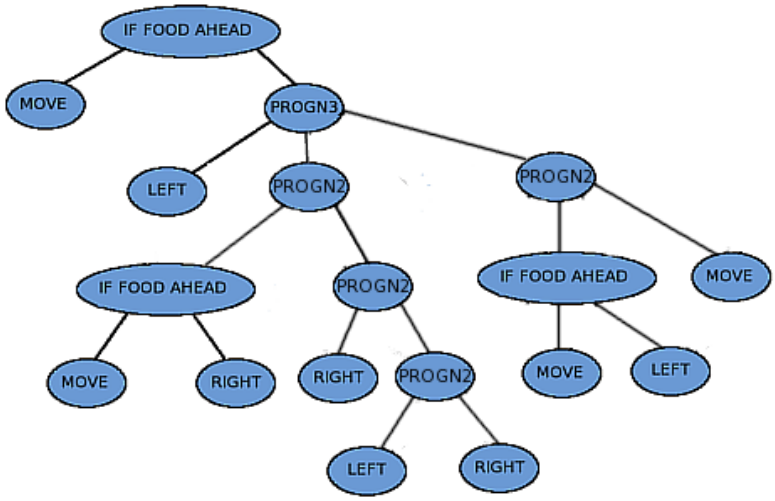
Algoritmus by vyjádřen následujícím výrazem [3]:

(PROGN3 (RIGHT)

(PROGN3 (MOVE) (MOVE)(MOVE)

(PROGN2 (LEFT) (MOVE)).

Každá následující generace jedinců však byla lepší než ta předchozí, i když stále během běhu algoritmu vznikaly algoritmy, které nedokázaly mravence vůbec účinně navigovat prostorem.



Obrázek 17 - Syntaktický strom optimálního řešení.
Převzato a upraveno z [3].

V 21. generaci byl vytvořen jedinec, který představoval optimální řešení. Algoritmus dokázal ve stanoveném čase navigovat mravence tím způsobem, že mravenec nalezl všechno jídlo v prostoru. Strom jedince představující optimální řešení (obrázek č. 17) byl složen z 18 uzlů a reprezentoval následující algoritmus [3]:

```
(IF-FOOD-AHEAD (MOVE)
    (PROGN3 (LEFT)
      (PROGN2 (IF-FOOD-AHEAD (MOVE)
              (RIGHT))
        (PROGN2 (RIGHT)
          (PROGN2 (LEFT)
            (RIGHT))))
      (PROGN2 (IF-FOOD-AHEAD (MOVE)
              (LEFT))
        (MOVE)))
```

Mravenec se pohyboval tím způsobem, že pokud před sebou ucítil potravu, postoupil vpřed a získal jídlo. Pokud potravu před sebou neucítil, nejprve se otočil doleva a otestoval, zda před sebou necítí jídlo. Pokud ano, přesunul se pro jídlo dopředu. Pokud jídlo před sebou neucítil, otočil se dvakrát doprava. Pak vykonal další sekvenci příkazů, kdy se otočil doleva a pak zpátky doprava (zcela zbytečně). Ve výsledku teď mravenec byl natočen doprava oproti

svému výchozímu směru a stále se nacházel na své výchozí pozici. Pokud mravenec teď ucítil před sebou jídlo, postoupil dopředu, jinak se otočil doleva a tím se teď byl natočen do svého výchozího směru a tímto směrem postoupil o jedno pole vpřed.

Mravenec se tedy vlastně pohyboval tím způsobem, že prohledal prostor kolem sebe, přesunul se za případnou potravou a zůstal v tomto směru natočen. Pokud potravu kolem sebe nenalezl, postoupil dopředu. Je zcela zřejmé, že dva příkazy mravenec vykonával zcela zbytečně: *PROGN2((LEFT) (RIGHT))*. Nicméně i přesto mravenec dokázal nalézt všechna jídla ve stanoveném čase a tím požadavky, kladené na optimální řešení splnil. Pokud bylo zapotřebí, aby mravenec nedělal zbytečné kroky, měla být funkce vhodnosti sestavena jiným způsobem. [3]

Koza demonstroval použití genetického programování na řešení dalšího a složitějšího problému umělého mravence, kdy prostor zvýšil na 100x100 polí a do tohoto prostoru umístil 157 návnad jídla pro mravence. Složitosti prostoru musel přizpůsobit nastavení parametrů algoritmu. Počet omezujících kroků, během nichž musí mravenec nalézt potravu, byl zvýšen na 3000, do množiny funkcí byla vložena čtyřaritní funkce *PROGN4 a* populace jedinců byla zvýšena na 2000. V 19. generaci algoritmus našel optimální algoritmus pro navigaci mravence, kdy mravenec nalezl všechny návnady během 1 808 kroků. [3]

Popsané demonstrační příklady dokazují univerzalitu genetického programování, kdy pokud jsou vhodně definovány množiny funkcí, množiny terminálů, dobře definovány operátory, dobře sestavená funkce vhodnosti a složitosti problému jsou přizpůsobeny parametry algoritmu, algoritmus dokáže nalézt optimální řešení.

3 Návrh, tvorba, ověření programu pro genetické programování

Program, jehož tvorba je hlavní náplní diplomové práce, bude realizovat symbolickou regresi. Symbolická regrese je metoda, pomocí které je možné získávat symbolickou podobu funkce, která by nejlépe proložila daná data. Do nedávné doby byla symbolická regrese považována za příliš komplikovaný problém, který vyžaduje příliš sofistikovanou metodu, a nebyl znám žádný algoritmus, který by dokázal dané úlohy řešit. S řešením přišel až John Koza v roce 1992, kdy demonstroval genetické programování na úlohách symbolické regrese a dokázal, že genetické programování je schopno tuto problematiku řešit. [11]

Na základě stanoveného cíle a požadavků na program, byl nejprve vytvořen návrh algoritmu genetického programování, který bude symbolickou regresi realizovat. Pak byl vytvořen objektový model, který bude uchovávat populaci jedinců, kteří reprezentují řešení v podobě syntaktického stromu, a který bude realizovat genetický algoritmus. Následně byl program implementován v jazyce C++ a otestován na příkladech, které jsou uvedené v [2], kde byly řešeny také prostřednictvím genetického programování.

3.1 Stanovený cíl a nezbytné požadavky

Cílem je navrhnout a vytvořit program, který prostřednictvím genetického programování bude na základě vstupní datové matice realizovat symbolickou regresi. Při symbolické regresi musí být použity základní aritmetické operátory (+, -, *, /, ^), trigonometrické funkce (sinus, cosinus), dekadický logaritmus a konstanty. Nezbytným požadavkem je, aby program byl spustitelný pod operačním systémem MS Windows XP a MS Windows VISTA. Dále musí být program snadno rozšiřitelný a z toho důvodu zdrojový kód musí být dostatečně okomentován. Program musí poskytovat dostatečné uživatelské rozhraní, aby z něho bylo možné sledovat průběh algoritmu, a musí poskytovat výstupy o průběhu algoritmu, nalezeném řešení a nastavení algoritmu takovým způsobem, aby výstup bylo možné uložit do datového souboru a použít ho k dalšímu zpracování. Na způsob konkrétní realizace samotného programu (zvolený programovací jazyk, použité knihovny) nejsou kladeny žádné podmínky.

3.2 Návrh technologického řešení

Z důvodu požadavků snadné rozšiřitelnosti bude program sepsán v programovacím jazyce C++. Programovací jazyk C++ patří mezi objektově orientované jazyky a tak další terminály, funkce a komponenty algoritmu mohou být do programu v případě jeho rozšíření snadno přidány odvozením z již existujících tříd nebo rozšířením stávajících tříd o nové metody. Jazyk C++ také umožňuje použitím ukazatelů dynamicky alokovat a uvolňovat paměť během programu, díky čemuž mohou být ušetřeny mnohé paměťové prostředky.

Grafické rozhraní bude realizováno prostřednictvím knihovny Qt 4.4. Jedná se o multiplatformní knihovnu, která je primárně určená k tvorbě grafického uživatelského rozhraní. V současnosti je majitelem licence knihovny Qt společnost Qt Software, která je vlastněna společností Nokia. Nokia zakoupila společnost Qt Software (dříve Trolltech) v roce 2008 s cílem podpořit snazší migraci aplikací pro její mobilní platformu. S příchodem společnosti Nokia došlo k mnohem dynamičtějšímu vývoji a i změně licenční politiky. Je reálné očekávat, že v následujících letech bude vývoj této knihovny dále podporován a nebude hrozit, že v době případného následného rozšíření programu nebude knihovna Qt zastaralá a program nebude možné portovat na nové operační platformy. Knihovna Qt také plně podporuje objektový přístup, je dobře dokumentovaná a má širokou základnu mezi programátory z důvodu jejího snadného použití. [18]

V současnosti (březen 2009) je možné pro použití knihovny Qt si vybrat ze tří možných licencí a to Qt GNU GPL v. 3.0, Qt GNU LGPL v. 2.1 a Qt Commercial Version. Díky tomu, že je knihovna multiplatformní, bude výsledný program zkompilovatelný pro platformy Windows, Linux/X11 a Mac OS X. [18]

3.3 Návrh algoritmu genetického programování pro symbolickou regresi

Prvním krokem při návrhu algoritmu genetického programování pro řešení zadané problematiky symbolické regrese je definovat množinu terminálů. V našem případě množinou terminálů, budou proměnné, které budou reprezentovat příslušné hodnoty v datové vstupní matici. Protože nedokážeme předem říci, kolik proměnných bude v datové matici, musíme

při návrhu programu počítat s tím, že počet proměnných je variabilní a bude se odvíjet od počtu sloupců v datové matici. Kromě proměnných bude možné pro symbolickou regresi použít konstanty, které budou generovány v rozsahu, který bude moct uživatel specifikovat.

Množina terminálů *T* pak bude definována jako:

$T= \{X_{1,}.., X_m, C\}$,

kde *T* je množina terminálů, X_i je *í*-tá proměnná, *m* – počet proměnných a *C* konstanta, které bude při vygenerování přidělena hodnota v určeném rozsahu.

Druhým krokem je definování množiny funkcí. Množina funkcí bude obsahovat základní elementární funkce. Z aritmetických operátorů tak bude množina obsahovat operátor sčítání, odečítání, dělení, násobení a mocniny. Dále bude množina obsahovat trigonometrické funkce sinus, cosinus a množina bude také obsahovat dekadickou logaritmickou funkci. Je také nutné specifikovat aritu funkce. Operátor sčítání, odečítání, dělení, násobení, mocniny budou funkce s aritou dvě a funkce cosinus, sinus a logaritmická funkce budou mít aritu jedna.

Množinu funkcí *F* pak můžeme definovat jako:

F = {+, -, *, /, ^, cos(), sin(), log()}.

Některé funkce nemusí být pro určitou hodnotu definovány (např. dělení nulou) a tak nemohou splňovat podmínku uzavřenosti. Z toho důvodu budou muset funkce být implementovány ve své “chráněné podobě”, v kterých budou muset být tyto situace ošetřeny.

Třetím krokem je definování podoby funkce vhodnosti. V našem případě je možné funkci vhodnosti sestavit na základě součtu chyb odhadu tedy jako součet absolutních rozdílů mezi hodnotou, kterou navrací nalezená funkce, a hodnotou, kterou navrací funkce námi hledaná.

Součet chyb odhadu *í*-tého jedince v populaci pak definujeme následujícím způsobem:

$$e_i = \sum_{j=1}^{N} |y_{i,j} - c_j|, \qquad (14)$$

kde e_i představuje součet chyb odhadu, y_{ij} jsou hodnoty nalezené *í*-té funkce pro *j*-tý případ, c_j hodnota hledané funkce pro j – tý případ a *N* je počet případů.

Z daného výrazu je patrné, že čím bude rozdíl chyb menší, tím bude nalezená funkce lépe prokládat daná data. Optimální řešení bude takové, kdy součet chyb odhadu nalezené *í*-té funkce bude roven nule, $e_i = 0$.

Pro potřeby porovnání s příklady ostatních autorů, kteří také prostřednictvím genetického programování řešili problematiku symbolické regrese a vhodnost potenciálních řešení vyjadřovali v intervalu <0,1>, bude funkce vhodnosti v námi realizovaném programu také nabývat hodnot v intervalu <0,1> a pro optimální řešení pak bude nabývat hodnoty 1.

Funkci vhodnosti pro *í*-tého jedince je pak na základě součtu chyb odhadu možno vyjádřit následujícím výrazem:

$$f_i = \frac{1}{(1+e_i)}, \quad i \in \{1,\ 2,\ ..,\ N\}, \tag{15}$$

kde f_i je vhodnost *í*-tého jedince, e_i je součet chyb *í*-tého jedince spočítaných dle vztahu (14) a N je počet jedinců v populaci.

Dále budeme definovat, jakým způsobem budou realizovány genetické operátory. Pro naše potřeby jako dostačující bude genetický operátor mutace a operátor jednobodového křížení, přičemž výslední jedinci operátorů budou muset splňovat podmínku, která bude omezovat jejich strom na určitý počet uzlů.

Koza omezil prohledávaný prostor obdobným způsobem, aby tak nedošlo k tomu, že by algoritmus prohledával prostor, kde se nacházejí pouze příliš složitá řešení. Na rozdíl od Kozy však složitost řešení nebudeme určovat hloubkou stromu, ale počtem uzlů. Pro operátor mutace bude také definován parametr určující maximální hloubku nově generované větve, jehož hodnotu bude moct uživatel zadat. Musí být také zajištěno, aby ostatní případné genetické operátory mohly být do programu snadno implementovány v případném rozšíření. [3]

Za selekční mechanismus bude použita turnajová selekce a to z důvodu její snazší implementace. Není nutné srovnávat jedince dle pořadí nebo propočítávat proporcionální vhodnost jedince vůči ostatním jedincům v populaci. Turnajová selekce má navíc tu výhodu, že prostřednictvím jejího parametru, který ovlivňuje velikost “ringu”, je možné snadno regulovat selekční tlak, což nám umožňuje měnit průběh algoritmu v případě, kdy uvízne v suboptimálním řešení a potýkáme se s nízkou variabilitou jedinců v populaci.

Pro zvýšení efektivnosti algoritmu bude v programu také implementována penalizace a elitářství. Tyto metody, které rozšiřují genetické programování, budou volitelné a bude záležet na uživateli, zda se rozhodne jich využít. Elitářství bude implementováno velice jednoduše, kdy v každé generaci bude vybrán jeden nejlepší jedinec, který bude automaticky vložen do generace následující, kde nahradí jedince nejhoršího. Penalizace bude mít lineární charakter. Pokud strom jedince překročí určitý počet uzlů, bude penalizován a to tím způsobem, že jeho vhodnost bude zhoršena úměrně k počtu uzlů, které strom obsahuje nad povolený limit.

Penalizaci použitou v programu je možné vyjádřit následujícím vztahem:

$$ep_i = e_i + e_i.(u_{max} - u_i).c_p, \quad i \in \{1, 2, .., N\}, \qquad (16)$$

kde ep_i je penalizovaný součet chyb, e_i je součet chyb odhadu *í*-tého jedince spočítaných dle vztahu (14), u_{max} je maximální povolený počet uzlů, u_i je počet uzlů *í*-tého jedince a c_p je penalizační koeficient.

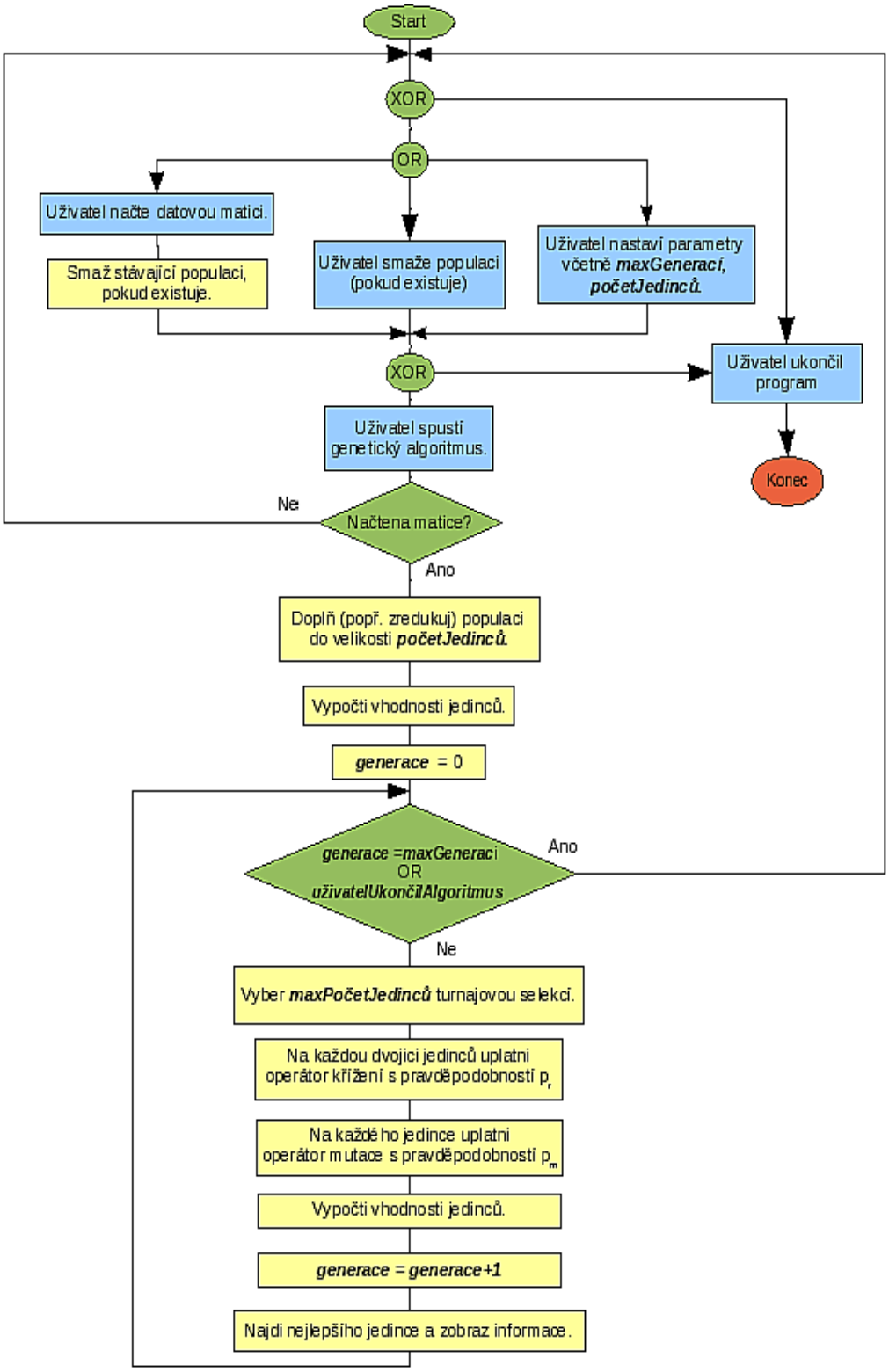
Pokud strom nepřekročí maximální počet povolených uzlů, penalizace nebude uplatněna. Hodnotu penalizačního koeficientu a maximální počet uzlů pro penalizaci bude moct stanovit uživatel a bude moct tak ovlivnit “šanci” pro složitá ale lepší řešení, že budou moct postoupit do následující generace.

Pro počáteční generaci bude moct uživatel definovat maximální hloubku nově generovaného jedince a také bude moct definovat, s jakou pravděpodobností bude generován uzel určité funkce, uzel proměnné a uzel konstanty. Tyto pravděpodobnosti budou závazné i pro operátor mutace, který dle nich bude generovat novou větev stromu. Bude také možné definovat rozsah hodnot, kterých budou moct konstanty nabývat, přičemž konstanty budou generovány s náhodnou hodnotou ležící v tomto intervalu.

Celý základní algoritmus programu je zachycen na obrázku č. 18. Po spuštění uživatel může načíst vstupní datovou matici a může změnit nastavení parametrů algoritmu. Dokud nebude ale vstupní datová matice načtena, nebude moct být zcela logicky genetický algoritmus spuštěn. V případě, že matice načtena je a uživatel spustí genetický algoritmus, program vykoná tolik cyklů genetického algoritmu, kolik uživatel specifikoval. Po každém

cyklu budou zobrazeny informace o nejlepším nalezeném řešení v dané generaci. Cyklus bude zastaven poté, co ho zastaví sám uživatel nebo pokud cyklus dosáhne počtu generací, které uživatel definoval.

Uživatel bude moct po zastavení běhu genetického algoritmu aktuální populaci smazat a pak při opětovném spuštění algoritmu bude v prvním kroku vygenerována nová počáteční populace, nebo uživatel může populaci ponechat a v tom případě genetický algoritmus bude pokračovat s populací stávající. Pokud uživatel po zastavení algoritmu změní počet jedinců v populaci a nesmaže stávající populaci, pak počet jedinců v populaci bude snížen tím způsobem, že z populace budou odstraněni nejhorší jedinci. V případě, že populaci je nutné doplnit do definovaného počtu, budou do populace dogenerování jedinci noví.



Obrázek 18 - Vývojový diagram zachycující základní algoritmus programu. [Zdroj vlastní]

3.4 Návrh rozhraní

Vstupem programu bude datová matice a výstupem informace o nejlepším nalezeném řešení a o průběhu algoritmu. Vstupní datová matice bude uložena v textovém souboru, který bude mít formát CSV[6]. Oddělovače řádků a sloupců bude moct uživatel definovat při načítání datové matice z textového souboru. První řádkový vektor matice bude obsahovat názvy proměnných, zbylé řádkové vektory budou představovat jednotlivé případy. Výsledek funkce, jejíž výraz hledáme, se bude nacházet v libovolném sloupci, jehož číslo bude moct definovat uživatel. Jinak se bude předpokládat, že tímto sloupcem je sloupec první.

Při načítání matice je třeba také ošetřit situace, kdy hodnoty matice nebudou za oddělovač desetinných míst používat tečku ale čárku.

Základní grafické uživatelské rozhraní musí poskytovat možnosti nastavení parametrů, které ovlivňují průběh algoritmu. Kromě zobrazení nastavení algoritmu je třeba také vhodnou formou zobrazit informace o průběhu algoritmu. Zcela logicky bude rozhraní obsahovat graf, který bude zobrazovat průběh algoritmu při hledání optimálního řešení a vhodnost nejlepšího jedince pro danou generaci. Rozhraní bude obsahovat také další informace o nejlepším dosud nalezeném řešení – průměrná chyba, průměrná čtvercová chyba, součet chyb odhadu apod. Některé ukazatele pro každou generaci budou vkládány do tabulek, které budou poskytovat informace o průběhu algoritmu. Tyto tabulky se stanou základem výstupu, který bude moct být uložen do textového souboru v podobě datové matice.

Bude také vhodné zobrazit strom nejlepšího jedince v populaci. Implementace takového zobrazení je ale velice složitá z důvodu, že zobrazené větve stromu se nesmí překrývat apod. Vzhledem k složitosti zobrazení stromu budou k zobrazení využity možnosti třídy Qt, která umožňuje zobrazit strom v určité formě. Nicméně toto zobrazení není pro naši problematiku syntaktického stromu dostatečně přehledné.

Celé grafické uživatelské rozhraní bude uspořádáno do několika oken, mezi kterými bude moct být přepínáno. Přepínání oken bude realizováno prostřednictvím záložek

6 **CSV** (Comma-separated values) - jednoduchý souborový formát určený pro výměnu datových matic. Soubor ve formátu CSV je složen z řádků, ve kterých jsou jednotlivé položky matice odděleny čárkou [9] . Čárka jako oddělovač není však vždy vhodná a tak se používají i jiné oddělovací znaky.

umístěných v horní části okna a grafické spouštěče případných akcí jako uložení výstupu do souboru, načtení matice apod. se budou nacházet v příslušných oknech, do kterých logicky zapadají dle jejich významu. Například tlačítko, po jehož kliknutí bude vyvolán dialog pro načítání datové matice, bude v okně Data.

Program bude obsahovat celkově 5 oken:

- okno *genetický algoritmus* – obsahuje možnosti nastavení algoritmu,
- okno *funkce nejlepšího jedince* – zobrazuje výraz nejlepšího řešení,
- okno *strom jedince* – zobrazuje syntaktický strom nejlepšího řešení,
- okno *záznam průběhu algoritmu* – zobrazuje tabulku o průběhu algoritmu,
- okno *data* – zobrazuje datovou matici.

Bližší upřesnění jednotlivých oken se nachází v uživatelské příručce (příloha A).

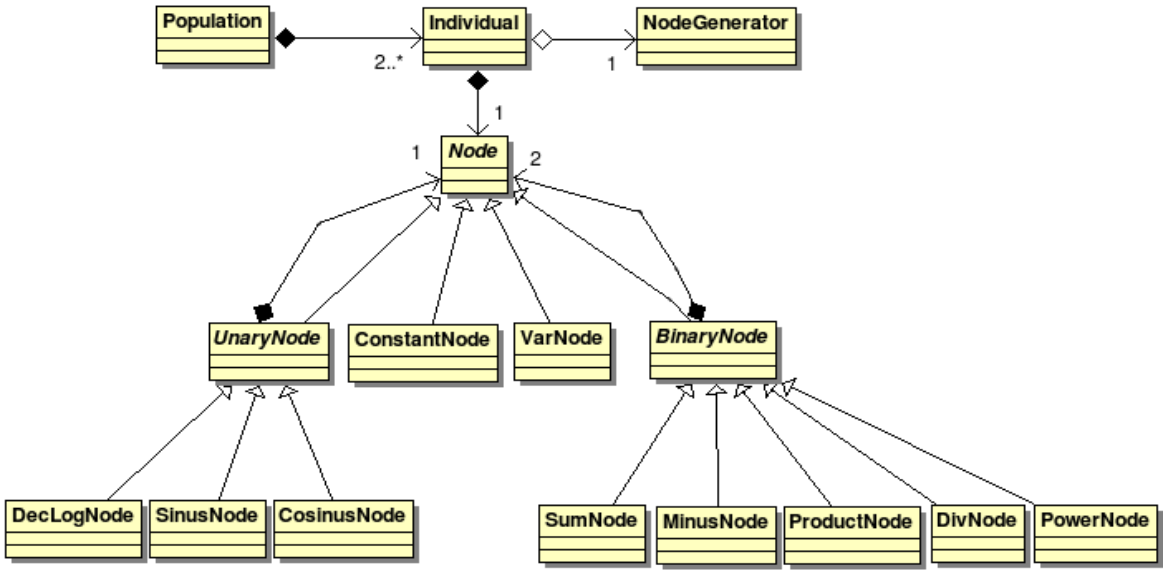
3.5 Návrh objektového modelu

Tato kapitola popisuje ve stručnosti navržený a implementovaný objektový model. Vzhledem ke složitosti navrženého modelu budou popsány pouze základní třídy a jejich vybrané metody. Třídy realizující grafické rozhraní budou v tomto popisu opomíjeny. Pro bližší informace o implementaci programu je vhodné shlédnout okomentovaný zdrojový kód programu na přiloženém CD, které je součástí diplomové práce.

Pro realizaci samotného algoritmu genetického programování byly identifikovány následující třídy:

- třída Node a třídy odvozené od třídy Node,
- třída NodeGenerator,
- třída Individual,
- třída Population,

které budou ve stručnosti přiblíženy. Objektový model je zachycen na diagramu tříd (obrázek č. 19).



Obrázek 19 - Diagram tříd objektového modelu. [Zdroj vlastní]

Třída Node a od ní odvozené třídy

Potenciální řešení budou reprezentovaná prostřednictvím syntaktického stromu, který je složen z uzlů a hran, a právě uzel syntaktického stromu bude v námi navrženém objektovém modelu reprezentován třídou Node. Třída Node zapouzdřuje také metody, které jsou nutné k manipulaci s podstromem, který následuje po daném uzlu. Pokud daný uzel bude kořenem stromu, bude tento uzel tedy zastupovat celý strom.

Samotná třída Node je definována jako abstraktní. Není tedy možné vytvořit její instanci. Z třídy Node jsou odvozeny třídy – BinaryNode, UnaryNode, VarNode a ConstantNode. Z tříd UnaryNode a BinaryNode jsou pak dále odvozeny třídy reprezentující zadané funkce tedy třídy SumNode, MinusNode, ProductNode, DivNode, PowerNode, CosinusNode, SinusNode a DecLogNode. Každá třída má některé metody definovány jiným způsobem. Třídy reprezentující uzly funkcí mají potomky přiděleny dle jejich arity. Třída BinaryNode si musí tedy uchovávat adresy na dva své potomky, třída UnaryNode na jednoho potomka. Terminální uzly (VarNode a ConstantNode) nemohou mít potomky přiděleny. Celý strom je pak možné zastupovat instancí třídy odvozené od třídy Node, která dle svého typu odkazuje na případné potomky a poskytuje metody, pomocí nichž je možné s daným stromem manipulovat a měnit jeho strukturu.

Třída Node poskytuje následující vybrané metody:

- *int getCountNodes()* - vrací počet uzlů stromu,
- *string getTextFormula()* - vrací textovou podobu funkce stromu,
- *int getType()* - vrací typ uzlu,
- *string getFuncName()* - vrací název uzlu,
- *Node *getLeftChild()* - vrací adresu levého potomka,
- *Node *getRightChild()* - vrací adresu pravého potomka,
- *Node *getChild()* - vrací adresu potomka (v případě unárního uzlu),
- *Node *clone()* - vytvoří kopii stromu a vrátí jeho adresu,
- *Node **findNode(int nodeNumber)* - vrací adresu ukazatele na uzel ve stromu, který je identifikován svým číslem,
- *float getResult(const int &i)* – vrací výsledek výrazu uzlu pro *i*-tý případ.
- *~Node()* - destruktor třídy, který smaže i případné potomky.

Třída NodeGenerator

Třída NodeGenerator generuje uzly na základě jejich stanovené pravděpodobnosti a nastavuje některé proměnné generovaným uzlům (adresy na vektor obsahující hodnoty proměnné, kterou uzel reprezentuje apod.). Prostřednictvím jejího rozhraní lze nastavit pravděpodobnosti generování jednotlivých typů uzlů a rozsah generovaných konstant.

Třída Individual

Třída Individual zajišťuje reprezentaci jedince a poskytuje metody, prostřednictvím kterých je možné zjišťovat informace o kvalitě potenciálního řešení, který je jedincem zastupován. Třída Individual obsahuje ukazatel na instanci třídy odvozené od třídy Node, která je kořenem stromu jedince. Také poskytuje metody *crossOver()* a *mutation()*, které aplikují příslušný genetický operátor na daného jedince. Konstruktor třídy je zodpovědný za tvorbu počátečního stromu. Uzly stromu jsou generovány prostřednictvím třídy *NodeGenerator*.

Vybrané metody třídy Individual:

- *float getFitness()* - navrací vhodnost jedince,
- *string getTextFormula ()* - navrací textovou podobu výrazu, který jedinec reprezentuje,
- *Individual *clone()* - vytvoří kopii jedince a vrátí jeho adresu,
- *Node *getFirstNode()* - vrací adresu kořenového uzlu stromu,
- *int getCountNodes()* - vrací počet uzlů stromu,
- *void crossOver()* - provede křížení s vybraným jedincem,
- *void mutation()* - provede mutaci jedince.

Třída Population

Třída population zapouzdřuje metody nutné pro manipulaci s populací. Třída si uchovává adresy na každého jednotlivce ve vektoru ukazatelů. Realizuje samotný algoritmus genetického programování prostřednictvím metody *tournamentReproduction()*, generuje jedince a prostřednictvím rozhraní zpřístupňuje nejlepšího jedince třídám grafického rozhraní.

Vybrané nejdůležitější metody:

- *void tournamentSelection()* - stávající generaci nahradí novou dle turnajové selekce,
- *void tournamentReproduction()* - provede definovaný počet cyklů (generací) genetického algoritmu při použití turnajové selekce.
- *void setPenalization()* - nastaví zapnutí penalizace,
- *void setElitismus()* - nastaví zapnutí elitismu,
- *Individual *getBestIndiv()* - vrátí adresu nejlepšího jedince v populaci.

3.6 Implementace

Program byl dle navrženého objektového modelu implementován v jazyce C++ s využitím knihovny Qt 4.4. Vzhledem k velkému množství parametrů je implementace grafického rozhraní poměrně složitá. Jako generátor pseudonáhodných čísel byl v programu

použit SIMD-oriented Fast Mersenne Twister (SFMT) generátor. Dle [8] se jedná o generátor, který představuje dobrý kompromis mezi rychlostí a požadovaným rovnoměrným rozložením, ke kterému se blíží.

První verze programu trpěly únikem paměti. K chybě docházelo při vytváření kopie jedince v metodě *Individual::clone()*, kdy předtím než novému jedinci byl přidělen zkopírovaný strom kopírovaného jedince, jeho implicitní konstruktor vygeneroval jedinci vlastní strom. Adresa na něho však po přidělení zkopírovaného stromu byla ztracena. Chyba byla nalezena a odstraněna. Žádné další implementační chyby nejsou známy.

3.7 Testování programu na vzorových příkladech symbolické regrese

Program byl testován na dvou vzorových příkladech, na nichž je v [2] demonstrován postup při řešení úloh prostřednictvím genetického programování. Jedná se o dva jednoduché příklady na měření odporu v elektrickém obvodu. Protože příklady byly již řešeny v [2] prostřednictvím genetického programování a průběh genetického programování při řešení daných úloh byl podrobně popsán, máme dobrou možnost srovnání.

První příklad byl nejprve programem řešen při výchozím nastavením parametrů a poté byl řešen při optimálním nastavení parametrů. Druhý příklad je o něco složitější a byl řešen pouze s optimálním nastavením parametrů.

3.7.1 První testovací příklad

Zadání prvního příkladu dle [2] je následující. Máme elektrický obvod, který obsahuje dva rezistory zapojené paralelně. Na základě Kirchhoffova zákona[7] a Ohmova zákona[8] lze snadno odvodit formuli pro celkový odpor *R*:

$$R = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}} = \frac{R_1 . R_2}{R_1 + R_2}, \qquad (17)$$

kde *R* je celkový odpor obvodu, R_1 odpor prvního rezistoru a R_2 odpor druhého rezistoru.

Předpokládejme, že tento vzorec ještě nebyl objeven, že neznáme ani jeden z výše uvedených zákonů a jediné, co máme k dispozici, jsou výsledky měření při experimentech s různými hodnotami rezistorů R_1 a R_2. Na základě dat získanými příslušnými experimenty se s využitím genetického programování pokusíme výše uvedený vzorec "objevit".

Celkově bylo provedeno 12 měření (tabulka č. 4), při nichž byly kombinovány různé hodnoty paralelně zapojených rezistorů. Při všech experimentech byl použit zdroj zajišťující konstantní napětí 10 V. Naším cílem je prostřednictvím symbolické regrese nalézt funkci v symbolickém tvaru, která by nejlépe aproximovala vztah mezi nezávislými proměnnými R_1, R_2 a závislou proměnnou *R*.

Tabulka 4 - Vstupní datová matice naměřených hodnot.[2]

Číslo měření	R1	R2	R
1	40	10	**8**
2	5	5	**2,5**
3	10	40	**8**
4	6	4	**2,4**
5	4	8	**2,66**
6	1	4	**0,8**
7	20	20	**10**
8	5	20	**4**
9	4	6	**2,4**
10	8	12	**4,8**
11	40	10	**8**
12	15	5	**3,75**

7 **Druhý Kirchhoffův zákon** - v uzavřeném obvodu, který vyčleníme v rozvětvené síti, se součet napětí na jednotlivých rezistorech rovná součtu elektromotorických napětí jednotlivých zdrojů. [20]

8 **Ohmův zákon** - napětí mezi konci vodiče je přímo úměrné proudu procházejícímu vodičem. [14]

Příklad byl nejprve řešen s nastavením parametrů algoritmu, kdy o charakteru řešení nemáme dostatek informací, a pak s optimálním nastavením parametrů, kdy předpokládáme určitou podobu výrazu. Bude tak demonstrován rozdíl, jak program bude realizovat symbolickou regresi při optimálním a neoptimálním nastavením.

Hledání optimálního řešení při použití všech možných funkcí a terminálů.

Nejprve předpokládejme, že nemáme žádnou představu o tom, jaké podoby by měla funkce nabývat, a proto jako množinu možných funkcí použijeme všechny funkce, které program nabízí. Základní funkce jako aritmetické operátory (násobení, dělení, sčítání, odečítání) budou použity s vyšší pravděpodobností. Méně časté funkce (sinus(), cosinus(), log(), mocnina) pak s nižší pravděpodobností. Program bude také náhodně generovat konstanty v rozsahu <-10;10>.

Celkové nastavení algoritmu:

Pro generaci: 1-500

Velikost ringu: 3

Pravděpodobnost výhry lepšího: 95%

Pravděpodobnost mutace: 3%

Max. hloubka mutace: 3

Pravděpodobnost křížení: 95%

Počet jedinců: 200

Počáteční hloubka stromu: 4

Maximální počet uzlů: 100

Pravděpodobnost uzlu proměnné: 20%

Pravděpodobnost uzlu konstanty: 12%

Pravděpodobnost uzlu sčítání: 12%

Pravděpodobnost uzlu odečítání: 12%

Pravděpodobnost uzlu součinu: 12%

Pravděpodobnost uzlu dělení: 12%

Pravděpodobnost uzlu mocniny: 5%

Pravděpodobnost uzlu sinus: 5%

Pravděpodobnost uzlu cosinus: 5%

Pravděpodobnost uzlu dekadického logaritmu: 5%

Není použit elitismus.

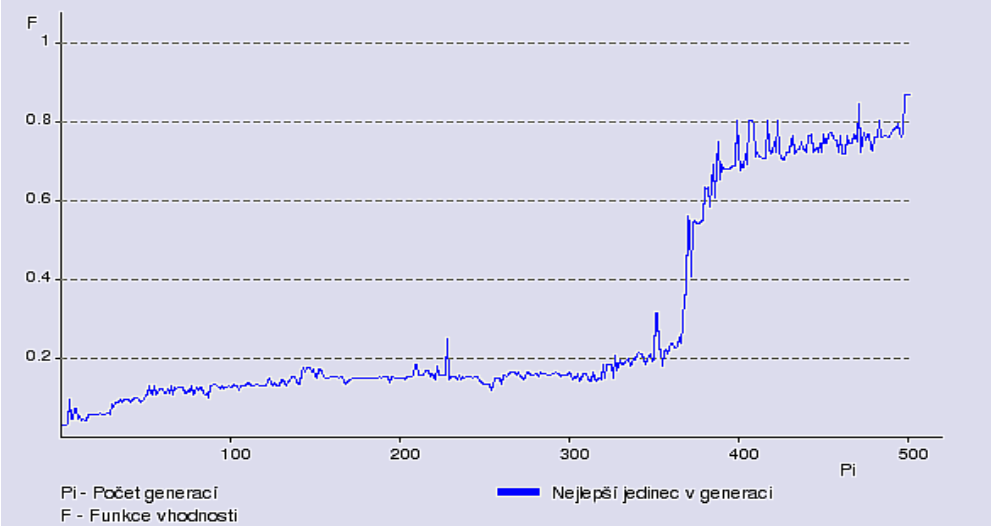
Penalizace je použita.

Max. počet uzlů pro penalizaci: 30

Penalizační koeficient: 0,05

Konstanty generovány v rozsahu: -10; 10

Průběh hledání optimálního řešení při daném nastavení je zachyceno na grafu č. 3. Po 500 generací bylo jako dosud nejlepší řešení označeno řešení s vhodností 0,869415, jehož funkce je složena z 53 uzlů, průměrná odchylka nalezené funkce od funkce hledané (průměrná chyba) je 0,005822 a průměrná čtvercová odchylka je 0,000074.



Graf 3 - Průběh hledání optimálního řešení při použití všech možných funkcí a terminálů. [Grafické rozhraní programu]

Nalezená funkce je definována následujícím výrazem:

((((3.516)/(*R2*))/(((3.516+(3.516+3.516))+3.516)))/(*R2*)+((((((3.516)/(3.516))/(3.516))/(3.516))/(*R2*))/(3.516)+(*R1*)/(((((((3.516)/((*R2*+(3.516+3.516))))/(3.516))/(*R2*))/(((3.516)/(3.516)+3.516))+(*R1*)/(*R2*))+(3.516)/(3.516))))).

Z tabulky č. 5 je patrné, že nalezená funkce odhaduje *R* poměrně dobře. Optimální řešení ale při těchto nastaveních do 500. generace nebylo nalezeno a nalezená funkce je poměrně složitá. Editací by bylo možno funkci dále zjednodušit a odstranit nadbytečné výrazy. Také by bylo vhodné optimalizovat numerické parametry funkce.

Tabulka 5 - Výsledná datová matice včetně odhadu R (Rozdíl představuje chybu odhadu).
[Upraveno z výstupní datové matice programu]

R1	R2	R	R – odhad	Rozdil
40	10	8	8,002721	-0,002721
5	5	2,5	2,510009	-0,010009
10	40	8	7,999978	0,000022
6	4	2,4	2,416569	-0,016569
4	8	2,67	2,670179	-0,000179
1	4	0,8	0,818178	-0,018178
20	20	10	9,999728	0,000272
5	20	4	4,000465	-0,000465
4	6	2,4	2,406708	-0,006708
8	12	4,8	4,800863	-0,000862
40	10	8	8,002721	-0,002721
15	5	3,75	3,761154	-0,011154

Hledání optimálního řešení při použití optimálního nastavení algoritmu.

Nyní předpokládejme, že máme dobrý odhad a definujeme minimální množinu terminálů a funkcí, které potřebujeme k nalezení optimální funkce [2]:

$T = \{R1, R2\}, F = \{*, /, +\}$,

kde T je množina terminálů, *R1* proměnná reprezentující hodnoty odporu naměřeného na prvním rezistoru, *R2* proměnná reprezentující naměřené hodnoty odporu na druhém rezistoru a F množina funkcí.

Funkce a terminály budou generovány se stejnou pravděpodobností. Počet jedinců v populaci bude snížen a maximální počet uzlů, od kterého bude zahájena penalizace, bude nastaven na menší hodnotu 5 (budou více upřednostňována jednodušší řešení).

Celkové nastavení algoritmu je pak následující:

Pro generaci: 1-500

Velikost ringu: 3

Pravděpodobnost výhry lepšího: 95%

Pravděpodobnost mutace: 5%

Max. hloubka mutace: 3

Pravděpodobnost křížení: 95%

Počet jedinců: 50

Počáteční hloubka stromu: 5

Maximální počet uzlů: 100

Pravděpodobnost uzlu proměnné: 25%

Pravděpodobnost uzlu konstanty: 0%

Pravděpodobnost uzlu sčítání: 25%

Pravděpodobnost uzlu odečítání: 0%

Pravděpodobnost uzlu součinu: 25%

Pravděpodobnost uzlu dělení: 25%

Pravděpodobnost uzlu mocniny: 0%

Pravděpodobnost uzlu sinus: 0%

Pravděpodobnost uzlu cosinus: 0%

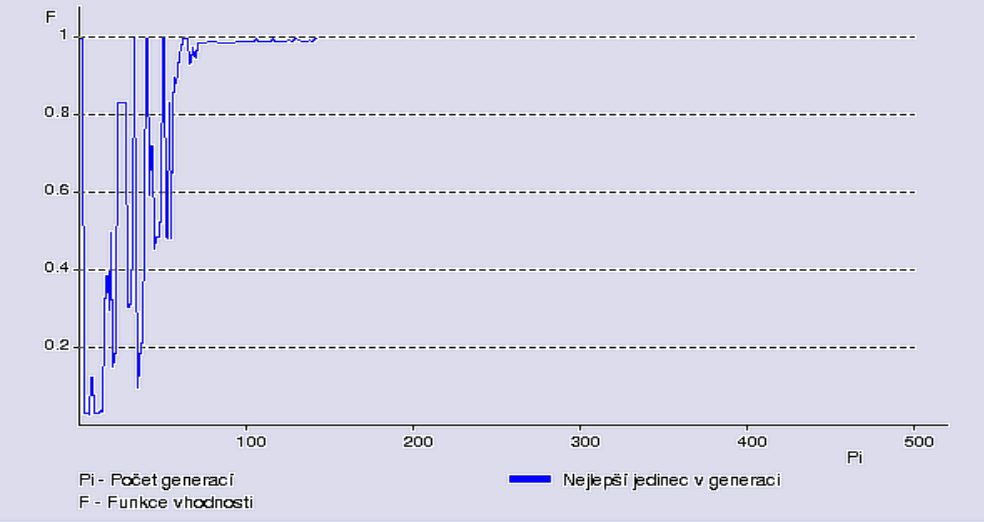
Pravděpodobnost uzlu dekadického logaritmu: 0%

Elitismus není použit.

Penalizace je použita.

Max. počet uzlu pro penalizaci: 5

Penalizační koeficient: 0,05



Graf 4 - Průběh hledání nejlepšího řešení při optimálním nastavením parametrů. [Grafické rozhraní programu]

Na grafu č. 4 je zaznamenán průběh algoritmu hledání optimálního řešení. Z grafu je patrné, že optimální řešení bylo nalezeno hned v počáteční populaci, ale protože není použit

elitismus a není uplatňován vysoký selekční tlak, bylo řešení v následujících generacích ztraceno. Nicméně algoritmus postupným zlepšováním populace optimální řešení znovu nalezl v 33. generaci a pak se pohyboval již pouze v okolí tohoto řešení.

Vzhledem k tomu, že byla uplatňována penalizace od malého počtu uzlů, bylo zabráněno tomu, aby jako optimální řešení byla nalezena funkce s vysokým počtem uzlů a tak zbytečně složitým výrazem.

Za optimální řešení je považováno řešení s počtem 7 uzlů a výrazem:
((*R1*R2*))/((*R1+R2*)).

Nalezenému řešení přísluší vhodnost s hodnotou 0,996313 a průměrnou odchylkou od požadované hodnoty 0,000278. Ačkoliv se nalezený výraz shoduje s hledanou funkcí, vhodnost řešení není rovna 1. Důvodem této skutečnosti je, že hodnoty pro R v datové matici byly zaokrouhlovány a tak pro 5. experiment (zvýrazněn v tabulce č. 6) rozdíl mezi funkcí optimálního řešení a hodnotou pro *R* činní 0,003333. Tento příklad je ukázkou, že ne vždy musí optimální řešení nabýt vhodnosti s hodnotou 1, protože někdy mohou být data zatížena chybou měření, či byla zkreslena manipulací.

Tabulka 6 - Výsledná datová matice včetně odhadu R při optimálním nastavení parametrů.
[Upraveno z výstupní datové matice programu]

R1	R2	R	R – odhad	Rozdíl
40,00	10,00	8,00	8,0000	0,0000
5,00	5,00	2,50	2,5000	0,0000
10,00	40,00	8,00	8,0000	0,0000
6,00	4,00	2,40	2,4000	0,0000
4,00	**8,00**	**2,67**	**2,6667**	**0,0033**
1,00	4,00	0,80	0,8000	0,0000
20,00	20,00	10,00	10,0000	0,0000
5,00	20,00	4,00	4,0000	0,0000
4,00	6,00	2,40	2,4000	0,0000
8,00	12,00	4,80	4,8000	0,0000
40,00	10,00	8,00	8,0000	0,0000
15,00	5,00	3,75	3,7500	0,0000

3.7.2 Druhý testovací příklad

Vycházíme ze zadání prvního příkladu, ale tentokrát navíc budeme uvažovat i proudy tekoucí příslušnými větvemi elektrického obvodu. Vstupní datová matice (tabulka č. 7) tak bude rozšířena o další dvě proměnné I_1 a I_2 reprezentující velikost proudů v jednotlivých větví.

Tabulka 7 - Vstupní datová matice naměřených hodnot druhého příkladu. [2]

Číslo měření	R1	R2	I1	I2	R
1	40	10	0,25	1	**8**
2	5	5	2	2	**2,5**
3	10	40	1	0,25	**8**
4	6	4	1,66	2,5	**2,4**
5	4	8	2,5	1,25	**2,66**
6	1	4	10	2,5	**0,8**
7	20	20	0,5	0,5	**10**
8	5	20	2	0,5	**4**
9	4	6	2,5	1,66	**2,4**
10	8	12	1,25	833	**4,8**
11	40	10	0,25	1	**8**
12	15	5	0,66	2,5	**3,75**

3.7.3 Průběh řešení druhého příkladu

Definujeme množinu terminálů:

T = {*R1*, *R2*, *I1*, *I2*},

kde *T* je množina terminálů, *R1* proměnná reprezentující hodnoty odporu naměřeného na prvním rezistoru, *R2* proměnná reprezentující naměřené hodnoty odporu na druhém rezistoru, *I1* proměnná reprezentující velikost proudu v první větvi a *I2* velikost proudu v druhé větvi.

Množinu funkcí *F* obohatíme o operaci odčítání (postupujeme stejně jako v [2]):

F = {*, /, +, -}.

Vzhledem k tomu, že předpokládáme jako optimální řešení složitější výraz, bude této skutečnosti přizpůsobeno nastavení algoritmu. Bude zvolen vyšší počet jedinců a jedinci budou penalizováni za vyšší počet uzlů.

Celkové nastavení algoritmu:

Pro generaci: 1-159
Velikost ringu: 3
Pravděpodobnost výhry lepšího: 95%
Pravděpodobnost mutace: 5%
Max. hloubka mutace: 5
Pravděpodobnost křížení: 95%
Počet jedinců: 300
Počáteční hloubka stromu: 4
Maximální počet uzlů: 100
Pravděpodobnost uzlu proměnné: 20%
Pravděpodobnost uzlu konstanty: 0%
Pravděpodobnost uzlu sčítání: 20%
Pravděpodobnost uzlu odečítání: 20%
Pravděpodobnost uzlu součinu: 20%
Pravděpodobnost uzlu dělení: 20%
Pravděpodobnost uzlu mocniny: 0%
Pravděpodobnost uzlu sinus: 0%
Pravděpodobnost uzlu cosinus: 0%
Pravděpodobnost uzlu dekadického logaritmu: 0%
Není použit elitismus.
Penalizace je použita.
Max. počet uzlů pro penalizaci: 20
Penalizační koeficient: 0.01

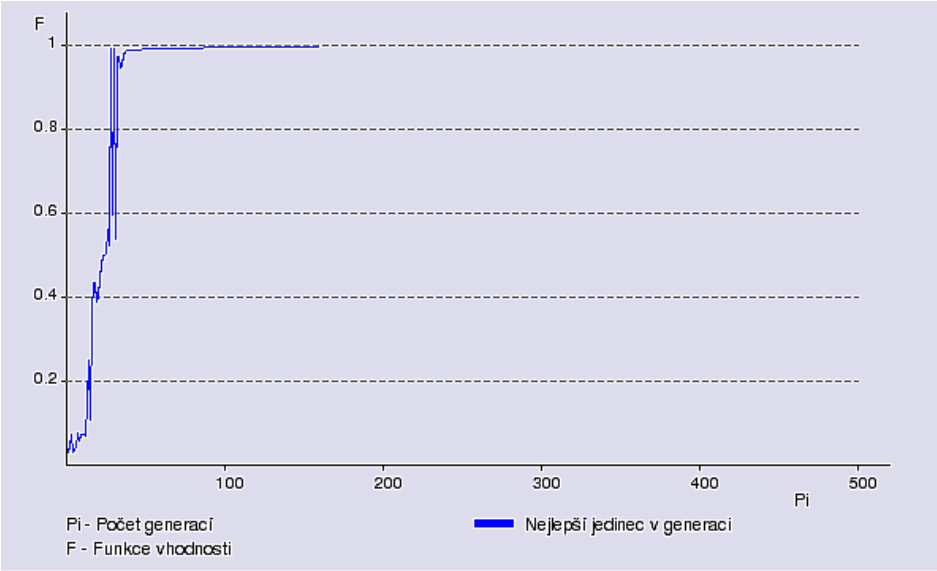
Průběh algoritmu je zachycen na grafu č. 5. Z grafu je patrné, že algoritmus poměrně rychle konvergoval k optimálnímu řešení. Samotné optimální řešení ale nalezl až v 97. generaci a do té doby se pohyboval v jeho okolí.

Nalezeným optimálním řešením je následující výraz:
((((*I1***R2*)**R1*))/(*I1*))/(((*R1*+*R2*)+(((*I2*-*I2*))/(*I1*))/(*R1*)))
s počtem 19 uzlů, vhodností 0,99667 a průměrnou odchylkou 0,000278.

Pokud získaný výraz zjednodušíme, dojdeme opět k výrazu:

(*R1* * *R2*)/(*R1* + *R2*),

který jsme získali již v předcházejícím příkladě.



Graf 5 - Průběh hledání optimálního řešení při měření velikosti proudů. [Grafické rozhraní programu]

Jak je vidět, velikost proudu v jednotlivých větvích elektrického obvodu není pro určení velikosti celkového odporu nutné měřit.

Prostřednictvím programu se nám opět podařilo nalézt optimální funkci, která nám co nejlépe proloží daná data.

3.7.4 Závěr testování

Program na těchto příkladech dokázal, že je schopen řešit úlohy symbolické regrese. Testování na těchto příkladech bylo několikrát opakováno vždy s podobným průběhem genetického algoritmus. Pokud bylo použito optimální nastavení parametrů, algoritmus vždy nalezl optimální řešení. Testování na příkladech dokázalo, že úspěšnost nalezení optimálního řešení je silně závislá na nastavení algoritmu. Také je výhodné během realizace algoritmu jeho nastavení měnit. V případě, že nemáme představu o tom, jak složité podoby by měl hledaný výraz nabývat, je vhodné zpočátku omezit jedince na nižší počet uzlů prostřednictvím

parametrů penalizace a maximálního počtu uzlů jedince a pokud algoritmus po určité delší době nedokázal najít dostatečně dobré řešení je vhodné parametry změnit, připustit složitější řešení a pokračovat algoritmem se stávající populací.

3.8 Návrhy na rozšíření programu

Program sice umožňuje řešit symbolickou regresi prostřednictvím základních elementárních funkcí, nicméně jsou zde určité možnosti jak činnost programu zefektivnit například implementací dalších genetických operátorů. Vzhledem k tomu, že je program napsán jako objektově orientovaný, rozšíření programu je snadné a lze ho realizovat prostřednictvím implementace nových metod nebo odvozením nových tříd z tříd stávajících.

Návrhy na rozšíření jsou následující:

- implementace dalších selekčních mechanismů, tak že uživatel si bude moct zvolit, jaký selekční mechanismus využije,
- implementace dalších funkcí zejména podmíněných výrazů,
- zdokonalení použití konstant v programu, tak aby program dokázal najít jejich optimální numerickou hodnotu,
- implementace dalších genetických operátorů jako operátor editace a operátor zapouzdření,
- zdokonalit zobrazení syntaktického stromu,
- možnost definování vlastní funkce a její vložení do počáteční populace.

Pro případné rozšíření programu je k diplomové práci přiložena stručná programátorská dokumentace (příloha B), kde je naznačen způsob, jakým je možno program dále rozšiřovat. Pro detailní seznámení s implementací programu je možno nahlédnout do zdrojového kódu, který je dostatečně okomentován.

Závěr

Cílem diplomové práce bylo přiblížit problematiku genetického programování a vytvořit programový nástroj, který by prostřednictvím genetického programování řešil úlohy symbolické regrese.

V první části práce byl popsán jednoduchý genetický algoritmus, tak jak ho navrhl a realizoval John Holland. Také bylo zmíněno s jakými nedostatky je při návrhu algoritmu nutné počítat a jak je možné tyto nedostatky potlačit rozšířením algoritmu o další prvky jako elitsmus nebo škálování.

V druhé části práce bylo přiblíženo genetické programování. Byl popsán rozdílný způsob reprezentace potenciálních řešení, který se liší od jednoduchého genetického algoritmu, a byly popsány všechny genetické operátory, které navrhl John Koza. Zmíněny také byly nedostatky genetického programování a možnosti jak se těmto nedostatkům vyvarovat. V závěru této části bylo genetické programování demonstrováno na příkladu tzv. “umělého mravence”.

Třetí část práce se věnovala samotnému návrhu a implementaci programového nástroje. Nejprve byl stanoven cíl a požadavky, které musí program naplnit. Pak byl vytvořen návrh algoritmu genetického programování řešící symbolickou regresi. Byl navržen jeho objektový model a následně byl program implementován v jazyce C++. Po implementaci byl program otestován na dvou vzorových příkladech. Program při řešení problematiky nalezl hledané optimální funkce. Jak ale bylo demonstrováno, kvalita řešení a průběh hledání je možné jak pozitivně, tak i negativně ovlivnit nastavením parametrů algoritmu a proto by uživatel měl mít vždy představu o tom, jaký význam mají jednotlivé parametry a jak by měly být správně nastaveny. Jejich nastavení se odvíjí jednak od složitosti hledaného řešení a jednak z průběhu samotného algoritmu, kdy během jeho běhu je možné parametry měnit a tím potlačit negativní jevy, které mohou nastat. V závěru části jsou pak navrženy náměty na další rozšíření programu.

Z důvodu, že program bude pravděpodobně v budoucnu dále rozšiřován, jako součást práce je v přílohách kromě uživatelské příručky, také přiložena “programátorská příloha”, která stručně popisuje, jakým způsobem je možné rozšíření realizovat.

Cíl diplomové práce byl naplněn. Problematika byla přiblížena a byl vytvořen program, který dokáže úspěšně řešit problematiku symbolické regrese prostřednictvím genetického programování, což bylo demonstrováno na vzorových příkladech. Jsou tu ale ještě další možnosti jak průběh algoritmu zefektivnit a rozšíření programu o tyto možnosti se může stát námětem dalších prací.

Genetické programování je poměrně novou metodou v oblasti výzkumu umělé inteligence a představuje nové možnosti, které ještě zdaleka nebyly plně využity. Metody využívající genetické programování k řešení složitých problémů jsou v praxi použitelné. Což ostatně dokazuje fakt, že prostřednictvím nich byly i optimalizovány některé ještě v nedávné době patentované vynálezy [2]. Genetické programování by mohlo být běžně použito při řešení složitých rozhodovacích problémů, při krizových situací, k optimalizaci řízení zásob, při návrhu a optimalizaci navrhovaných zařízení, v počítačových hrách a také najde uplatnění při empirickém výzkumu. Bohužel programové nástroje řešící dané problematiky prostřednictvím genetického programování nejsou zatím nikterak rozšířeny. Genetické programování je dnes dále intenzivně rozvíjeno v akademické sféře a očekává se další jeho rozšíření, které posune možnosti výzkumu umělé inteligence o velký krok vpřed.

Použitá literatura

[1] BOBEK, Miroslav. Systémy hromadné obsluhy a jejich číslicová simulace. Liptovský Mikuláš : Vojenská akademie Antonína Zápotockého, 1981. 130 s.

[2] HYNEK, Josef. Genetické algoritmy a genetické programování. [s.l.] : Grada Publishing, a.s., 2008. 179 s. ISBN 978-80-247-2695-3

[3] KOZA, John R. Genetic Programming : On The Programming of Computers by Means of Natural Selection. [s.l.] : MIT Press, 1998. 805 s. ISBN 0-262-11170-5.

[4] KUHLMANN, Hans, HOLLICK, Mike. Genetic Programming in C/C++ [online]. [2008] [cit. 2008-07-05]. Dostupný z WWW: <http://www.cis.upenn.edu/~hollick/genetic/paper2.html>.

[5] MITCHELL, Mellanie. Introduction to Genetic Algorithms. [s.l.] : The MIT Press, 1998. 221 s. [cit. 2008-6-28]Dostupný z WWW: <http://books.google.com>. ISBN 978-0-262-63185-3.

[6] POLI, Ricardo, LANGDON, William B., MCPHEE, Nicholas F. A Field Guide to Genetic Programming . [s.l.] : [s.n.], 2008. 250 s. [cit. 2008-6-28] Dostupný z WWW: <http://www.lulu.com/items/volume_63/2167000/2167025/2/print/book.pdf>. ISBN ISBN 978-1-4092-0.

[7] REEVES, Colin R, ROWE, Jonathan E. Genetic algorithms : Principles and perspectives : a guide to GA theory. [s.l.] : Springer, 2003. 332 s. [cit.2008-09-15]. Dostupný z WWW: <books.google.com/books>. ISBN 1402072406.

[8] SAITO, Mutsua. An Application of Finite Field : Design and Implementation of 128-bit Instruction-Based Fast Pseudorandom Number Generator . [s.l.], 2007. 20 s. Vedoucí diplomové práce Makoto Matsumoto. [cit. 2008-10-20] Dostupný z WWW: <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/M062821.pdf>.

[9] SHAFRANOVICH, Y. Common Format and MIME Type for Comma-Separated Values (CSV) Files. [online], 2005[cit. 2009-2-2]. Dostupný z WWW: <http://tools.ietf.org/html/rfc4180>.

[10] WEISE, Thomas. Global Optimization Algorithms : Theory and Application. [s.l.] : [s.n.], c2008. 686 s. 2. [cit. 2009-1-5]Dostupný z WWW: <http://www.it-weise.de/projects/book.pdf>.

[11] ZELIVKA, Ivan. Symbolic regression [online]. 2000 [cit. 2009-04-10]. Dostupný z WWW: <http://www.mafy.lut.fi/EcmiNL/ecmi35/node70.html>.

[12] Biographical Sketches : John Henry Holland [online]. [2000] [cit. 2009-04-23]. Dostupný z WWW: <http://www.cs.oswego.edu/~blue/hx/courses/cogsci1/s2001/section05/subsection5/main.html>.

[13] Dr. Lawrence J. Fogel [online]. [2007] [cit. 2009-04-20]. Dostupný z WWW: <http://www.cs.oswego.edu/~blue/hx/courses/cogsci1/s2001/section05/subsection5/main.html>.

[14] Encyklopedie fyziky : Elektrický odpor vodiče, Ohmův zákon pro část obvodu [online]. 2006-2009 [cit. 2009-04-20]. Dostupný z WWW: <http://www.cs.oswego.edu/~blue/hx/courses/cogsci1/s2001/section05/subsection5/main.html>.

[15] Home Page of John R. Koza [online]. 2008 [cit. 2009-04-20]. Dostupný z WWW: <http://www.cs.oswego.edu/~blue/hx/courses/cogsci1/s2001/section05/subsection5/main.html>.

[16]H.-P. Schwefel [online]. [2007] [cit. 2009-04-20]. Dostupný z WWW: <http://www.cs.oswego.edu/~blue/hx/courses/cogsci1/s2001/section05/subsection5/main.html>.

[17] Qt : Code less. Create more. Deploy everywhere. [online]. 2009 [cit. 2008-12-20]. Dostupný z WWW: <http://www.qtsoftware.com>.

[18] SIMD-oriented Fast Mersenne Twister (SFMT) : twice faster than Mersenne Twister [online]. 2008, 26.8.2008 [cit.2009-04-06]. Dostupný z WWW: <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html>

[19] Techmania : Kirchhoffovy zákony [online]. c2008 [cit. 2009-04-20]. Dostupný z WWW: <http://www.techmania.cz/edutorium/art_exponaty.php?xkat=fyzika&xser=456c656b74f8696e612061206d61676e657469736d7573h&key=397>.

[20] Technische Universität Berlin : Prof. Dr.-Ing. Ingo Rechenberg [online]. [2000] [cit. 2009-04-20]. Dostupný z WWW: <http://www.cs.oswego.edu/~blue/hx/courses/cogsci1/s2001/section05/subsection5/main.html>.

Seznam příloh

Příloha A Uživatelská příručka.
Příloha B Programátorská dokumentace.

Příloha A - Uživatelská příručka

Základní popis grafického rozhraní

Grafické rozhraní je složeno z 5 oken, mezi nimiž je možno přepínat prostřednictvím záložek v horní části hlavního okna programu. Dokud není načtena datová matice a neproběhla aspoň jedna generace algoritmu, většina položek rozhraní zůstává prázdná. Program neobsahuje žádné menu. Načtení datové matice a uložení textových souborů obsahující výstupy programu je realizováno prostřednictvím tlačítek, která jsou umístěna v dolní části oken a nachází se v příslušných oknech, kam dle významu akce, kterou reprezentují, přísluší.

Význam jednotlivých záložek:

- Záložka **Genetický algoritmus** – obsahuje nastavení genetického algoritmu, graf o průběhu hledání nejlepšího řešení, tlačítko, které spouští samotný algoritmus genetického programování a tlačítko, které smaže stávající generaci.

- Záložka **Funkce jedince** – zobrazuje základní ukazatele popisující nejlepší nalezené řešení v aktuální generaci. Zaznamenává nejlepší nalezený výraz pro každou generaci. Prostřednictvím tlačítka *Uložit jako tabulku* ukládá záznamy do textového souboru, který je vhodný pro načtení do tabulkového procesoru pro další zpracování.

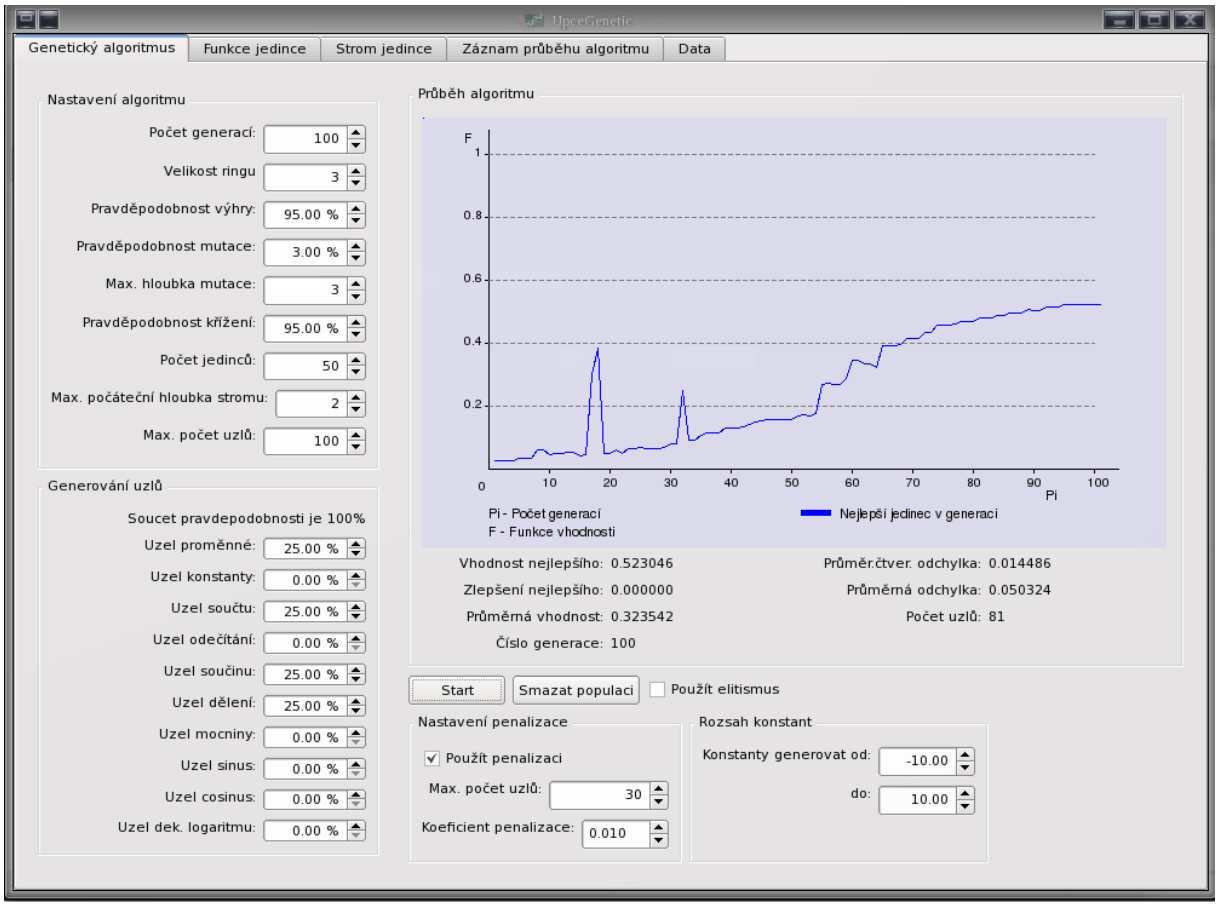
- Záložka **Strom jedince** – zobrazuje syntaktický strom nejlepšího jedince v populaci.

- Záložka **Záznam průběhu algoritmu –** obsahuje tabulku, která zobrazuje a zaznamenává sledované ukazatele pro každou generaci. V dolní části okna se nachází tlačítko, prostřednictvím něhož je možné uložit tabulku včetně nastavení parametrů algoritmu jako textový soubor.

- Záložka **Data** – zobrazuje načtenou datovou matici včetně hodnoty, kterou vrací nejlepší řešení aktuální generace pro každý případ datové matice. Obsahuje také ukazatele sledující načtenou datovou matici a přesnost nalezeného řešení (chyby odhadu). V dolní části okna se nachází dvě tlačítka. Prostřednictvím prvního tlačítka je možné načíst vstupní datovou matici a prostřednictvím druhého tlačítka je možné aktuální datovou matici včetně odhadů pro jednotlivé případy uložit.

Záložka genetický algoritmus

Pod touto záložkou se nachází okno (obrázek č. 1), kde je možné nastavit veškeré parametry algoritmu genetického programování. Pro potřeby popisu si okno rozdělíme na čtyři části – levou horní část, levou dolní část, pravou horní část a pravou dolní část.



Obrázek 1 - Záložka genetický algoritmus.

V pravé horní části se nachází graf, který zobrazuje průběh algoritmu. V grafu je zobrazena vhodnost nejlepšího jedince v dané generaci. Číslo dané generace v grafu je relativní v tom smyslu, že pokud algoritmus přerušíme a opět spustíme s tím, že budeme pokračovat ve stávající populaci, bude se graf vykreslovat znovu a číslo populace v grafu se bude určovat znovu od 0, ačkoliv skutečné číslo generace je jiné (je zobrazeno mezi ukazateli o současné generaci pod grafem). Skutečným číslem generace rozumíme číslo generace od vytvoření populace až po její smazání.

V záložce genetický algoritmus je možné nastavit následující parametry:

• **Počet generací** (levá horní část) – nastavení počtu generací. Jakmile algoritmus dosáhne generace definované touto hodnotou, zastaví se. Pokud algoritmus opětovně spustíme, provede opět tolik generací kolik jich je definováno touto hodnotou.

• **Velikost ringu** (levá horní část) – nastavení velikosti ringu při turnajové selekci. Definuje kolik jedinců, bude selekčním mechanismem vybráno, aby se účastnilo turnajové selekce. Nejlepší jedinec z nich (pokud je pravděpodobnost výhry 100%) se pak bude selekčním mechanismem vybrán a bude moct předat své geny do následující generace. Zvýšením této hodnoty lze zvýšit selekční tlak. Doporučená hodnota 3-4.

• **Pravděpodobnost výhry** (levá horní část) – pravděpodobnost, že při turnajové selekci vyhraje silnější jedinec (jedinec, který reprezentuje lepší řešení). Touto hodnotou je možné korigovat selekční tlak. Doporučená hodnota 85-95%.

• **Pravděpodobnost mutace** (levá horní část) – pravděpodobnost, že na jedince, který byl vybrán selekčním mechanismem, bude uplatněn operátor mutace. Zvýšením této hodnoty lze zvýšit variabilitu v populaci. Doporučená hodnota 3-6%.

• **Max. hloubka mutace** (levá horní část) – v případě uplatnění mutace na jedince, je náhodně vybrán uzel a je smazán včetně jeho případného podstromu. Na jeho místo ve stromu je pak vygenerován nový podstrom. Při generování nový podstrom nepřesáhne hloubku definovanou touto hodnotou. Snížením této hodnoty, je možné zabránit tvorbě složitých jedinců. Doporučená hodnota je 4 -7. Záleží ale na odhadované složitosti optimálního řešení. Čím předpokládané řešení je složitější, tím by tato hodnota měla být vyšší.

• **Pravděpodobnost křížení** (levá horní část) – definuje, s jakou pravděpodobností bude na dvojici jedinců vybraných selekčním mechanismem uplatněn operátor křížení. Doporučená hodnota je 85-95%.

• **Počet jedinců** (levá horní část) – definuje počet jedinců v populaci. V případě, že jsme běh algoritmu zastavili a snížíme počet jedinců v populaci, při následném spuštění algoritmu bude z populace odstraněno tolik nejhorších

jedinců, kolik jich představuje rozdíl mezi současným a požadovaným počtem jedinců v populaci (lze tak vlastně realizovat operátor decimace). V případě, že požadujeme vyšší počet jedinců, než kolik jich současná populace obsahuje, bude populace doplněna nově vygenerovanými jedinci.

• **Maximální počáteční hloubka stromu** (levá horní část) – definuje maximální povolenou hloubku, kterou nesmí nově generovaný strom nového jedince překročit. Doporučuje se začínat s nižší hloubkou, i když předpokládáme složitější řešení. Doporučená hodnota 4-7.

• **Maximální počet uzlů** (levá horní část) – definuje maximální počet uzlů jedince. Pokud jedinec operátorem křížením nebo mutace získá více uzlů, než kolik povoluje tato hodnota, nebude vložen do následující generace, ale místo něho bude do generace vložen jeden z rodičů (v případě mutace původní jedinec).

• **Generování uzlů** (celá levá dolní část) – určuje, s jakou pravděpodobností budou generovány jednotlivé typy uzlů. Součet všech pravděpodobností musí být roven 100%, jinak algoritmus při spuštění nahlásí chybu. Pokud pro daný typ uzlu definujeme pravděpodobnost 0%, nebude daný typ uzlu použit (nebude se nacházet v množině funkcí, terminálů definovaných k řešení daného problému).

• Tlačítko **Start** (pravá dolní část) - zahájí algoritmus genetického programování při daném nastavení. Opětovné kliknutí na toto tlačítko vyvolá zastavení algoritmu. V některých situacích může být odezva programu na zastavení algoritmu být dlouhá. Dlouhá odezva je způsobena tím, že algoritmus se zastaví, až když dokončí výpočet vhodností některých jedinců, což může být výpočetně velice náročné. Pokud algoritmus přerušíme, můžeme změnit nastavení algoritmu a dalším stisknutím tlačítka pokračujeme v algoritmu se stávající populací. Stejnou akci je také možné vyvolat stisknutím klávesy F5.

• Tlačítko **Smazat populaci** (pravá dolní část) – smaže stávající populaci. Při opětovném spuštění algoritmu je vygenerována populace nová. Populaci je vhodné smazat, pokud algoritmus uvízl delší dobu v suboptimálním řešení, nebo když jedinci stávající populace jsou příliš složití, algoritmus nedokáže vygenerovat lepšího jedince a nejlepší nalezené řešení je nízce ohodnoceno.

• Možnost **Použít elitismus** (pravá dolní část) – při zaškrtnutí bude použit eletismus. V každé generaci bude nalezen nejlepší jedinec a ten pak bude vložen do následující generace v nezměněné podobě. Bude tak zajištěno, že neztratíme dosud nejlepší nalezené řešení.

• Možnost **Použít penalizaci** (pravá dolní část) – při zaškrtnutí bude uplatňována na každou generaci penalizace, která bude penalizovat složitější jedince a zhoršovat jim vhodnost, bude tak pro ně obtížnější předat své složitější geny následující generaci. Vhodné použít pokud jedinci jsou příliš složití a požadujeme jednodušší řešení, přitom ale chceme dát možnost složitějším jedincům se prosadit v případě, že obsahují lepší řešení.

• **Max. počet uzlů** (pravá dolní část) – definuje, od jakého počtu uzlů bude zahájena penalizace. Pokud strom jedince tuto hodnotu přesáhne, bude mu úměrně k rozdílu mezi počtem jeho uzlů a touto hodnotou zhoršena vhodnost. Rozdíl bude ještě vynásoben koeficientem penalizace.

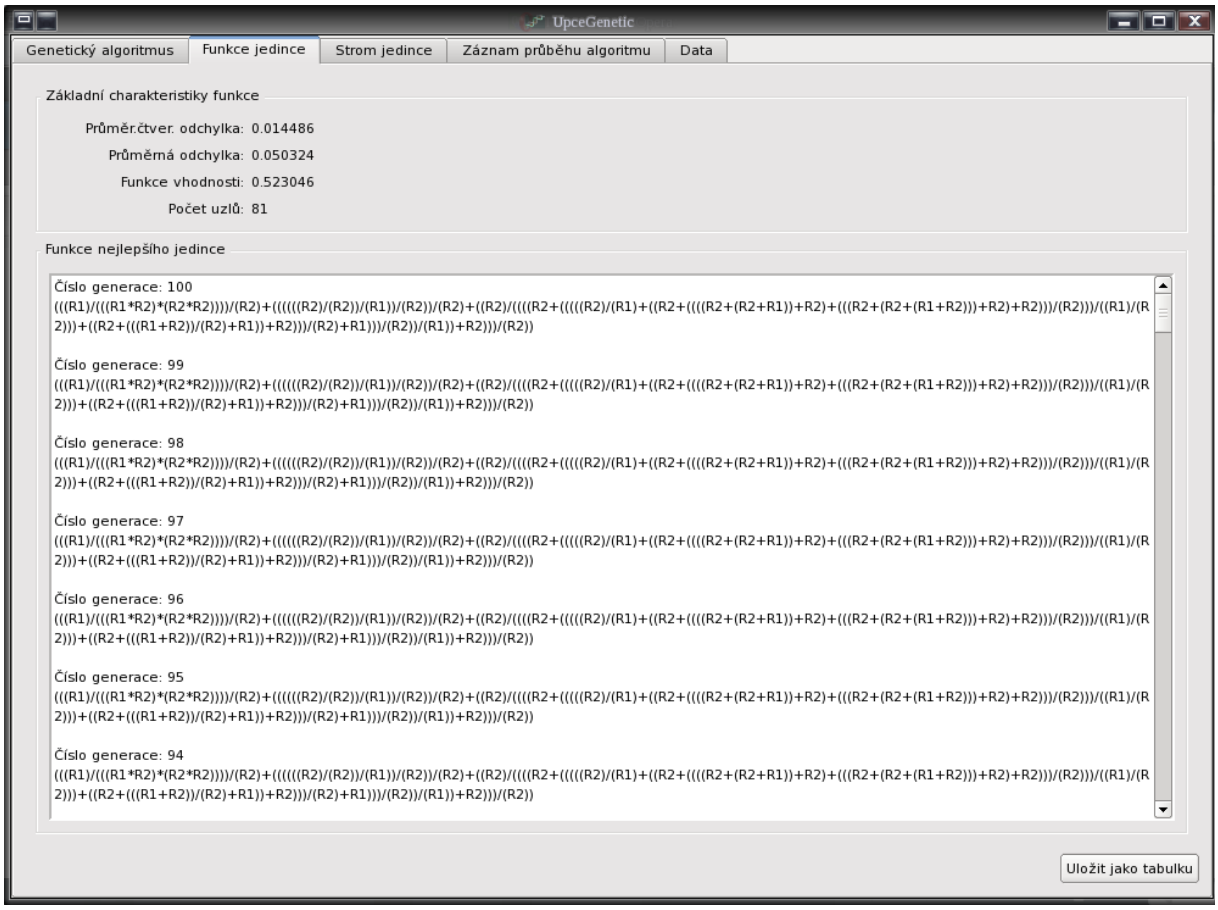
• **Koeficient penalizace** (pravá dolní část) – určuje, jak silně bude vhodnost jedinců, na které bude uplatněna penalizace zhoršena. Pokud si nepřejeme složité jedince, měla by být tato hodnota vyšší. Pokud mírně složité jedince chceme tolerovat, tato hodnota by měla být nižší. Doporučená hodnota se odvíjí od složitosti jedinců a je obtížné ji takto stanovit. Měla by se pohybovat v mezích 0,05-0,001.

• **Rozsah konstant** (pravá dolní část) – při použití terminálu konstanty, jsou hodnoty konstanty generovány v tomto rozsahu.

Záložka funkce jedince

V této záložce (obrázek č. 2) se zobrazuje výraz funkce reprezentující nejlepší nalezené řešení. Daný výraz se zaznamenává pro každou generaci. V případě, že není použit elitismus a nejlepší řešení bylo ztraceno, je možné daný výraz zde dohledat.

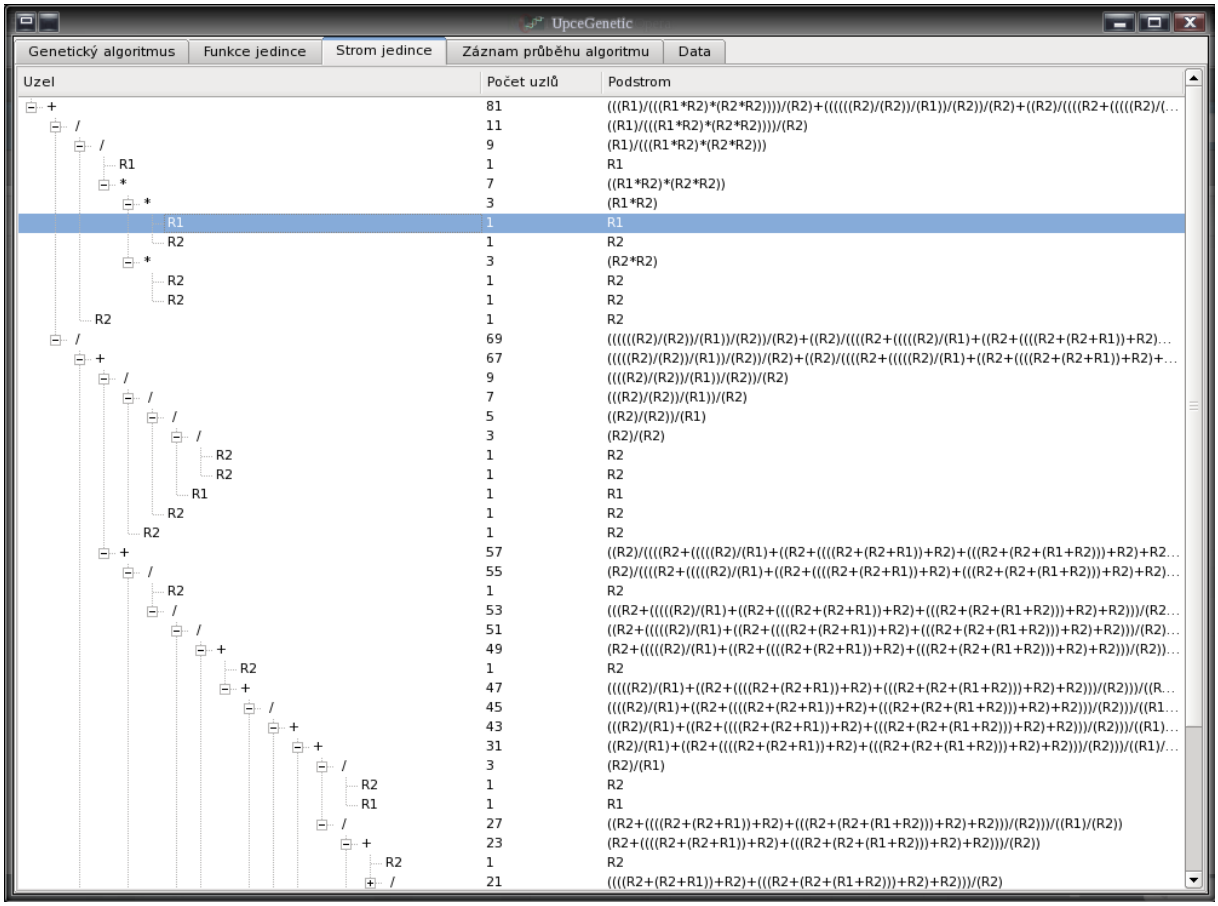
V dolní části se pak nachází tlačítko, po jehož stisknutí se vyvolá dialog k uložení textového souboru, který bude obsahovat tabulku obsahující výrazy nejlepších řešení pro každou generaci. Dialog je také možné vyvolat klávesovou zkratkou “Ctrl+F”.



Obrázek 2 - Záložka funkce jedince.

Záložka strom jedince

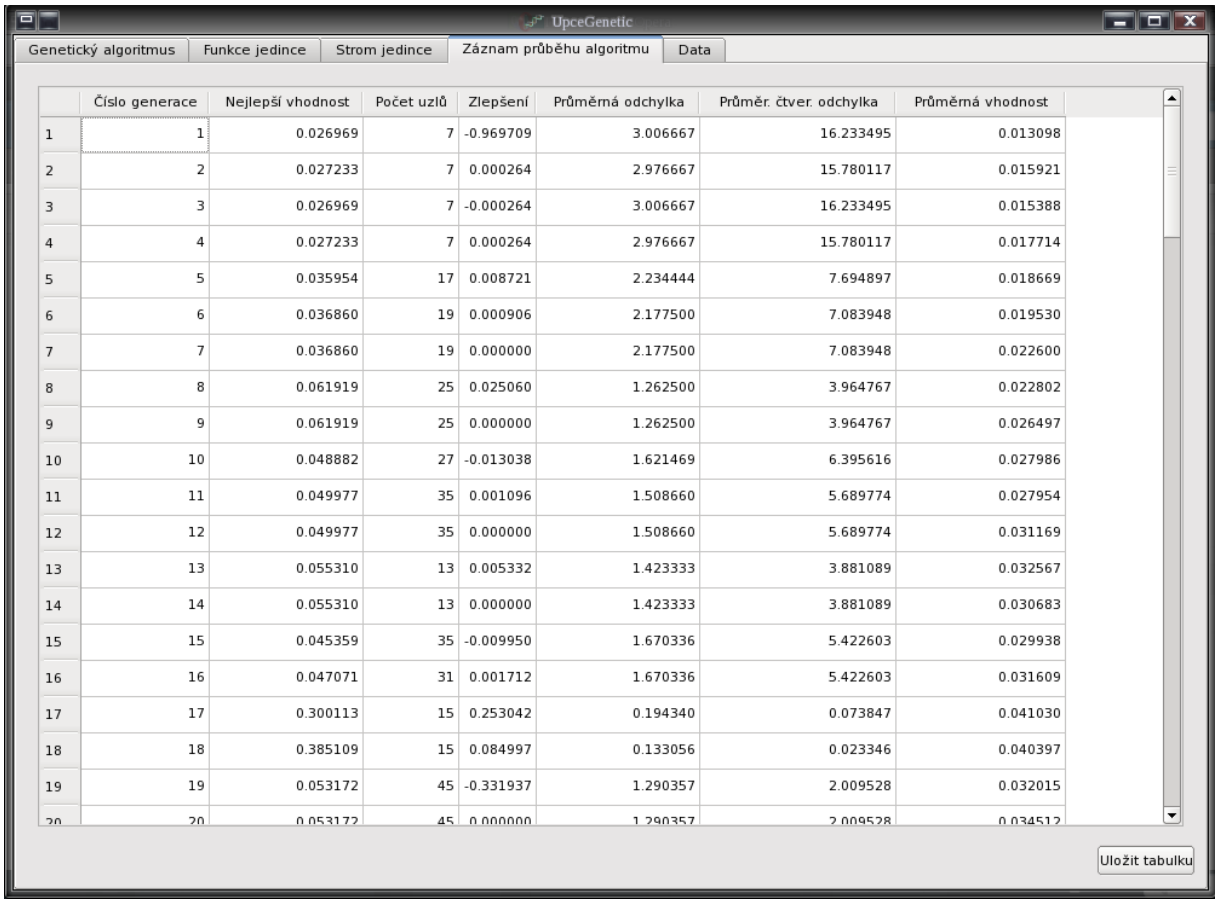
Záložka strom (obrázek č. 3) jedince zobrazuje syntaktický strom nejlepšího jedince v aktuální generaci. Vzhledem k tomu, že nemá smysl v této formě vidět, jak se podoba syntaktického stromu během algoritmu mění (forma zobrazení není dostatečně vhodná ke sledování), je zobrazení syntaktického stromu dostupné pouze tehdy, když je algoritmus genetického programování zastaven.



Obrázek 3 - Záložka strom jedince.

Záložka záznam průběhu algoritmu

V záložce (obrázek č. 4) se zobrazuje tabulka obsahující záznamy z každé generace o vybraných ukazatelích, kteří sledují kvalitu nejlepšího řešení v generaci a průměrnou vhodnost jedinců v dané generaci. Zobrazenou tabulku je možné uložit do textového souboru. Dialog pro uložení tabulky je vyvolán po kliknutí na tlačítko v dolní části okna nebo po stisknutí klávesové zkratky “Ctrl+P”. Do textového souboru kromě tabulky bude uloženo také nastavení algoritmu. V případě, že algoritmus byl v některé generaci zastaven, bude do textového souboru uloženo i nastavení, které bylo před tímto zastavením.

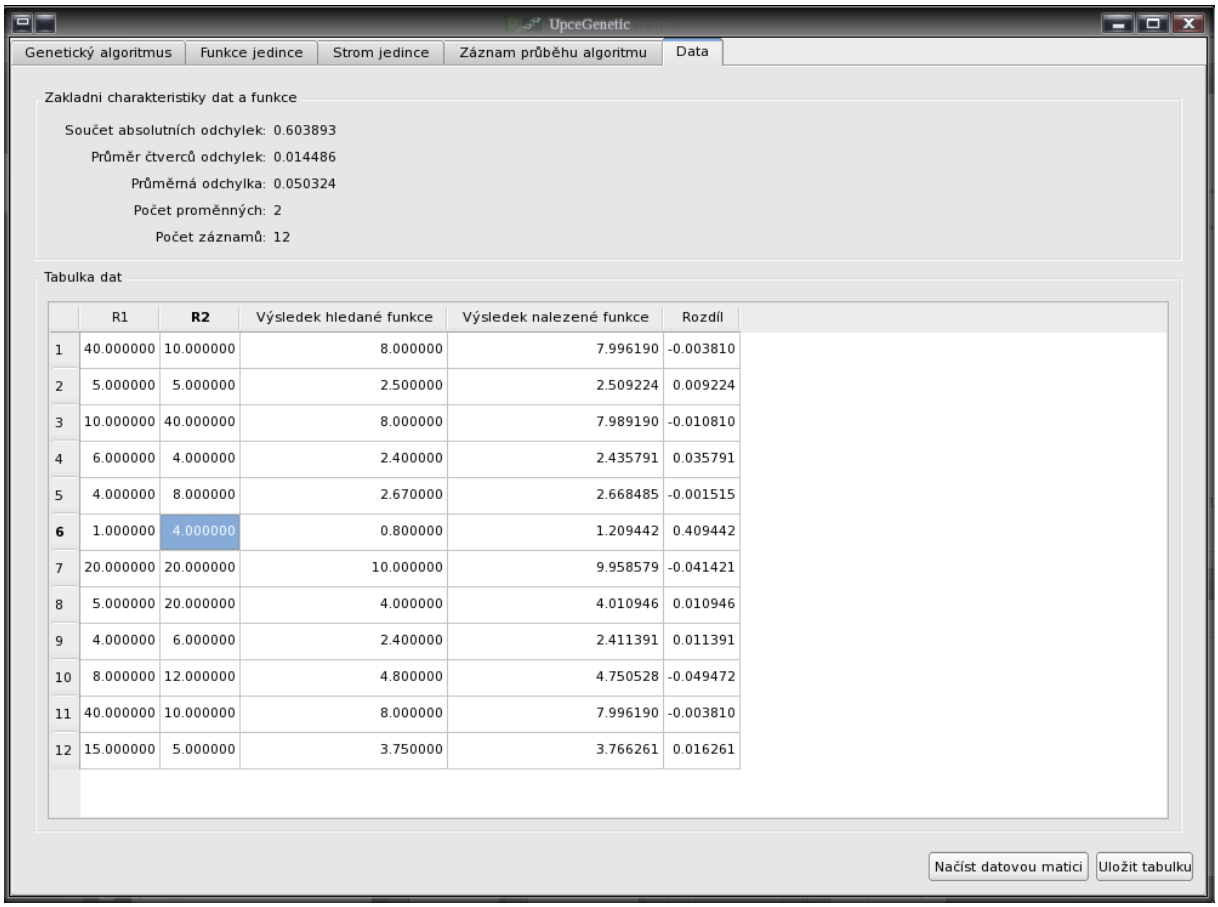


	Číslo generace	Nejlepší vhodnost	Počet uzlů	Zlepšení	Průměrná odchylka	Průměr. čtver. odchylka	Průměrná vhodnost
1	1	0.026969	7	-0.969709	3.006667	16.233495	0.013098
2	2	0.027233	7	0.000264	2.976667	15.780117	0.015921
3	3	0.026969	7	-0.000264	3.006667	16.233495	0.015388
4	4	0.027233	7	0.000264	2.976667	15.780117	0.017714
5	5	0.035954	17	0.008721	2.234444	7.694897	0.018669
6	6	0.036860	19	0.000906	2.177500	7.083948	0.019530
7	7	0.036860	19	0.000000	2.177500	7.083948	0.022600
8	8	0.061919	25	0.025060	1.262500	3.964767	0.022802
9	9	0.061919	25	0.000000	1.262500	3.964767	0.026497
10	10	0.048882	27	-0.013038	1.621469	6.395616	0.027986
11	11	0.049977	35	0.001096	1.508660	5.689774	0.027954
12	12	0.049977	35	0.000000	1.508660	5.689774	0.031169
13	13	0.055310	13	0.005332	1.423333	3.881089	0.032567
14	14	0.055310	13	0.000000	1.423333	3.881089	0.030683
15	15	0.045359	35	-0.009950	1.670336	5.422603	0.029938
16	16	0.047071	31	0.001712	1.670336	5.422603	0.031609
17	17	0.300113	15	0.253042	0.194340	0.073847	0.041030
18	18	0.385109	15	0.084997	0.133056	0.023346	0.040397
19	19	0.053172	45	-0.331937	1.290357	2.009528	0.032015
20	20	0.053172	45	0.000000	1.290357	2.009528	0.034512

Obrázek 4 - Záložka záznam průběhu algoritmu.

Záložka data

Pod záložkou data (obrázek č. 5) se nachází okno, které zobrazuje samotnou vstupní datovou matici. V dolní části okna se nachází dvě tlačítka. Tlačítko *Načíst datovou matici* po kliknutí zobrazí uživatelský dialog, prostřednictvím něhož uživatel může zvolit soubor, který obsahuje vstupní datovou matici. Vstupní datová matice musí být v takovém formátu, že v prvním řádku obsahuje záhlaví sloupců – názvy proměnných. Každý řádek pak reprezentuje jeden případ a v každém sloupci pak je uvedena hodnota dané proměnné pro daný případ. Jeden ze sloupců musí reprezentovat závislou proměnnou, jejíž vztah k ostatním nezávislým proměnným v datové matici bude algoritmus určovat prostřednictvím symbolické regrese. Proměnné musí mít vyplněné hodnoty pro všechny případy. Prázdný znak pro hodnotu je programem nepřijatelný a je považován za chybu ve vstupní datové matici.



	R1	R2	Výsledek hledané funkce	Výsledek nalezené funkce	Rozdíl
1	40.000000	10.000000	8.000000	7.996190	-0.003810
2	5.000000	5.000000	2.500000	2.509224	0.009224
3	10.000000	40.000000	8.000000	7.989190	-0.010810
4	6.000000	4.000000	2.400000	2.435791	0.035791
5	4.000000	8.000000	2.670000	2.668485	-0.001515
6	1.000000	4.000000	0.800000	1.209442	0.409442
7	20.000000	20.000000	10.000000	9.958579	-0.041421
8	5.000000	20.000000	4.000000	4.010946	0.010946
9	4.000000	6.000000	2.400000	2.411391	0.011391
10	8.000000	12.000000	4.800000	4.750528	-0.049472
11	40.000000	10.000000	8.000000	7.996190	-0.003810
12	15.000000	5.000000	3.750000	3.766261	0.016261

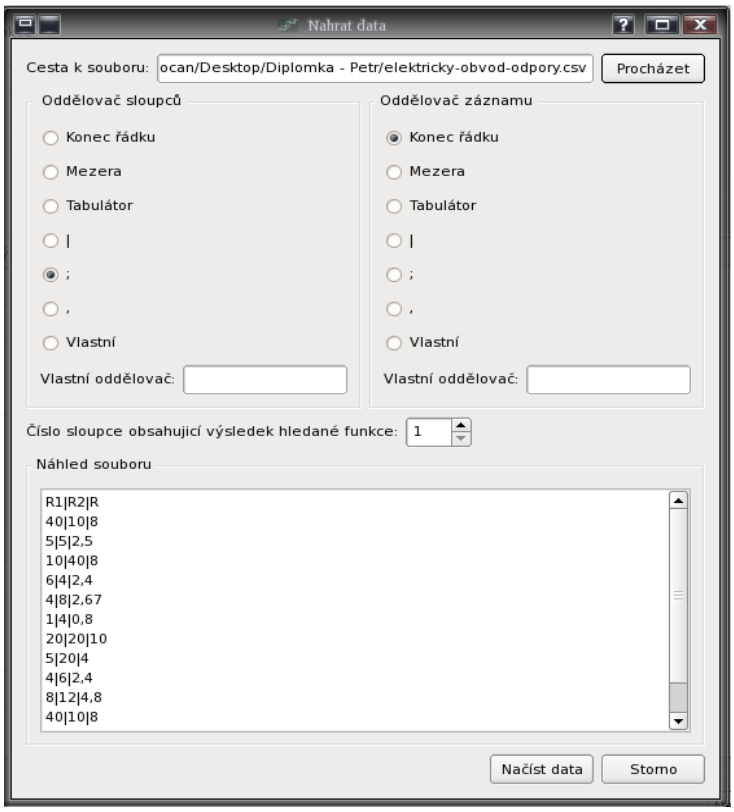
Obrázek 5 - Záložka data.

Před načtením datové matice se uživateli zobrazí dialog, ve kterém kromě cesty k souboru musí zvolit vhodné oddělovače sloupců a případů (řádků). V dolní části dialogu se

mu zobrazuje náhled na soubor, aby tak měl představu o obsahu textového souboru. Kromě oddělovačů musí uživatel také definovat, v kterém sloupci se nachází závislá proměnná, jejíž vztah k ostatním hledáme.

Po dialogu následuje načtení matice, kdy program nejprve ověří, zda jsou oddělovače správně zvolené a jestli dle nich obsahuje každý řádek shodný počet sloupců. Program přijímá jak hodnoty s desetinnou tečkou, tak i s desetinnou čárkou. Po úspěšném načtení se matice objeví v tabulce okna.

Po spuštění algoritmu je tabulka v okně pak rozšířena o dva sloupce, v kterých je zobrazována hodnota odhadu nejlepšího řešení v aktuální generaci včetně rozdílu mezi odhadem a skutečnou hodnotou závislé proměnné pro daný případ. Tabulku je možné uložit do textového souboru po kliknutí na druhé tlačítko *Uložit tabulku*, které je umístěno v dolní části okna.



Obrázek 6: Dialog pro načtení datové matice.

Stručný návod k použití programu

Před samotným použitím programu připravíme textový soubor, který bude obsahovat vstupní datovou matici, kde ve sloupcích budou jednotlivé proměnné a v řádcích hodnoty proměnných pro daný případ, který je reprezentován řádkem. Řádky musí být mezi sebou odděleny určitým textovým oddělovačem (nejčastěji znak pro nový řádek) a stejně tak sloupce musí být mezi sebou vzájemně odděleny (nejčastěji středník). Poslední případ musí být také ukončen řádkovým oddělovačem!

Pokud máme textový soubor, připraven spustíme program UpceGenetic. V záložce *Data* kliknutím na tlačítko *Načíst datovou matici*, které je umístěné v dolní části okna, vyvoláme dialog, který nás požádá o specifikaci cesty k textovému souboru, který jsme si předtím připravili a který obsahuje vstupní datovou matici. Zvolíme správné oddělovače sloupců a řádku, zvolíme hodnotu čísla sloupce, který obsahuje závislou proměnnou, jejíž vztah k ostatním proměnným budeme hledat a dáme načíst datovou matici. Pokud oddělovače budou správné a hodnoty matice bude v pořádku, budou načtena data. Zda data byla načtena korektně, si můžeme ihned ověřit v tabulce v záložce *Data*, v které se právě nacházíme.

Po načtení datové matice se přesuneme na záložku *Genetický algoritmus*, kde nastavíme parametry generického parametru a samotný algoritmus spustíme tlačítkem *Start*, které je umístěno v pravé dolní části okna. Na grafu pak vidíme průběh algoritmu. Přepínáním po jednotlivých záložkách můžeme sledovat, jak se aktuální nalezené řešení v jednotlivých generacích přibližuje řešení optimálnímu.

Algoritmus můžeme kdykoliv přerušit stisknutím tlačítka *Start*, v záložce *Genetický algoritmus* a můžeme změnit nastavení parametrů algoritmu. Vzhledem k tomu, že algoritmus před svým zastavením musí dokončit jeden cyklus, může být odezva programu při výpočetně náročných řešení dlouhá. Tlačítko *Start* je vhodné stisknout jenom jednou, protože opětovné jeho stisknutí opět vyvolá spuštění algoritmu. Pokud dojdeme k závěru, že současná generace představuje natolik nevhodná řešení, že z nich nemůže být vygenerováno řešení optimální, můžeme celou populaci smazat kliknutím na tlačítko *Smazat populaci.*

Pokud nalezneme řešení optimální, je vhodné si do textových souborů uložit průběh algoritmu, datovou matici a nastavení algoritmu. Tlačítka pro uložení souborů se nachází v příslušných záložkách v dolní části.

Příloha B – Programátorská dokumentace

Rozvržení zdrojového kódu do souborů

Zdrojový kód programu je rozdělen do hlavičkových a zdrojových souborů. Třídy a jejich metody jsou deklarovány ve dvou hlavičkových souborech *gui.h* a *upce-genetic.h.* Soubor *gui.h* obsahuje deklaraci tříd, které zajišťují grafické rozhraní. Soubor *upce-genetic.h* obsahuje deklaraci tříd, které tvoří samotné jádro programu. Jednotlivé zdrojové soubory, pak jsou většinou pojmenovány dle tříd, jejichž metody definují. Některé zdrojové soubory však obsahují definice metod více tříd. Pro zkompilování jsou také vyžadovány některé hlavičkové soubory knihovny Qt.

Rozdělení zdrojového kódu v souborech je následující:

- *dataLoadDialog.cpp* – definice metod třídy DataLoadDialog,
- *dataTable.cpp* – definice metod třídy DataTable,
- *dSFMT.c* – implementace generátoru pseudonáhodných čísel,
- *dSFMT.h, dSFMT-params11213.h, dSFMT-params1279.h, dSFMT-params19937.h, dSFMT-params2203.h, dSFMT-params4253.h, dSFMT-params521.h, dSFMT-params.h* – hlavičkové soubory generátoru pseudonáhodných čísel,
- *dSFMT-LICENSE.txt* – soubor obsahující licenční omezení generátoru,
- *exceptions.cpp* – definice metod vyjímek,
- *functions.cpp* – globální funkce,
- *functionTree.cpp* – definice metod třídy FunctionTree,
- *functionViewer.cpp* – definice metod třídy FunctionViewer,
- *graph.cpp* – definice metod třídy Graph,
- *gui.cpp* – definice metod třídy MainWindow,
- *gui.h* – deklarace tříd grafického rozhraní,
- *individual.cpp* – definice metod třídy Individual,
- *logTable.cpp* – definice metod třídy LogTable,
- *main.cpp* – soubor obsahující funkci main(),
- *nodes.cpp* – definice metod tříd odvozených od třídy Node,
- *outputFileMaker.cpp* – definice metod třídy OutputFileMaker,
- *population.cpp* – definice metod třídy Population,
- *upce-genetic.h* - deklarace tříd tvořících jádro programu.

Možné způsoby rozšíření programu

Stručně bude naznačen možný postup při rozšiřování programu, pro hlubší pochopení je zapotřebí prostudovat vybrané zdrojové soubory.

V případě, že je třeba rozšířit algoritmus genetického programování o vybrané genetické operátory, je vhodné nové genetické operátory definovat jako nové metody třídy *Individual*. Inspirací mohou být již definované metody *Individual::mutation() a Individual::crossOver(Indiv2 *Individual)*. Nově definované metody pak musí být zavolány v metodě *Population::tournamentReproduction()*. Tato metoda implementuje samotný genetický algoritmus, který je založen na turnajové selekci.

Pokud je zapotřebí definovat nové terminály nebo funkce, je třeba je definovat jako novou třídu odvozenou od abstraktních tříd *UnaryNode*, *BinaryNode* nebo *Node*. Takto odvozené třídě pak musí být dodefinovány virtuální metody třídy Node. Nutná je také úprava metod třídy *NodeGenerator*, která generuje uzly stromu a musí být upraveno grafické rozhraní, aby bylo možné definovat pravděpodobnost generování uzlu.

Pokud je třeba modifikovat způsob realizace ukládání do souborů (forma uložení, přidání dalších sledovaných hodnot do ukládané datové matice), je třeba modifikovat metody třídy *OutputFileMaker*.

Pokud je třeba rozšířit grafické rozhraní o možnosti nastavení dalších parametrů, je vhodné v případě jednoho parametru, upravit okno záložky genetický algoritmus (v implementačním kódu označena jako AlgoPage) v konstruktoru třídy *MainWindow* (soubor *gui.cpp*). V případě, že máme parametrů více nebo chceme vložit do grafického rozhraní další jiné větší rozšíření, vytvoříme novou záložku jako třídu odvozenou od třídy *QWidget* a tu pak vložíme do grafického rozhraní opět v konstruktoru třídy *MainWindow*.