Analytické programování v C#

Analytic programming in C#

Bc Eva Kaspříková

Diplomová práce
2008

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2007/2008

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Eva KASPŘÍKOVÁ**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Analytické programování v C Sharp**

Zásady pro vypracování:

1. **Zpracujte rešerši algoritmů symbolické regrese.**
2. **Naprogramujte algoritmus Analytického programování v jazyce C Sharp.**
3. **Otestujte výsledný software na vybraných příkladech.**
4. **Vypracujte dokumentaci k programu.**

Rozsah práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **ZELINKA, Ivan. Umělá inteligence v problémech globální optimalizace. Praha : BEN, 2002. 189s. ISBN 80-7300-069-5.**
2. **OPLATKOVÁ, Zuzana. Analytic programming. Zlín : UTB-FT, 2003. 73s. Diplomová práce.**
3. **KOZA, J.R. (1990), Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems, Stanford University Computer Science Department technical report STAN-CS-90-1314.**
4. **NAGEL, Ch., et al. Professional C Sharp 2005 with .NET 3.0 [s.l.] : Wrox Press, 2007. 1798 s. ISBN 9780470124727.**
5. **KAČMÁŘ, Dalibor. Programujeme .NET aplikace : ve Visual Studiu .NET. Praha : Computer Press, 2001. 330s. ISBN 80-7226-569-5.**

Vedoucí diplomové práce: **Ing. Pavel Vařacha**
Ústav aplikované informatiky

Datum zadání diplomové práce: **20. února 2008**

Termín odevzdání diplomové práce: **19. května 2008**

Ve Zlíně dne 20. února 2008

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Analytické programování je metoda, která generuje z elementárních funkcí mnohdy velmi složité funkcionály, které mohou být využity při symbolické regresi. Symbolická regrese je prokládání dat vhodnou matematickou formulí. V teoretické části této práce jsou popsány tři různé algoritmy pro symbolickou regresi. Jedná se o Genetické programování, Gramatickou evoluci a Analytické programování. Protože Analytické programování potřebuje ke svému chodu nějaký evoluční algoritmus je zde popsána Diferenciální evoluce a algoritmus SOMA. Součástí práce je také popis implementace analytického programování v jazyce C# a vyhodnocení výsledků implementace.

Klíčová slova: Analytické programování, Genetické programování, Gramatické evoluce, SOMA, Diferenciální evoluce.

ABSTRACT

Analytic programming is a method, which generates sometimes very complex functionals from basic functions. Such functionals can be used for symbolic regression. Symbolic regression is based on intercutting of the data by the appropriate mathematical formula. In the theoretical part of this work, three different algorithms for symbolic regression are described. Among them are Genetic programming, Grammatical evolution and Analytic programming. Because the analytic programming, in order to run, needs some evolution algorithm, Differential evolution and SOMA algorithm is described here. Last part of this work, is also a description of the implementation of analytic programming in C# language and evaluation of this implementation.

Keywords: Analytic programming, Genetic programming, Grammatical evolution, SOMA, Differential evolution.

Prohlašuji, že jsem na diplomové práci pracovala samostatně a použitou literaturu jsem citovala. V případě publikace výsledků, je-li to uvolněno na základě licenční smlouvy, budu uvedena jako spoluautor.

Ve Zlíně …….

Podpis diplomanta

OBSAH

ÚVOD

Úkolem této diplomové práce bylo naprogramovat algoritmus Analytického programování v programovacím jazyce C#. Analytické programování je jedním z algoritmů pro symbolickou regresi. Symbolická regrese spočívá v nalezení matematické formule, která co nejlépe prokládá zadané body. Dalšími algoritmy, které řeší symbolickou regresi jsou Genetické programování a Gramatická evoluce. Všechny tyto algoritmy jsou popsány v teoretické části této práce.

Protože Analytické programováni potřebuje ke svému běhu nějaký evoluční algoritmus, jsou zde popsány evoluční algoritmy SOMA a Diferenciální evoluce. Oba tyto algoritmy jsem použila v programu. Evoluční algoritmy se používají pro optimalizaci různých problémů. Optimalizace spočívá v hledání globálního extrému (především minima). Evoluční algoritmy mohou pracovat s různými typy argumentů (reálná čísla, celá čísla nebo s diskrétní množinou čísel).

Analytické programování pracuje s množinou funkcí a terminálů, které se říká základní množina – GFS. Aby uživatel mohl použít libovolnou funkci, hlavní aplikace neobsahuje žádné předem vytvořené funkce. Při řešení je využito zásuvných modulů – pluginů. Popis řešení je v praktické části této práce. Aby byla ověřena funkčnost programu, je zapotřebí jej otestovat na vybraných funkcích. Výsledek testování je shrnut v závěru práce.

I. TEORETICKÁ ČÁST

1 SYMBOLICKÁ REGRESE

Termínem symbolická regrese se označuje proces prokládání předložených dat vhodnou matematickou formulí. Dlouhou dobu byla tato činnost výhradní doménou lidského mozku, ale v několika posledních desetiletích ji mohou vykonávat i počítače. Prvním takovým algoritmem je Genetické programování (GP) [3]. Dalšími dvěma vhodnými algoritmy jsou Gramatická evoluce (GE) [5] a Analytické programování (AP) [6].

Myšlenka, jak vyřešit různé problémy právě pomocí symbolické regrese byla poprvé předložena J. Kozou, který využívá genetické algoritmy pro Genetické programování. Genetické programování je vlastně symbolická regrese, která používá evoluční algoritmy místo lidského mozku. Schopnost řešit velmi složité problémy byla mnohokrát vyzkoušena, a proto si GP dnes vede tak dobře, že se může aplikovat např. k výrobě vysoce propracovaných elektronických obvodů.

V posledním desetiletí 20. století, C. Ryan vyvinul novou metodu symbolické regrese, která se nazývá Gramatická evoluce. Gramatická evoluce může být považována za metodu rozvíjející GP, protože je postavená na principech, které jsou společné pro oba algoritmy. Stejně jako u GP je cílem GE automatický návrh programů. Důležitou vlastností GE je návrh programů v libovolném jazyce, nejenom v jazyce LISP, jako je tomu u GP.

Nejnovější metodou je Analytické programování. AP je nástrojem pro symbolickou regresi, které je založeno na odlišných principech než GP a GE. Je to univerzální metoda, která může být používána s libovolným evolučním algoritmem.

1.1 Genetické programování

Genetické programování bylo představeno na konci 80 let minulého století J. Kozou. Je to nejstarší metoda automatického vytváření programu pomocí genetických algoritmů. Tato metoda je založena na programovacím jazyce LISP, který je schopný pracovat se symbolickými výrazy. Během existence GP bylo pomocí něj provedeno mnoho příkladů prokládání dat, syntézy logických výrazů, optimalizace trajektorie robota apod.

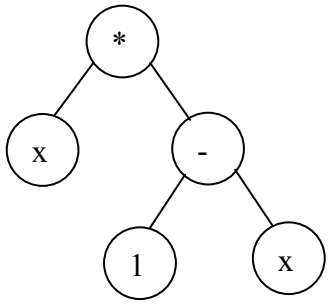
Hlavní princip GP je založený na genetickém algoritmu, který pracuje s populací jedinců a je implementovaný v programovacím jazyce LISP. Jedinci, kteří se v GP používají, nejsou binární řetězce, jak je obvyklé pro genetické algoritmy, ale skládají se z objektů jako sin,

cos, +, MojeFunkce, apod. Tyto funkce mohou být originální LISP funkce nebo funkce definované uživatelem. [3]

Nová populace je vytvářena analytickou cestou. Výsledkem nejsou hodnoty parametrů, ale samotná funkce. Parametry v řetězci chromozomu jsou reprezentovány funkcemi. V nejjednodušším případě jsou zde proměnné, konstanty, základní aritmetické funkce a elementární funkce.

1.1.1 Zápis výrazu

Z dané skupiny funkcí lze například vytvořit x*(1-x). Výraz je reprezentovaný syntaktickým stromem jako na obrázku (Obr. 1).



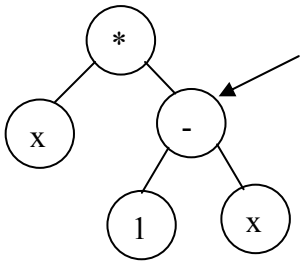
Obr. 1 Syntaktický strom

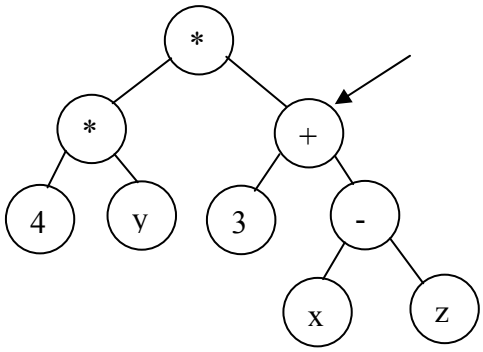
Kořenem stromu je operace na nejvyšší úrovni. Z kořene vede tolik větví, kolik má daná operace argumentů. Výraz se rozpadne na několik částí, na které opět aplikujeme předchozí postup tak dlouho, dokud nenarazíme na nerozložitelné výrazy, jako jsou čísla nebo proměnné. Interpretace syntaktického stromu je jednoduchá. Výraz se vyhodnocuje od listů směrem nahoru ke kořenu stromu.

1.1.2 Křížení v Genetickém Programování

S reprezentací výrazu pomocí syntaktického stromu lze provádět operace křížení a mutace. Obě operace se provádějí jako modifikace struktury stromu nebo jako modifikace jednotlivých uzlů stromu. Postup křížení je zobrazen na obrázku (Obr. 2). U každého z rodičů vybereme křížící bod – uzel a podstromy těchto uzlů vyměníme. Tím vzniknou opět platné programy, které se však liší od původních rodičů.

Uzel křížení je znázorněn šipkou

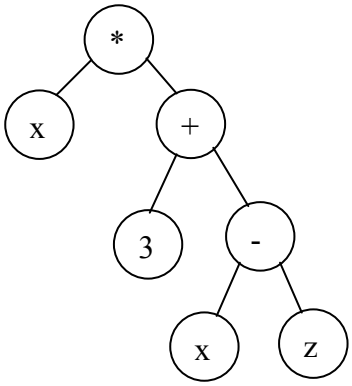


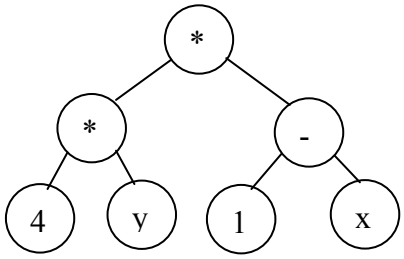


Rodič 1: x*(1-x)

Rodič 2: 4*y*(3+x-z)

Po křížení





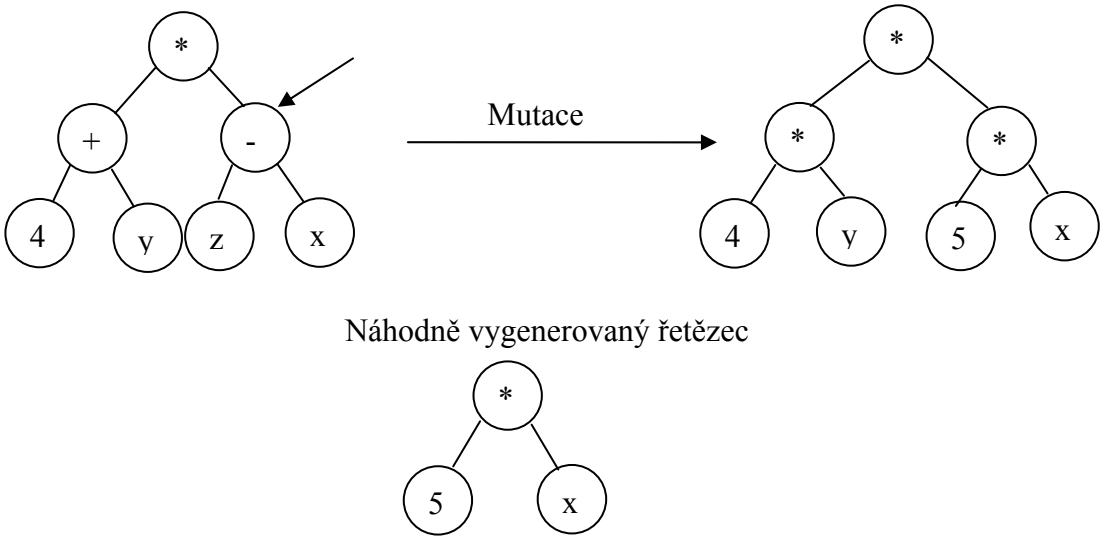
Potomek 1: x*(3+x-z)

Potomek 2: 4*y*(1-x)

Obr. 2 Křížení v genetickém programování

1.1.3 Mutace v Genetickém Programování.

Mutace se provádí podobně jako výše uvedené křížení. Vybereme nějaký uzel stromu a jeho podstrom nahradíme náhodně vygenerovaným jiným podstromem, jak je vidět z obrázku (Obr. 3). Další možností mutace je změnit informaci obsaženou v uzlu. Například v případě čísla můžeme provést jeho náhodnou změnu.



Obr. 3 Mutace v GP

1.2 Gramatická evoluce

Počátky Gramatické evoluce (GE) sahají do roku 1998. Jde o kombinaci genetických algoritmů a genetického programování. Pomocí GE je možné navrhovat programy v libovolném jazyce, který může být popsán bezkontextovou gramatikou, resp. Backus-Naurovou formou (BNF). Na rozdíl od genetického programování, jsou jedinci reprezentováni produkčními pravidly dané gramatiky. Pravidla jsou zakódována do binárního řetězce a používá se jednoduché jednobodové křížení. Výhodou je, že generované programy nejsou omezeny konkrétním programovacím jazykem a takto reprezentovaní jedinci bývají na rozdíl od stromové reprezentace výrazně menší [5].

1.2.1 Backus-Naurova forma

BNF je notace používaná k vyjádření bezkontextové gramatiky, která se používá pro popis formálních jazyků. Bezkontextová gramatika je definována jako čtveřice {N,T,P,S}. N je konečná množina neterminálních symbolů (neterminálů). Neterminál je symbol, který je možné přepsat na řetězec terminálů nebo neterminálů. T je konečná množina terminálních symbolů (terminálů). Terminál je symbol, který je součástí cílového jazyka, např. if,+. P je konečná množina přepisovacích pravidel a S označuje počáteční neterminál.

V BNF mají přepisovací pravidla tvar

$$N::=\alpha,$$

Kde N je neterminál a α je řetězec terminálních a neterminálních symbolů. Pravidlo se používá tak, že ve vygenerované frázi můžeme výskyt neterminálu N nahradit řetězcem α. Různá přepisovací pravidla aplikujeme postupně na vznikající řetězec tak dlouho, dokud nedostaneme řetězec obsahující pouze terminální symboly.

1.2.2 Zakódování chromozomu

Genotyp v gramatické evoluci je stejně jako u genetických algoritmů binární řetězec a říká se mu chromozom. V gramatické evoluci má chromozom takovou funkci, že reprezentuje posloupnost pravidel tak, jak budou postupně aplikována během generování programu.

Chromozomy mohou mít proměnnou délku. Celkový řetězec je rozdělen na osmibitové podřetězce kódující čísla 0-255, které se nazývají *kodony*. Kodony jsou postupně čteny od začátku chromozomu a na základě jejich hodnoty je použito odpovídající pravidlo pro nahrazení nejlevějšího neterminálu. Protože pravidel je většinou méně než 256, je číslo pravidla, které se pro rozvinutí neterminálu použije stanoveno takto:

$$pravidlo = kodon \ \text{mod} \ počet\_pravidel\_pro\_daný\_neterminál.$$

Čtení chromozomu pokračuje zleva doprava, dokud nedojde k jedné z následujících situací:

a. všechny neterminály byly přepsány na terminály
b. je třeba rozvinout ještě jeden nebo více neterminálů, ale už byl přečten poslední kodon v chromozomu.

V prvním případě, pokud ještě zbývají v chromozomu nějaké nepoužité kodony, zbytek kodonu ignorujeme. Ve druhém případě je nutné dokončit odvození programu, aby bylo možné ohodnotit daného jedince. Chybějící kodony budeme číst znovu od začátku chromozomu. Aby nedošlo k zacyklení, používá se omezení na počet průchodů chromozomem.

Následující příklad ukazuje proces sestavení programu. Budeme uvažovat výrazy, ve kterých se mohou vyskytovat operace {+,-,*,/} a proměnné X,Y. Množina terminálů je tedy T={+,-,*,/,X,Y}. Množina neterminálů N={expr, op, var}. Startovací symbol S je <expr>. Množina pravidel P je následující:

<expr>	::=	<op> <expr> <expr>	(0)
	|	<var>	(1)
<op>	::=	+	(0^i)
	|	-	(1^i)
	|	*	(2^i)
	|	/	(3^i)
<var>	::=	X	(0^{ii})
	|	Y	(1^{ii})

Budeme mít následující chromozom

00101000	10100010	01000011	00000110	01111101	11100111	10010010	10001011
40	162	67	12	125	231	146	139

Generování programu začíná rozvinutím startovacího symbolu – <expr>. Pro něj máme dvě pravidla. První kodon chromozomu má hodnotu 40. Protože 40 mod 2=0, bude se <expr> přepisovat pomocí pravidla (0) na <op> <expr> <expr>. Následovat bude rozvinutí symbolu <op>. Ze čtyř možností se vybere pomocí kodonu 162 pravidlo (2^i) a <op> se přepíše na *. Dále budeme pokračovat rozvinutím levého ze dvou zbývajících symbolů. Pomocí kodonu 67 vybereme pravidlo (1) a <expr> se přepíše na <var>. Kodon 12 určuje, že symbol <var> bude přepsán podle pravidla (0^{ii}) na X. Zbývající symbol <expr> se

přepíše podle pravidla (1) na <var>, což určuje kodon 125. Nakonec bude symbol <var> přepsán podle pravidla (1[ii]) na Y. Generování výrazu znázorňuje obrázek (Obr. 4).

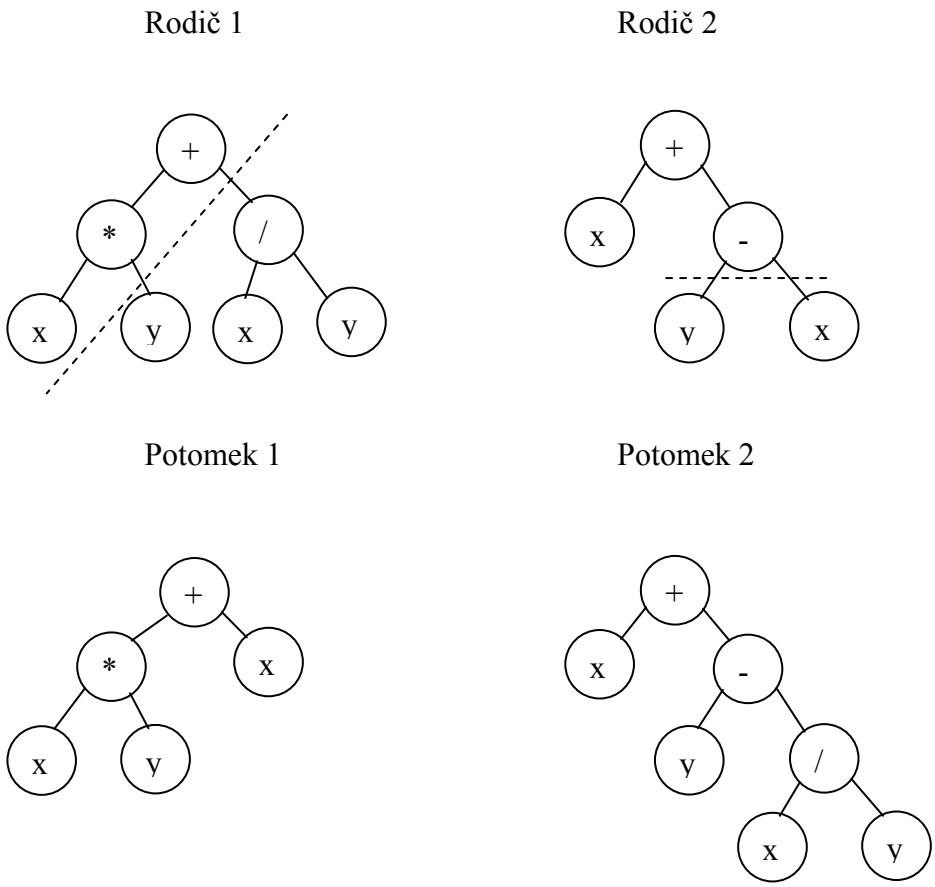
Obr. 4 Generování výrazu v GE

Toto zakódování umožňuje reprezentaci jednoho pravidla několika různými řetězci. To umožňuje tzv. tiché mutace, kdy drobné změny v chromozomu nezpůsobí změnu v programu.

1.2.3 Křížení v GE

V GE se standardně používá jednobodové křížení, které je aplikováno na chromozomy. V místě křížení se chromozomy rozdělí na dvě části. První část reprezentuje kostru nového jedince a druhá reprezentuje větve, které byly z rodičovského stromu odříznuty. Křížení se dokončí tak, že kostra nového jedince bude doplněna uzly a podstromy, které budou generovány použitím kodonů ze zbývající části druhého rodiče. Tyto geny si buď zachovají stejnou interpretaci, jakou měly v původně, nebo ji změní v závislosti na kontextu, v jakém budou použity. Záleží na tom mezi kolika pravidly se má rozhodnout.

Jak je z obrázku (Obr. 5) zřejmé, řezem bude většinou oddělena a nahrazena celá skupina větví. Pokud jde o vzájemné obměny genetického materiálu, je efekt křížení u GE vyšší než u jednoduchého prohazování podstromů, které je obvyklé u GP.



Obr. 5 Křížení v GE

GE přidává dva nové operátory – zdvojení (duplicate) a ořezání (prune). Zdvojení provádí prostou kopii genu. Nově vytvořený gen nemusí být přímo vedle svého vzoru, ale může být umístěn kdekoli v chromozomu. Počet genů, které budou duplikovány je zvolen náhodně. Duplikace umožňuje vytvářet nové, složitější funkce a je také vhodná v případech, kdy je potřeba z kratších chromozomů vytvořit složitější řetězce. Jedinci v GE nepotřebují nutně použít všechny jejich geny. Pro snížení počtu přebytečných genů se používá operátor ořezání, který je aplikován s určitou pravděpodobností na všechny jedince obsahující geny, které se neuplatní při přepisu chromozomu.

Gramatická evoluce byla testována na mnoha známých úlohách, jako jsou například symbolická regrese, symbolická integrace apod. Výsledky naznačují, že jde o velice úspěšnou metodu [5].

1.3 Analytické programování

AP je také nástrojem pro symbolickou regresi a je založené na odlišných principech než GP a GE. AP může využívat jakýkoli evoluční algoritmus, např. Diferenciální evoluci nebo algoritmus SOMA. AP ukazuje využití evolučních algoritmů pro analytické řešení syntézy (symbolickou regresi), a to byl hlavní důvod pro název „analytické programování“ [7].

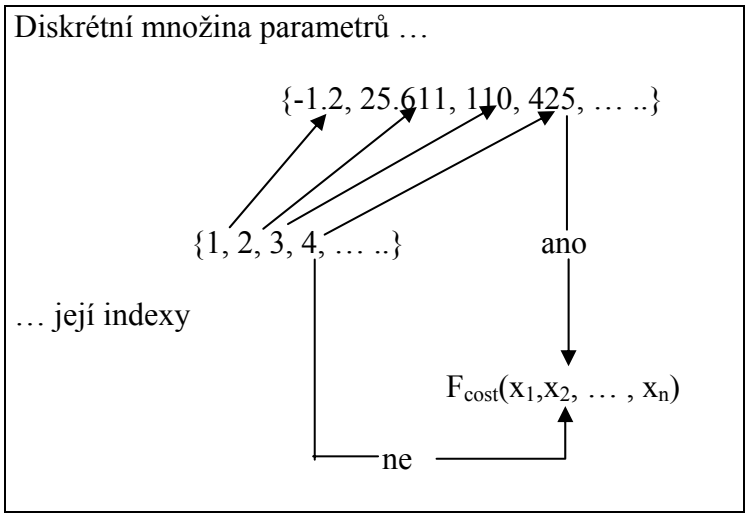
1.3.1 Základní myšlenka AP

Analytické programování bylo inspirováno numerickými metodami v Hilbertově funkcionálním prostoru a genetickým programováním. Principy AP jsou někde mezi těmito dvěma filosofiemi: Z GP přebírá myšlenku evolučního vytvoření symbolických řešení, zatímco hlavní myšlenka funkcionálního prostoru a budování výsledné funkce pomocí procesu prohledávání je převzata z Hilbertových prostorů. AP je postaveno na množině funkcí, operátorů a takzvaných terminálů, což jsou většinou konstanty nebo nezávislé proměnné, například:

i. funkce: sin, cos, and, or,…
ii. operátory: +,-,*,/,…
iii. terminály: 3.14, 110, t,…

Všechny tyto matematické objekty vytvářejí množinu, ze které se AP snaží syntetizovat vhodné řešení. Hlavní princip AP je založen na manipulaci s diskrétními množinami (discrete set handling, DSH). DSH vytváří univerzální rozhraní mezi evolučním algoritmem a problémem, který má být symbolicky vyřešen. Proto může AP využívat téměř jakýkoli evoluční algoritmus.

U evolučních algoritmů určuje interval hodnot, pro které je účelová funkce definována Specimen [1]. Populace se náhodně mění v těchto mezích. Někdy jsou požadovány pouze diskrétní hodnoty. V tomto případě populace pracuje s indexy. Budeme mít množinu diskrétních hodnot např. {-1.2, 25.611, 110, 425, … ..}. Pro každé číslo z této množiny je vytvořen index. Index určuje pořadí čísla v množině. Při vytváření populace jedinců se pracuje s indexy. Při vyhodnocení účelové funkce se místo indexu vloží hodnota, na kterou index ukazuje tak jak je to na obrázku (Obr. 6).



Obr. 6 Schéma DSH

1.3.2 Základní množina – GFS

Množina matematických objektů je tvořena funkcemi, operátory a tzv. terminály. Tato množina se nazývá základní množina (general functional set – GFS). AP rozděluje GFS na několik podmnožin podle počtu argumentů jednotlivých funkcí. Na proměnné a konstanty můžeme pohlížet jako na funkce, které nemají žádný argument. Obsah GFS závisí pouze na uživateli. Různé funkce a terminály mohou být smíchány dohromady. Například GFS_{all} je množina všech funkcí, operátorů a terminálů, GFS_{2arg} je podmnožina, která obsahuje pouze funkce se dvěma argumenty, GFS_{0arg} reprezentuje pouze terminály.

GFS_{all} = {+,-,*,/,Sin,Cos, Abs, Mod, x, t, …}

GFS_{2arg} = {+,-,*,/, Mod, …}

GFS_{1arg} = {Sin, Cos, Abs, …}

GFS_{0arg} = {x, t, …}

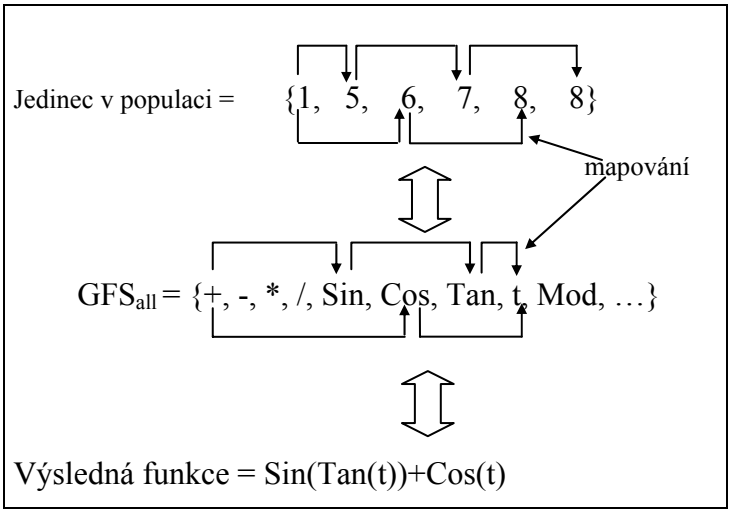
GFS nemusí být vytvořena pouze z čistě matematických funkcí, ale může obsahovat také uživatelsky definované funkce např. funkce, které reprezentují elementy elektronických obvodů nebo příkazy pro řízení robota.

1.3.3 Mapovací operace

Důležitou částí AP je sekvence matematických operací, které jsou použity pro programovou syntézu. Tyto operace transformují jedince (existujícího v evolučním algoritmu) na použitelný program (funkci). Matematicky řečeno, jedná se o mapování z prostoru jedinců na prostor programů (funkcí). Toto mapování se skládá ze dvou částí, z DSH a bezpečnostních procedur, které nedovolí vytvářet patologické jedince.

DSH vytvoří celočíselný indexer, který je použit v evolučním procesu jako alternativní jedinec, na který se namapují libovolné matematické objekty obsažené v GFS. Tento indexer je tedy v podstatě vektorem ukazatelů na jednotlivé prvky GFS. Evoluční algoritmus tak vnímá jedince jako vektor diskrétních hodnot a pouze pro potřebu jeho ohodnocení se provede přemapování na konkrétní funkci, která je použita k výpočtu vhodnosti jedince [10].

AP je tedy v podstatě sérií funkčních mapování. Přitom je ale nezbytné dodržet pravidla, která zajistí, že každý jedinec bude reprezentovat jednoznačnou a nedefektní funkci. Jak takové mapování v praxi funguje ukazuje příklad na obrázku (Obr. 7).



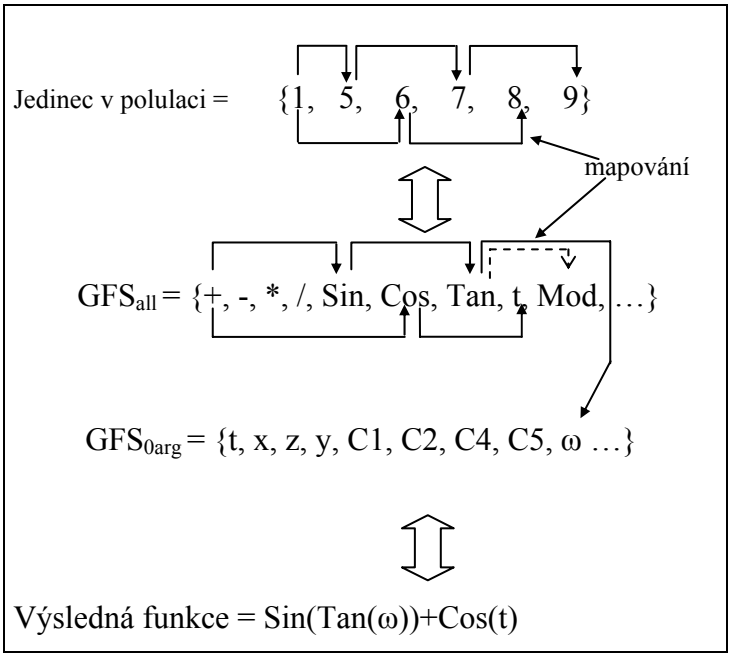
Obr. 7 Mapování v AP

Číslo 1 na pozici prvního parametru znamená, že je použit operátor + z GFS. Operátor + musí mít dva argumenty, pro jeho argumenty jsou určeny ukazatele 5 (*Sin*) a 6 (*Cos*).

Funkce *Sin* a *Cos* jsou funkcemi jedné proměnné. Pro *Sin* určuje jeho argument ukazatel 7 (*Tan*) a pro *Cos* ukazatel 8 (t). Protože je jako argument pro *Cos* zvoleno t, je tato větev výsledné funkce uzavřena (t nemá žádný argument). Zůstává nám funkce jednoho argumentu *Tan* a jelikož další nepoužitý ukazatel je opět 8, *Tan* je namapován na t, které je na 8. pozici v GFS.

1.3.4 Bezpečnostní procedury

Abychom zabránili vytvoření patologických funkcí, musí AP obsahovat několik bezpečnostních opatření. To nejdůležitější je založeno na skutečnosti, že GFS je rozdělena do podmnožin obsahujících funkce o stejném počtu argumentů. Díky tomu je možné vytvořit speciální bezpečnostní proceduru, která měří jak daleko od konce jedince se AP při mapování nalézá a podle toho volí z vhodné podmnožiny GFS, aby tak předešla patologické - neuzavřené - funkci. Pokud je funkcí požadováno více argumentů než kolik ukazatelů jedinec obsahuje (blíží se konec jedince), je vybrána funkce z podmnožiny funkcí s menším počtem argumentů. Stejný ukazatel je přesměrován z GFS_{all} například do GFS_{0arg}, takovýto případ je uveden na obrázku (Obr. 8).



Obr. 8 Bezpečnostní procedury v AP

1.3.5 Křížení, mutace a ostatní evoluční operátory

V průběhu evoluce dané populace možných řešení se používá různých evolučních operátorů, jako jsou například křížení a mutace. AP na rozdíl od GP a GE nepracuje s žádnými evolučními operátory. Ty jsou plně v kompetenci použitého evolučního algoritmu.

Úspěšnost při hledání řešení pomocí AP je možné na straně AP ovlivnit pouze vhodnou volbou GFS. Na straně evolučnho algoritmu je pak množina uživatelem nastavovaných parametrů závislá na konkrétním evolučním algoritmu. Jeden klíčový argument však musí mít všechny EA stejný. Je jím daná účelová funkce (Cost Function, CF), kterou je třeba minimalizovat.

1.3.6 Posílená evoluce

V průběhu evoluce jsou syntetizováni méně či více úspěšní jedinci. Některé jedince přitom můžeme použít k posílení evoluce, abychom získali ještě lepší řešení. Hlavní myšlenka spočívá v přidání již vytvořeného a částečně úspěšného jedince do množiny terminálů. Rozhodnutí, který jedinec bude takto použit je závislé na uživatelem definovaném prahu.

Například pokud je práh nastaven na 5 a všichni jedinci v populaci mají větší hodnotu CF než 5, běží evoluce dál na základní GFS. Pokud je ovšem nejlepší jedinec v aktuální populaci lepší než 5, pak je kompletně přidán do GFS a je označen jako terminál. Od této chvíle pracuje AP s rozšířenou GFS obsahující částečně úspěšná řešení. Díky tomu je posílené AP schopno najít vhodné řešení mnohem rychleji než AP bez posílené evoluce. Tento fakt byl verifikován mnoha tisíci úspěšných simulací prováděných na nejrůznějších řešených problémech [10].

1.3.7 Cost Function v AP

Pomocí AP se vytváří nové funkce – programy. AP může být použito pro nalezení funkce, která co nejlépe prokládá dané body. Vhodná účelová funkce pro takové případy je rovna absolutní hodnotě rozdílu originální hodnoty dat a právě nalezené funkce. Cílem AP je minimalizovat tento rozdíl. V nejlepším případě by se hodnota účelové funkce měla co nejvíce blížit nule. Pro každý bod zvlášť se vypočítá rozdíl originální hodnoty a hodnoty nalezené funkce v tomto bodě. Hodnota účelové funkce je rovna součtu absolutních hodnot těchto rozdílů.

2 EVOLUČNÍ ALGORITMY

Pro běh AP je zapotřebí nějakého evolučního algoritmu. V následující části jsou popsány evoluční algoritmy SOMA a Diferenciální evoluce, které jsou pak použity v programu.

Evoluční algoritmy (EA) jsou používány pro nalezení nejlepšího řešení v optimalizačních problémech. Mnoho inženýrských problémů může být definováno jako optimalizační problém. Optimalizační problém spočívá v nalezení globálního extrému (minima nebo maxima).

2.1 Základní pojmy

Před popisem EA je potřeba vysvětlit některé pojmy. Výrazem „účelová funkce“ se rozumí funkce, kterou optimalizujeme. Optimalizace spočívá v nalezení minima nebo maxima. Účelová funkce se označuje f(x) nebo také $f_{cost}(x)$.

Populace je množina jedinců. Každý jedinec představuje aktuální řešení daného problému. V podstatě je to množina argumentů účelové funkce, jejíž optimální číselná kombinace je hledána. S každým jedincem je spojena hodnota účelové funkce (CV – cost value), která v podstatě říká, jak vhodný je jedinec pro další vývoj populace.

K vytvoření populace je potřeba nadefinování tzv. vzoru (Specimen), podle kterého se vygeneruje celá počáteční populace. Specimen se také používá pro korekci jedinců. Pro každý parametr jedince jsou definovány hranice intervalu, v němž je hodnota každého parametru jedince a typ proměnné (celočíselný, reálný apod.).

2.2 SOMA

Samoorganizující se Migrační Algoritmus (SOMA) je algoritmus existující od roku 1999 [1]. Jeho činnost je založena na geometrických principech. SOMA pracuje s populacemi podobně jako ostatní evoluční algoritmy, ale během jeho běhu nejsou vytvářeni noví potomci.

2.2.1 Princip SOMA

Vznik SOMA byl inspirován soutěživě-kooperativním chováním inteligentních jedinců řešících společný problém. Chování tohoto typu lze objevit prakticky kdekoli na světě. Jako příklad lze použít chování smečky lovících vlků, včelího úlu, termitích kolonií apod.

U těchto příkladů je společným úkolem např. hledání potravy, v rámci níž jedinci spolupracují, ale i, byť nevědomky, soutěží. Ve fázi spolupráce si navzájem jednotliví jedinci sdělují jakou kvalitu hledaného momentálně nalezli a na základě toho se snaží přizpůsobovat své chování. Ve fázi soutěžení (předcházející fázi spolupráce) se každý jedinec snaží vyhrát nad ostatními - snaží se nalézt co nejlepší zdroj potravy apod. Po ukončení fáze soutěže nastane opět fáze spolupráce a jedinci si vymění informace o tom, který z nich má nejlepší zdroj potravy. Ostatní opustí své nalezené zdroje potravy a migrují (fáze soutěžení) směrem k jedinci s nejlepším zdrojem potravy a během této migrace se snaží nalézt ještě lepší zdroj. To se opakuje dokud se všichni nesejdou u nejvydatnějšího zdroje potravy. Na tomto silně zjednodušeném principu funguje i algoritmus SOMA [1].

2.2.2 Parametry SOMA

Algoritmus SOMA je ovlivňován speciální množinou parametrů, které se dělí na řídící a ukončovací. Řídící parametry mají vliv na kvalitu běhu algoritmu a ukončovací za předem nadefinovaných podmínek běh algoritmu ukončují. Všechny parametry jsou zobrazeny v tabulce (Tab. 1).

Tab. 1 Parametry SOMA

Parametr	Doporučený rozsah	Význam parametru
Mass	<1.1,5>	určuje, jak daleko se aktivní jedinec zastaví od tzv. vedoucího jedince
Step	<0.11,Mass>	určuje po jakých krocích bude mapována cesta aktivního jedince
PRT	<0,1>	perturbace, podle ní se tvoří perturbační vektor
D	dáno problémem	počet argumentů účelové funkce
NP	<10,definuje uživatel>	počet jedinců v populaci
Migrace	<10,definuje uživatel>	udává kolikrát se populace jedinců přeorganizuje
AcceptedError	<±libovolný, def. uživatel>	ukončovací parametr, definuje maximální rozdíl mezi nejhorším a nejlepším jedincem

2.2.3 Fáze SOMA

Vzhledem k tomu, že SOMA je založen na již zmíněném principu, jsou jeho jednotlivé varianty nazývány strategiemi. Základní verze SOMA (strategie AllToOne) se skládá z následujících kroků:

1. Definice parametrů. Před startem SOMA je nutné nadefinování řídících a ukončovacích parametrů (Specimen, Step, Mass, Accepted Error, NP, PRT a Migrace). Nezbytně nutné je rovněž nadefinovat účelovou funkci, která bude optimalizována. Účelová funkce by měla vracet skalár, který bude použit jako měřítko kvality daného jedince.

2. Tvorba populace. V tomto kroku je vytvořena prvopočáteční populace. Za pomoci specimenu a generátoru náhodných čísel je pro každý parametr jedince generováno náhodné číslo.

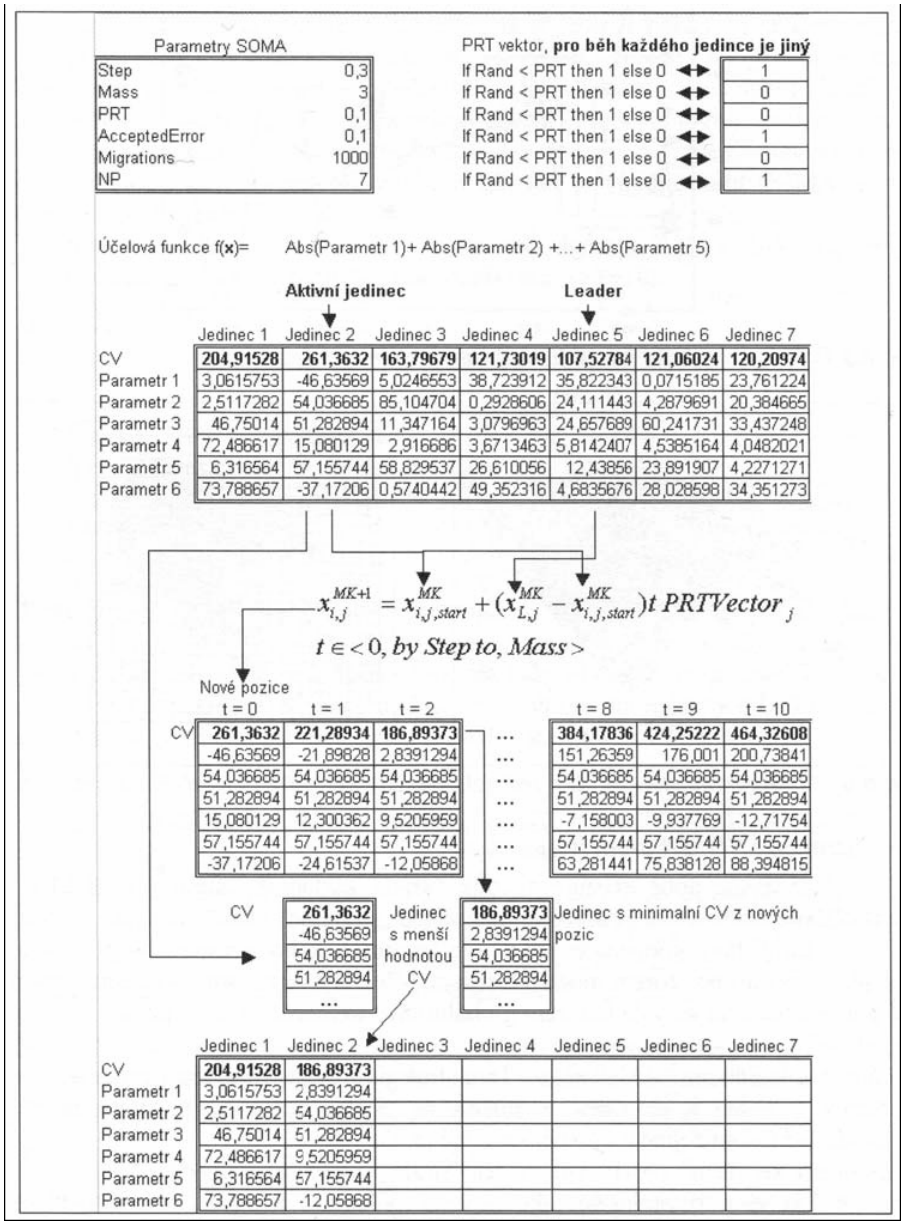
3. Migrační kola. Každý jedinec je ohodnocen účelovou funkcí a je zvolen Leader (jedinec s nejlepší hodnotou účelové funkce) pro následující migrační kolo. V tomto okamžiku se začnou ostatní jedinci pohybovat směrem k Leaderovi pomocí skoků, jejichž velikost je dána parametrem Step. Po každém skoku si každý jedinec na takto získané nové pozici přepočítá svou hodnotu účelové funkce a pokud je lepší než předchozí, tak si ji zapamatuje. Pohyb jedince směrem k Leaderovi po skocích pokračuje tak dlouho, dokud není dosaženo pozice, jež je dána parametrem Mass. Po ukončení běhu se jedinec vrací na pozici, kde byla nalezena nejlepší hodnota účelové funkce během jeho cesty.

Předtím, než daný jedinec započne svou cestu směrem k Leaderovi, je vygenerován prázdný PRTVector o dimenzi = D včetně vygenerované sekvence náhodných čísel, jejichž počet je roven D. Ty jsou porovnány s parametrem PRT. Jestliže je n-té vygenerované číslo větší nežli PRT parametr, pak je n-tý parametr PRTVectoru nastaven na 0 a v opačném případě na 1. Parametry jedince, které jsou takto nastaveny na 0 se nepřepočítávají, jsou zmraženy - snižuje se počet stupňů volnosti pohybu jedince. Tento proces nahrazuje mutaci známou u jiných evolučních algoritmů. Díky tomu se rapidně zvyšuje robustnost SOMA algoritmu ve smyslu nalezení globálního extrému.

4. Testování naplnění ukončovacích parametrů. V tomto okamžiku je kontrolováno, zda je rozdíl mezi Leaderem (nejlepší jedinec) a nejhorším jedincem menší nežli AcceptedError. Také je kontrolováno, zda byly vykonány migrační cykly v počtu, jenž je

dán parametrem Migrace. Pokud není splněna ani jedna podmínka, proces se vrací na krok 3.

5. Stop. Návrat nejlepšího jedince - nejlepší nalezené řešení po posledním migračním kole.



Obr. 9 Princip SOMA (převzato z [1])

2.2.4 Strategie SOMA

V současné době existuje několik variací základního algoritmu SOMA, pro jejichž obecné označení se používá rovněž výraz „strategie“.

„Všichni k jednomu“ (AllToOne). Tato strategie již byla popsána v předchozí sekci. Název „Všichni k jednomu“ znamená, že všichni jedinci z populace migrují k Leaderovi.

„Všichni ke všem“ (AllToAll). V této strategii neexistuje Leader. Všichni jedinci migrují ke všem ostatním. Tato strategie výpočetně náročnější, ale je zde vyšší pravděpodobnost, že bude nalezen globální extrém, neboť se během migrace jedince prohledá větší část prostoru.

„Adaptivně všichni ke všem“ (AllToAllAdaptive). Tato strategie je totožná se strategií „Všichni ke všem“ s tím rozdílem, že aktuálně migrující jedinec se přesouvá do nové pozice po každé aktuálně dokončené migraci. Z této pozice pak provádí migraci k dalším zbý-vajícím jedincům.

„Všichni k jednomu náhodně“ (AllToOneRand) je strategie ve které se všichni jedinci pohybují opět k jednomu Leaderovi, který ovšem není nejlepším jedincem, ale je pro migraci každého jedince náhodně vybrán z populace.

„Svazky“ (Clusters). SOMA s vytvářením svazků je úprava, která se dá použít na kteroukoliv předchozí strategii. Jedinci účastnící se migračního procesu jsou rozděleni do tzv. svazků. V každém z nich pak probíhá samostatný SOMA.

2.3 Diferenciální evoluce

Diferenciální evoluce (*Differential Evolution*, DE) je stejně jako SOMA poměrně nový typ EA (od r. 1995), který vyvinul a poprvé použil Ken Price a Rainer Storm. Jeho schéma je dost podobné algoritmům genetickým, s nimiž má několik společných rysů, jako je například tvorba potomků (zde však pomocí 4 rodičů a ne 2 jak je tomu u genetických algoritmů), používání tzv. generací apod.

2.3.1 Parametry DE

Také činnost a kvalita Diferenciální evoluce je ovlivněna jejími parametry. Jejich označení a význam je uveden v tabulce (Tab. 2).

Tab. 2 Parametry Diferenciální evoluce

Parametr	Doporučený rozsah	Význam parametru
CR	<0,1>	práh křížení
NP	<2D,100D>	velikost populace
F	<0,2>	mutační konstanta
Generations	dáno problémem	počet kol šlechtění populace
D	<10,definuje uživatel>	počet argumentů účelové funkce

2.3.2 Princip činnosti Diferenciální evoluce

Cílem Diferenciální evoluce je v cyklech zvaných „generace“ vyšlechtit co nejlepší populaci jedinců ve smyslu hodnot účelové funkce, jenž je spojena s každým jedincem. Princip činnosti je znázorněn také na obrázku (Obr. 10). Během každé generace se provádí následující kroky:

1. Stanovení parametrů - jde o parametry, které určují chod evoluce. Jsou to parametry F - mutační konstanta <0, 2>, CR - práh křížení <0, 1>, NP - počet jedinců v populaci, D - rozměr jedince (jsou to v podstatě argumenty účelové funkce). Dále je nutné nadefinovat prototyp jedince - Specimen.

2. Tvorba populace - populace se tvoří vygenerováním množiny jedinců (matice) podle prototypového (Specimen) vektoru. U každého jedince se musí počítat s jedním prvkem navíc a tím je hodnota účelové funkce.

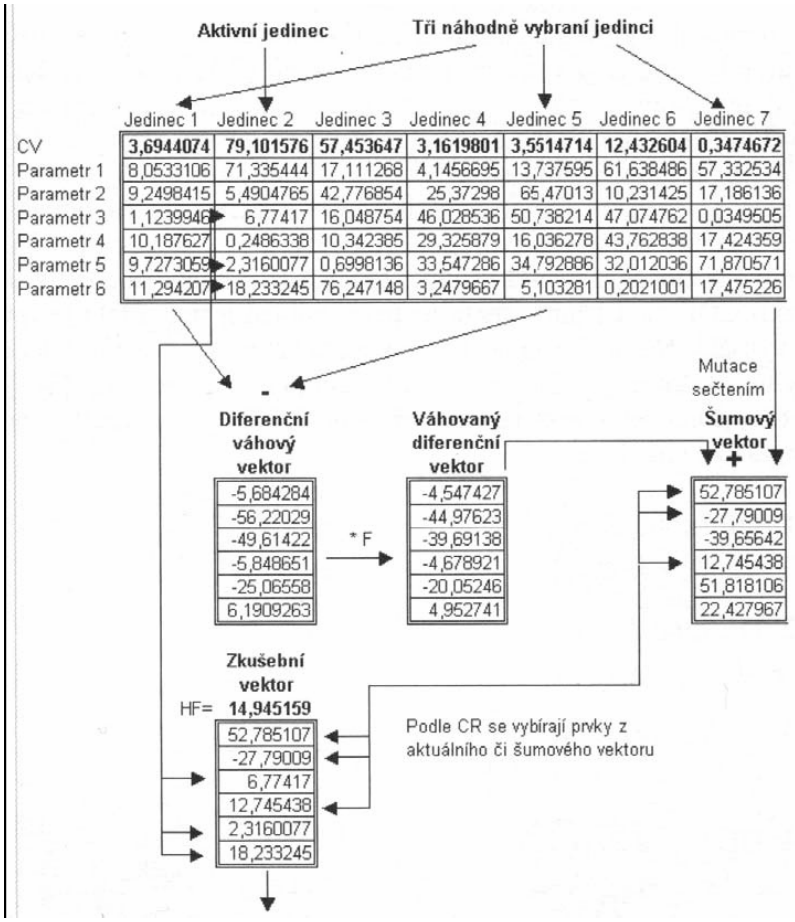
3. Započetí cyklu generace - během každé generace se provádí ještě cyklus, který zabezpečuje postupné evoluční šlechtění každého jedince z populace. V tomto cyklu se postupně vybírá jeden jedinec za druhým až do konce populace a pro každého z nich je proveden následující evoluční cyklus.

4. Evoluční cyklus - v tomto cyklu je prováděna mutace a křížení. Vedle cílového vektoru se náhodně zvolí tři další různé vektory (jedinci) z populace. První dva se od sebe odečtou a získá se tak tzv. diferenční vektor. Ten se vynásobí mutační konstantou „*F*“, která jej tím pádem změní (zmutuje), a získá se „váhovaný diferenční vektor“. Ten se přičte k třetímu náhodně vybranému vektoru a získá se tzv. „šumový vektor“. Poté se připraví tzv. „zkušební vektor“ a z cílového a šumového vektoru se bere postupně jeden prvek za

druhým a pro takto vybranou každou dvojici se generuje náhodné číslo v rozsahu 0-1 a porovnává s konstantou CR. Pokud je toto číslo menší než CR, pak se do příslušné pozice v tzv. „zkušebním vektoru“ umístí prvek z vektoru šumového a v opačném případě z vektoru cílového. Tak se získá zkušební vektor, jehož hodnota účelové funkce se porovná s hodnotou účelové funkce cílového vektoru. Na pozici cílového vektoru v nové populaci je vybrán ten vektor (jedinec), který má hodnotu účelové funkce lepší. Tím je zajištěno, že se do nové generace dostanou jedinci s lepšími nebo stejnými vlastnostmi. Celý evoluční cyklus se opakuje až do vyčerpání populace.

5. Testování naplnění ukončovacích parametrů - Diferenciální evoluce je ukončena pouze tehdy, provede-li se uživatelem zadaný počet generací. Jiný ukončovací parametr tento algoritmus nemá.

6. Vyhodnocení - celý proces generací se opakuje, dokud není vyčerpán zadaný počet generací. Během každé generace se uschová hodnota účelové funkce nejlepšího jedince do vektoru historie, který po ukončení znázorňuje průběh evolučního procesu.



Obr. 10 Princip Diferenciální evoluce 9 (převzato z [1])

II. PRAKTICKÁ ČÁST

3 IMPLEMENTACE

AP je implementováno v jazyce C#. C# je čistě objektově orientovaný jazyk na platformě .NET, který kombinuje vlastnosti známých a oblíbených programovacích jazyků a přidává k nim některé své nové vlastnosti. Jazyk C# se nám tak ideálně hodí pro náš vývoj a implementaci AP.

3.1 Uživatelsky definované funkce

AP pracuje s množinou funkcí a terminálů - GFS. Aby uživatel nemusel pracovat pouze se standardními matematickými funkcemi, může si všechny ostatní funkce, které bude chtít v GFS používat naprogramovat sám.

K tomu, aby si uživatel mohl pro aplikaci naprogramovat vlastní funkce je využito technologie pluginů (zásuvných modulů). Pluginy slouží k tomu, aby dodávaly již hotové aplikaci nějakou novou funkcionalitu, aniž by bylo potřeba mít přístup ke zdrojovým kódům aplikace a nějak do nich zasahovat.

Nejprve bylo nutné nadefinovat rozhraní, přes které bude probíhat komunikace. Od tohoto rozhraní je odvozena třída pluginu. Rozhraní musí být v oddělené dll knihovně, aby jej mohly používat pluginy nezávisle na hlavní aplikaci.

Rozhraní je umístěno v knihovně `PlugInterf`. Funkce, které budou v GFC jsou rozděleny do tří skupin. Funkce s jedním argumentem, funkce se dvěma argumenty a funkce s více než dvěma argumenty. Funkce s jedním argumentem jsou implementovány pomocí rozhraní `IMathPrim`, funkce se dvěma argumenty pomocí rozhraní `IMathDual` a funkce s více než dvěma argumenty pomocí rozhraní `IMath`. Každé rozhraní obsahuje pouze jednu metodu `operace()`. Tato metoda vrací číslo typu `double`, které je výsledkem operace, kterou si uživatel naprogramuje.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PlugInterf
{
    public interface IMathPrim
    {
       double operace(double x);
    }
    public interface IMathDual
    {
       double operace(double x, double y);
    }

    public interface IMath
    {
        double operace(double[] pole);
    }
}
```

Po vytvoření rozhraní je možné naprogramovat plugin (dll knihovnu). Stačí vytvořit nový projekt typu „Class library“ a v ní třídu, která je odvozená od rozhraní. Tato knihovna musí mít v záložce References vložen odkaz na knihovnu `PlugInterf.dll`, ve které je specifikováno rozhraní. Knihovna `MojeOperace.dll` obsahuje některé základní funkce.

```
using PlugInterf;


public class Plus : IMathDual
{
    public double operace(double x, double y) { return x + y; }
}

public class Sin : IMathPrim
{
    public double operace(double x) { return Math.Sin(x); }
}
```

```
public class Soucet : IMath
{
    public double operace(double[] pole)
    {
        double vysledek = 0;
        for (int i = 0; i < pole.Length; i++)
        {
            vysledek += pole[i];
        }
        return vysledek;
    }
}
```

Knihovna s rozhraním je společná jak pro hlavní aplikaci, tak pro pluginy. V hlavní aplikaci proto musí být také uveden odkaz v References na tuto knihovnu.

3.2 Třídy

K tomu, aby pomocí AP mohl být sestaven nějaký výraz je třeba znát GFS, jedince a hodnoty proměnných.

3.2.1 GFS

Pro reprezentaci množiny GFS je vytvořena třída `GFS`.

```
class GFS
    {
        private PrvekGFS[] pole;
        private int pocet;
        private int pocProm;
        private string[] promenne;

        public GFS(int p){…}
        public void VstupPrvku(string r1, string r2){…}
        public int VratPocet(){…}
        public int VratPocProm(){…}
        public PrvekGFS VratPrvek(int i){…}
        public string VratPromenna(int i){…}
    }
```

V poli `pole` jsou uloženy jednotlivé členy GFS. Každý prvek pole představuje jednu funkci nebo terminál. Struktura `PrvekGFS` popisuje danou funkci nebo terminál.

```
struct PrvekGFS
    {
        private string s;
        private int arg;
        private double h;
    }
```

Řetězec `s` označuje název funkce nebo terminálu, v proměnné `arg` je uložen počet argumentů funkce (pro terminály je v `arg` uložena 0). V případě konstant je v proměnné `h` uložena jejich hodnota.

3.2.2 Jedinec

Jedinec představuje v AP výraz, který je vytvořený z prvků množiny GFS. Jednotlivé členy jedince ukazují do množiny GFS. Protože EA předává jedince jako pole reálných čísel, je nutné tato čísla převést na čísla celá v rozsahu od 0 do počtu prvků množiny GFS. V metodě `VstupJedince(double[] p)` se každý prvek pole nejprve převede na kladné číslo a zaokrouhlí se. Jestliže je výsledné číslo větší nebo rovno počtu prvků GFS, tak se od něj tak dlouho tento počet odečítá dokud není menší.

```
class Jedinec
    {
        private int[] pole;
        private int pocet;
        private int pocetGFS;
    }
```

```
public void VstupJedince(double[] p)
        {
            this.pole = new int[p.Length];
            double pom;
            for (int i = 0; i < p.Length; i++)
            {
                pom =Math.Abs(p[i]);
                pom = Math.Round(pom);
                while (pom >= pocetGFS)
                {
                    pom -= pocetGFS;
                }
                pole[this.pocet] = (int)pom;
                this.pocet++;
            }
        }
```

3.2.3 Hodnoty

Třída `Hodnoty` slouží k uložení bodů, kterými má procházet hledaná funkce.

```
class Hodnoty
    {
        private double[,] pole;
        private int pocet;
        public int dim;
    }
```

V proměnné `pole` jsou uloženy souřadnice jednotlivých bodů, v proměnné `pocet` je uložen počet bodů a v proměnné `dim` počet souřadnic bodu.

3.2.4 Výraz

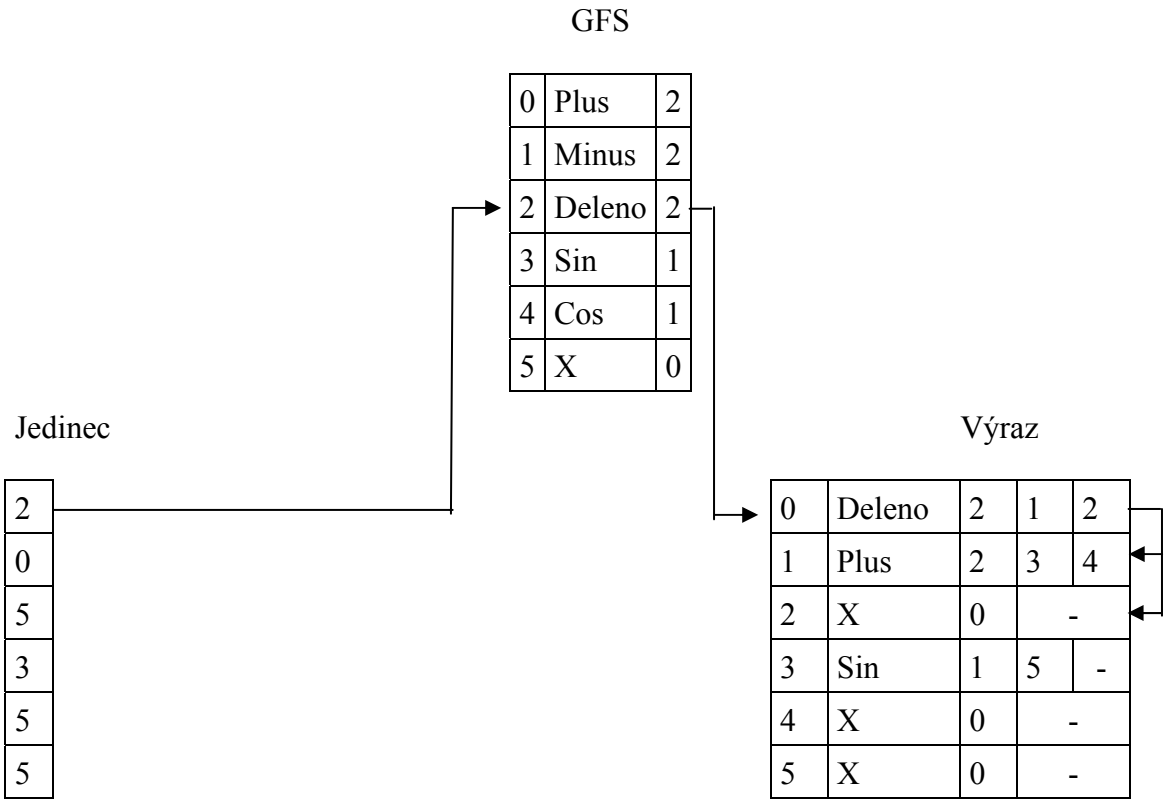
Jakmile známe GFS, jedince a hodnoty proměnných, můžeme pomocí AP sestavit požadovaný výraz. Poté za proměnné dosadíme konkrétné hodnoty a určíme tak CV (cost value) daného jedince. Výraz se ukládá do pole. Jednotlivé prvky pole jsou funkce nebo terminály z GFS, na které ukazují jednotlivé členy jedince. Podle toho kolik má daná funkce argumentů, tolik prvků má pole `pole_odkazu`. Tady se ukládají indexy prvků v poli výrazu, které jsou argumenty funkce.

3.2.5 Sestavení výrazu

Při sestavování výrazu se bere jeden člen jedince z druhým a podle jeho hodnoty se určí na jaký prvek v GFS ukazuje. Do pole výrazu se vloží na první volné místo název funkce a počet argumentů. Pokud má funkce nějaké argumenty, uloží se do `pole_odkazu` indexy prvků pole výrazu, kde se budou nacházet argumenty. V proměnné `poslední_odkaz` je uložen poslední použitý index.

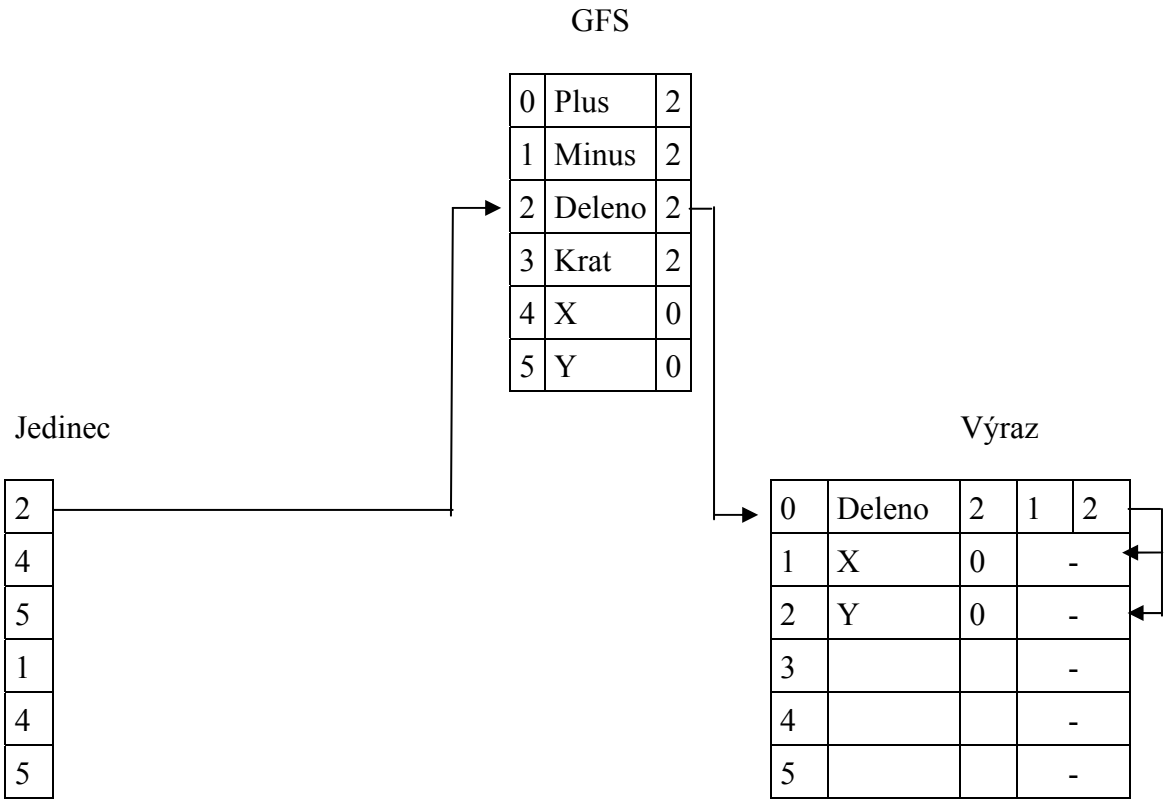
```
public void VlozPrvek(string s, int arg, double h)
       {   if (!stop)
           {   if (this.index == this.posledni_odkaz)
               stop = true;
               else  {
                     VratPrvek(index).VlozRetezec(s);
                     VratPrvek(index).VlozArg(arg);
                     VratPrvek(index).VlozHodnota(h);
                     for (int i = 0; i < arg; i++)
                     {   VratPrvek(index).VlozOdkaz(i, posledni_odkaz);
                         posledni_odkaz++; }
                     this.index++; }
           }
       }
```

V proměnné `index` je index aktuálního prvku pole výrazu. Jak vypadá sestavení výrazu je zobrazené na obrázku (Obr. 11).



Obr. 11 Sestavení výrazu - 1

Sestavování výrazu může skončit dříve než jsou vyčerpány všechny parametry jedince. Stane se tak v případě, že všechny funkce už mají své argumenty, jak je vidět na obrázku (Obr. 12). V tomto případě zůstaly poslední tři parametry nevyužity.



Obr. 12 Sestavení výrazu - 2

3.2.6 Oprava patologických výrazů

Jedinec může představovat i špatně vytvořený výraz. Může se stát, že některé funkce nebudou mít argumenty. To znamená, že indexy odkazů budou větší než je celkový počet prvků výrazu. Proto je nutné před vyhodnocením výrazu provést opravu. Oprava spočívá v tom, že se v cyklu prochází jeden prvek výrazu za druhým a pokud je nějaké číslo v `pole_odkazu` větší než celkový počet prvků výrazu je tento konkrétní prvek výrazu nahrazen terminálem. Jinak by argumentem byl neexistující prvek a výraz by nebylo možné vyhodnotit. Oprava se provádí v metodě `Oprava()`.

```
public void Oprava()
  {
      for (int i = 0; i < this.index; i++)
      {
          for (int j = 0; j < VratPrvek(i).VratArg(); j++)
          {
              if (VratPrvek(i).VratOdkaz(j) >= this.index)
              {
                  VratPrvek(i).VlozArg(0);
                  VratPrvek(i).VlozRetezec(this.prom);
              }
          }
      }
  }
```

V tabulce (Tab. 3) je sestaven výraz (X-Y)*(?+?). Funkce Plus nemá žádné argumenty, proto bude nahrazena terminálem X. Výsledný výraz je potom (X-Y)+X.

Tab. 3 Oprava výrazu

index	Název – s	arg	hodnota	p	pole_odkazu	
0	Krat	2	0	true	1	2
1	Minus	2	0	true	3	4
2	~~Plus~~ X	~~2~~ 0	0	true	~~5~~	~~6~~
3	X	0	0	true	-	
4	Y	0	0	true	-	

3.2.7 Ohodnocení jedince – výpočet výrazu

Abychom mohli provést ohodnocení jedince, je nutné dosadit za všechny proměnné konkrétní hodnoty a vypočítat daný výraz. Dosazení hodnoty za proměnnou se provádí v metodě `UlozHodnotu(string r, double h)`. Výraz se prochází po jednotlivých prvcích a pokud se jedná o proměnnou představovanou řetězcem `r`, je do `h` uložena hodnota proměnné.

```
public void UlozHodnotu(string r, double h)
  {
      for (int i = 0; i < this.index; i++)
      {
          if (VratPrvek(i).VratArg() == 0)
          {
              if (VratPrvek(i).VratRetezec() == r)
              {
                  VratPrvek(i).VlozHodnota(h);
              }
          }
      }
  }
```

Po uložení hodnoty je možné provést výpočet, který probíhá v metodě `Vypocet()`. Nejprve musíme načíst assembly z daného souboru. V proměnné `knihovna` je uložen název souboru.

```
Assembly assembly = Assembly.LoadFrom(knihovna);
```

Dále musíme vytvořit objekty pro každé rozhranní.

```
IMathDual plugin;
IMathPrim plugin1;
IMath plugin2;
```

Pole `poleVyraz` se prochází od posledního prvku. Jakmile narazím na funkci, která ještě nebyla vyhodnocena tak se do pomocných argumentů uloží konkrétní hodnoty a dynamicky se vytvoří objekt daného typu. Pak už lze volat metodu pluginu `operace()`, které se předají argumenty a výsledek se uloží pomocí `VlozHodnota()` do `h`. Proměnná `p` se nastaví na `false` a tím bude zajištěno, že se daná funkce nebude počítat znova.

```
if ((VratPrvek(i).VratArg() == 2)&&(VratPrvek(i).VratP()))
{
  arg1 = VratPrvek(VratPrvek(i).VratOdkaz(0)).VratHodnota();
  arg2 = VratPrvek(VratPrvek(i).VratOdkaz(1)).VratHodnota();
  t = assembly.GetType(VratPrvek(i).VratRetezec());
  plugin = (IMathDual)Activator.CreateInstance(t);
  v=plugin.operace(arg1, arg2);
  VratPrvek(i).VlozHodnota(v);
  VratPrvek(i).VlozP(false);
}
```

Výsledná hodnota výrazu je uložena v proměnné h prvního prvku. V následujícím příkladě je ukázáno vytvoření a výpočet výrazu.

GFS = {Plus, Minus, Krat, Deleno, x, y, 2, 3.14}

Jedinec = {3, 7, 1, 5, 6}

x = 10, y = 14

V *Tab.1* je uveden vytvořený výraz : 2 * (x + y)

Tab. 4 Vytvořený výraz

index	Název – s	arg	hodnota	p	pole_odkazu	
0	Krat	2	0	true	1	2
1	2	0	2	true	-	
2	Plus	2	0	true	3	4
3	X	0	0	true	-	
4	Y	0	0	true	-	

Za proměnné x a y jsou dosazeny hodnoty 10 a 14.

Tab. 5 Dosazení hodnot za proměnné

index	název – s	arg	hodnota	p	pole_odkazu	
0	Krat	2	0	true	1	2
1	2	0	2	true	-	
2	Plus	2	0	true	3	4
3	x	0	10	true	-	
4	y	0	14	true	-	

V cyklu se pole prochází od konce doku nenarazím na první funkci, která ještě nebyla vyhodnocena. V našem případě je to funkce Plus. Tato funkce má 2 argumenty, které jsou na místech 3 a 4. Provede se výpočet funkce s hodnotami 10 a 14. Výsledek 24 se uloží do hodnoty u funkce Plus a p se nastaví na *false*.

Tab. 6 Vyhodnocování výrazu – 1 krok

index	název – s	arg	hodnota	p	pole_odkazu	
0	Krat	2	0	true	1	2
1	2	0	2	true	-	
2	Plus	2	24	false	3	4
3	x	0	10	true	-	
4	y	0	14	true	-	

Další nevyhodnocená funkce je funkce Krat. Tato funkce má také 2 argumenty, které jsou umístěny na 1 a 2 místě. Index 2 má funkce Plus, která byla vyhodnocena v předcházejícím kroku, proto známe její hodnotu. Funkce Krat je vyhodnocena s argumenty 2 a 24. Výsledek 48 je uložen do proměnné hodnota a p je nastaveno na *false*. Všechny funkce už byly vyhodnoceny a výsledek je uložen v prvku s indexem 0 v proměnné hodnota.

Tab. 7 Vyhodnocování výrazu –2 krok

index	název – s	arg	hodnota	p	pole_odkazu	
0	Krat	2	48	false	1	2
1	2	0	2	true	-	
2	Plus	2	24	false	3	4
3	x	0	10	true	-	
4	y	0	14	true	-	

3.2.8 Cost Value

K tomu, aby mohl evoluční algoritmus správně pracovat je potřeba každého jedince ohodnotit – přiřadit mu CV. Ohodnocení jedince probíhá v metodě `CostValue()` takto: pro každého jedince je vytvořen daný výraz. Do výrazu se postupně dosazují hodnoty každého bodu a provádí se výpočet výrazu. Tím dostaneme pro každý bod nějakou hodnotu výrazu. Dílčí CV pro daný bod je roven absolutní hodnotě rozdílu hodnoty výrazu pro tento bod a očekávané hodnoty. Výsledná CV je rovna součtu všech dílčích CV pro každý bod. Hledaný jedinec je takový, který má CV minimální – co nejblíže roven 0.

```
public double CostValue(double[] ind)
  {
      double cv = 0;
      double pom = 0;
      j = new Jedinec(g.VratPocet());
      j.VstupJedince(ind);
      v = new Vyraz(j.VratPocet(),g.VratPromenna(0),this.knihovna);
      for (int i = 0; i < j.VratPocet(); i++)
      {
          string r = g.VratPrvek(j.VratPrvek(i)).VratString();
          int a = g.VratPrvek(j.VratPrvek(i)).VratArg();
          double hod = g.VratPrvek(j.VratPrvek(i)).VratH();
          v.VlozPrvek(r, a, hod);
      }
      v.Oprava();
      for (int i = 0; i < h.VratPocet(); i++)
      {
          for (int k = 0; k < g.VratPocProm(); k++)
          {
             v.UlozHodnotu(g.VratPromenna(k), h.VratHodnota(i, k));
          }
          pom = h.VratHodnota(i, h.dim - 1) - v.Vypocet();
          pom = Math.Abs(pom);
          cv += pom;
          v.Vynulovani();
      }
      return cv;
  }
```

Funkce `CostValue()` je volána evolučním algoritmem pokaždé, kdy je potřeba ohodnotit nějakého jedince.

4 TESTOVÁNÍ

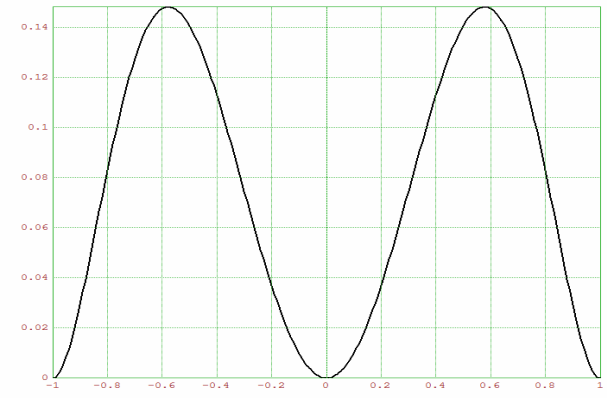
Naprogramovaný software byl otestován na několika funkcích. Protože AP programování potřebuje pro svůj běh nějaký evoluční algoritmus použila jsem algoritmus SOMA All-to-One a Diferenciální evoluci. Hlavní aplikace neobsahuje funkce pro GFS, je proto nutné si nejdříve vytvořit knihovnu, ze které budeme načítat prvky GFS. Pro oba evoluční algoritmy se musí nastavit jejich řídící parametry.

4.1 Testování na vybraných příkladech

Pro potřeby testování jsem vytvořila dvě knihovny. Knihovna `MojeOperace.dll` obsahuje funkce +,-,*,/,sin,cos, tan. Knihovna `LogickeOperace.dll` obsahuje operace And, Or, Nand a Nor.

4.1.1 Testování funkce $x^6 - 2x^4 + x^2$

Graf funkce $x^6 - 2x^4 + x^2$ je na obrázku (Obr. 13).



Obr. 13 Graf funkce $x^6 - 2x^4 + x^2$

Pro každý EA musíme nastavit jeho parametry. Dále je nutné stanovit GFS a body, kterými má funkce procházet.

GFS = {Plus, Minus, Krat, Deleno, x}

Pro x je vygenerováno 20 hodnot v intervalu <-1,1>.

Nastavení parametrů SOMA je uvedeno v tabulce (Tab. 8).

Tab. 8 Nastavení parametrů SOMA

Parametr	Hodnota
Mass	3
Step	0,11
Prt	0,2
D	30
NP	50
Migrace	100

Ve všech 20 testech byla nalezena hledaná funkce.

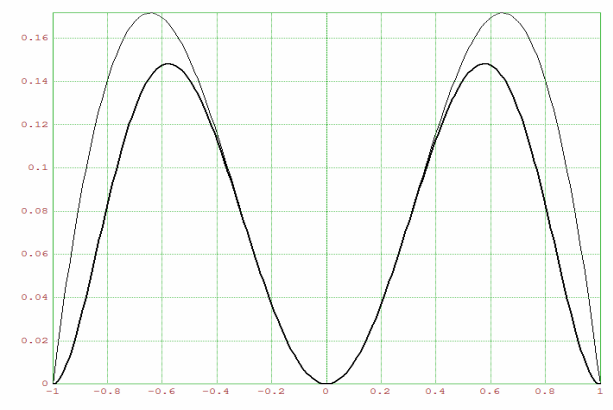
Nastavení parametrů DE je uvedeno v tabulce (Tab. 9).

Tab. 9 Nastavení parametrů pro DE

Parametr	Hodnota
CR	0,8
F	0,31
D	30
NP	100
Migrace	1000

Ve většině případů se opět podařilo nalézt požadovanou funkci. Nejhorší získaná funkce je dána vztahem (1) . Rozdíl mezi hledanou funkcí a funkcí (1) je zobrazen na obrázku (Obr. 14).

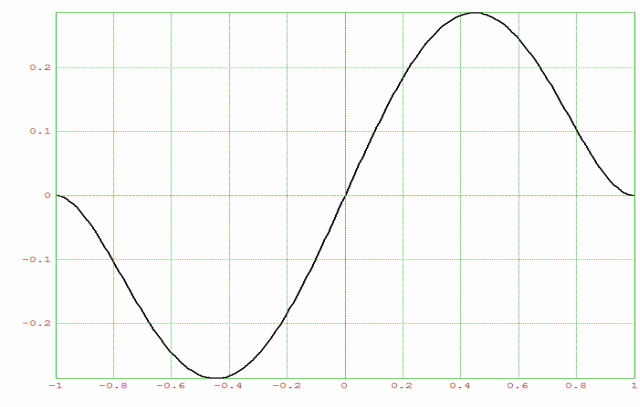
$$\frac{(1-x^2)*x^2}{x^2+1} \tag{1}$$



Obr. 14 Porovnání funkcí

4.1.2 Testování funkce $x^5 - 2x^3 + x$

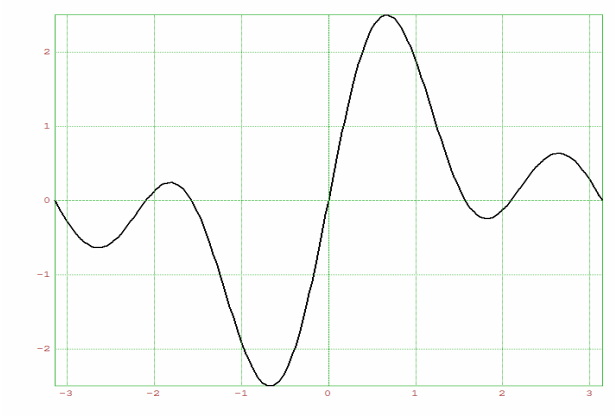
Graf funkce $x^5 - 2x^3 + x$ je na obrázku (Obr. 15). Množina GFS zůstává stejná jako v předchozím případě. Hodnoty x jsou generovány opět z intervalu <-1,1>. Při nastavení parametrů pro SOMA podle tabulky (Tab. 8) a pro DE podle tabulky (Tab. 9), je v každém z 20 testů nalezena požadovaná funkce.



Obr. 15 Graf funkce $x^5 - 2x^3 + x$

4.1.3 Testování funkce $\sin(3x) + \sin(2x) + \sin(x)$

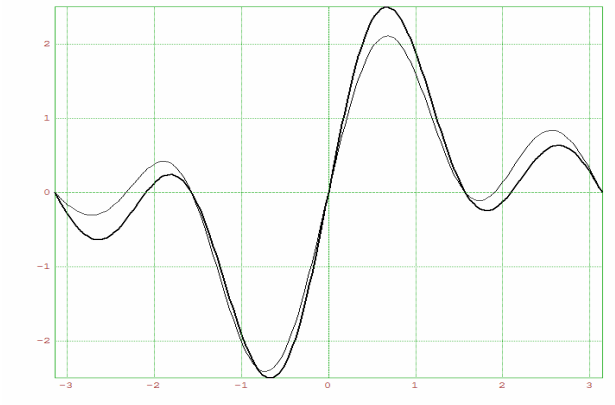
Graf funkce $\sin(3x) + \sin(2x) + \sin(x)$ je na obrázku (Obr. 16). Hodnoty x budou nyní generovány z intervalu <-Π, Π >. GFS rozšíříme o funkce Sin a Cos.



Obr. 16 Graf funkce $\sin(3x)+\sin(2x)+\sin(x)$

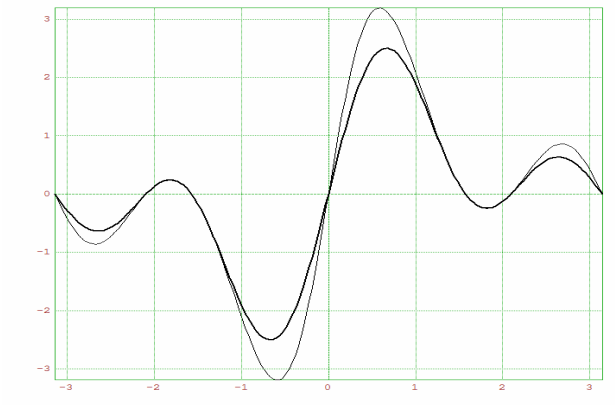
Nastavení evolučních algoritmů bylo stejné jako v předcházejících případech. Nalezené funkce většinou velmi dobře odpovídaly zadaným bodům. Některé funkce jsou zobrazeny na následujících obrázcích v porovnání s grafem hledané funkce.

$$(4x\cdot\cos(x)+2x+\cos(x)-1)\cdot\sin(\cos(x))\cdot\frac{\sin(x)}{x} \tag{2}$$



Obr. 17 Hledaná funkce a funkce (2)

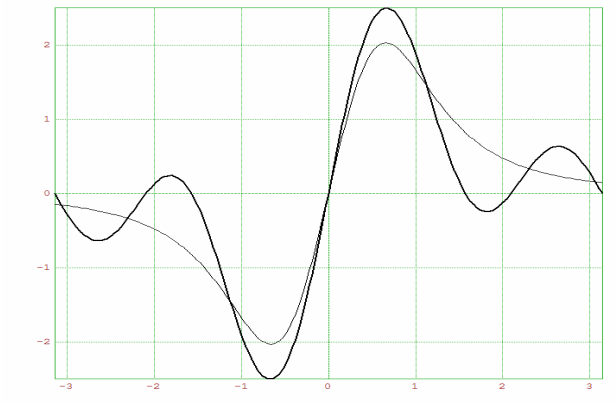
$$3 \cdot \cos(x) \cdot \cos(\sin(\sin(x))) \cdot (\sin(x) + \sin(2x)) \tag{3}$$



Obr. 18 Hledaná funkce a funkce (3)

Pokud bychom měli GFS={Plus,Minus,Krat,Deleno,x}, to znamená, že by neobsahovala funkce sin a cos, tak lze nalézt také uspokojivé řešení. Jedno z takových řešení je dáno vztahem (4). Srovnání je na obrázku (Obr. 19).

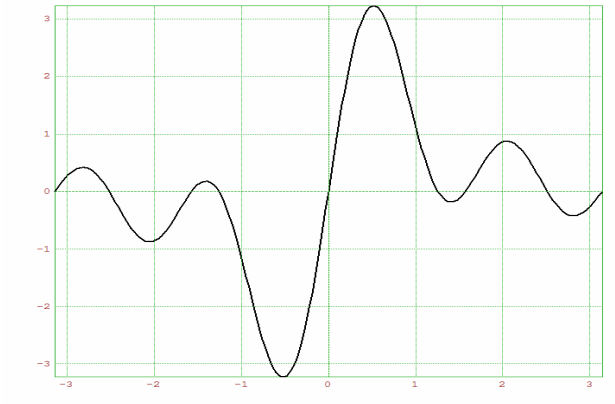
$$\frac{5x}{x^4 + x^2 + 1} \tag{4}$$



Obr. 19 Hledaná funkce a funkce (4)

4.1.4 Testování funkce $\sin(4x)+\sin(3x)+\sin(2x)+\sin(x)$

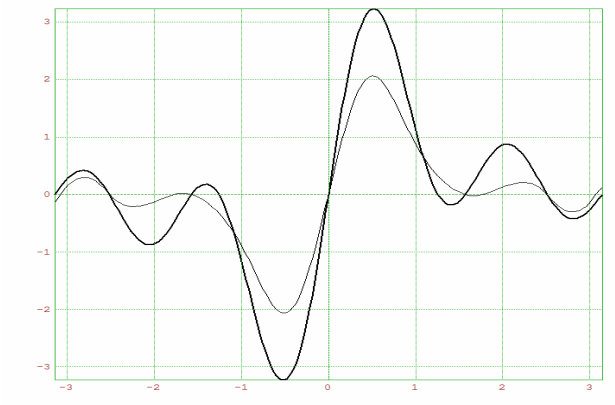
Graf funkce $\sin(4x)+\sin(3x)+\sin(2x)+\sin(x)$ je na obrázku ().



Obr. 20 Graf funkce
$\sin(4x)+\sin(3x)+\sin(2x)+\sin(x)$

Jedna z nalezených funkcí je dána vztahem (5). Porovnání funkcí je na obrázku ().

$$\frac{\cos(x)}{\frac{x}{\sin(\sin(x^2))}+\frac{\sin(1)}{\sin(x)-2x}} \tag{5}$$



Obr. 21 Hledaná funkce a funkce (5)

4.1.5 Boolean 3-symmetry problém

Boolean 3-symmetry problém je následující : Jestliže hodnoty, které vstupují do systému jsou symetrické, výstup je True (pravda). V opačném případě je výstup False (nepravda). Pravdivostní tabulka pro Boolean 3-symmetry problém je uvedena v tabulce (Tab. 10).

Tab. 10 Pravdivostní tabulka pro Boolean 3-symmetry problém

Vstup 1	Vstup 2	Vstup 3	Výstup
True	True	True	True
True	True	False	False
True	False	True	True
False	True	True	False
True	False	False	False
False	True	False	True
Flase	False	True	False
False	False	False	True

Účelová funkce je počítána použitím Hammingovy vzdálenosti mezi výstupem v pravdivostní tabulce a výstupem syntetizované formule (6). Maximální teoretická hodnota (nejhorší řešení ze všech) této účelové funkce je 8 pro 3-symetrický problém. Minimální hodnota je rovna 0. Cílem je najít nejlepší řešení, to znamená, aby CV byla rovna 0. Pro numerické počítání byly False a True nahrazeny čísly 0 a 1 [7].

$$f_{\cos t} = \sum_{i=1}^{2^n} \left| PT_i - F_i \right| \tag{6}$$

PT_i - i-tý výstup v pravdivostní tabulce

F_i - i-tý výstup syntetizované formule

Nastavení parametrů pro SOMA je uveden v tabulce (Tab. 11) a nastavení pro Diferenciální evoluci v tabulce (Tab. 12).

Tab. 11 Nastavení parametrů SOMA pro Boolean 3-symmetry problém

Parametr	Hodnota
Mass	3
Step	0,11
Prt	0,2
D	30
NP	300
Migrace	30

Tab. 12 Nastavení parametrů DE pro Boolean 3-symmetry problém

Parametr	Hodnota
CR	0,8
F	0,31
D	30
NP	300
Migrace	800

GFS je tvořena množinou logických funkcí And, Nand, Or, Nor a tří proměnných pro vstupy A,B,C. Pro tyto funkce je vytvořena knihovna `LogickeFunkce.dll`. Ve všech 20 testech byla hodnota CV = 0. Jedna z nalezených formulí je dána (7).

(((C or ((A and C)or A))and((C or (A or C))or C))nand((C and A)nor(A nand A)and(A and C))) (7)

ZÁVĚR

Cílem této diplomové práce bylo naprogramovat algoritmus Analytického programování v programovacím jazyce C#. V teoretické části práce byly popsány tři algoritmy pro symbolickou regresi. Vedle Analytického programování se jednalo o Genetické programování a Gramatickou evoluci. AP potřebuje ke svému běhu nějaký evoluční algoritmus. Výhodou AP je možnost použít téměř jakýkoliv evoluční algoritmus. V implementaci byly využity algoritmy SOMA a Diferenciální evoluce. Popis principů obou těchto algoritmů je také uveden v teoretické části práce.

Praktická část této práce spočívala v implementaci algoritmu AP a otestování výsledného software na vybraných příkladech. Aby mohl uživatel použít v GFS libovolnou funkci, hlavní aplikace neobsahuje žádné takové předem vytvořené funkce, ale využívá interface a metodu tzv. zásuvných modulů – pluginů. Uživatel si tak může vytvořit vlastní knihovnu funkcí, které potom použije při stanovení GFS. Do hlavní aplikace přitom vůbec nezasahuje. Hlavní aplikace komunikuje s vytvořenou knihovnou přes interface (rozhraní), které daná knihovna implementuje.

Vytvořený software byl otestován na několika vybraných funkcích. Ve většině případů se podařilo nalézt vhodné řešení. V případě funkcí se 3 sin a 4 sin se podařilo nalézt pouze přibližné řešení. I tato řešení, jak je vidět z přiložených grafů, byla velmi blízko požadovaným funkcím a proto je lze označit za uspokojivá. Poslední test se týkal „Boolean 3-symmetry“ problému. Pro tento případ byla vytvořena knihovna LogickeOperace.dll, která obsahuje potřebné logické operace. Při daném nastavení obou evolučních algoritmů bylo dosaženo ve všech případech úspěšného řešení.

CONCLUSION

The goal of this diploma thesis was to program an Analytic programming algorithm in the C# programming language. In the theoretical part of the work, three algorithms for symbolic regression were described. Next to the Analytic programming algorithm, Genetic programming and Grammatical evolution were also mentioned. Analytic programming needs for its run the evolution algorithm. An advantage of AP is the ability to use almost any evolution algorithm. SOMA and Differential evolution were used here during the implementation. The description of the principles of both of these algorithms is also referred in the theoretical part of this work.

The practical part is based on the implementation of AP algorithm and the testing of the resulting software on the selected examples. In order to enable the user to use any function in GFS, the main application does not contain any such, in advance prepared, functions, but uses interface and plugins architecture. The user can then program his own library of functions to use them for GFS. In such case, there is no need to interfere the main application and it communicates with the created library through an interface which the library implements.

Created software was tested with several selected functions. In most cases, the right solution has been found. In case of functions with 3 sin and 4 sin, I have managed to find approximate solution only. Nevertheless, even these solutions, as we can see from the accompanying graphs, were very close to the desired functions and therefore we can mark them as satisfactory. Last test was related to Boolean 3-symmetry problem. For this case, LogickeOperace.dll has been created; the library contains the necessary logical operations. Under the given settings for both evolution algorithms, successful solution in all cases has been accomplished.

SEZNAM POUŽITÉ LITERATURY

[1] ZELINKA, Ivan. *Umělá inteligence v problémech globální optimalizace.* Praha : BEN, 2002. 189 s. ISBN 80-7300-069-5.

[2] OPLATKOVÁ, Zuzana. *Analytic programming*. Zlín : UTB-FT, 2003. 73 s. Diplomová práce.

[3] KOZA J. R. 1998, *Genetic Programming II*, MIT Press.

[4] KOZA, J.R. *Genetic Programming : A Paradigm for Genetically Breedenig Populations of Computer Programs to Solve Problems*. Stanford University : Computer Science Department technical report, 1990. 131 s. ISBN STAN-CS-90-1314.

[5] O'NEILL, M., RYAN, C. *Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language* : Kluwer Academic Publisher, 2003.

[6] ZELINKA I., OPLATKOVÁ Z. 2003, *Analytic programming – Comparative Study*. CIRAS'03, The second International Conference on Computational Intelligence, Robotics, and Autonomous Systems, Singapore, 2003, ISSN 0219-6131.

[7] ZELINKA I., OPLATKOVÁ Z., NOLLE L. *Analytic Programming - Symbolic Regression by Means of Arbitrary Evolutionary Algorithms* In: Special Issue on Inteligent Systems, International Journal of Simulation, Systems, Science and Technology, Volume 6, Issue 9, August 2005, pp 44 - 56, ISSN 1473-8031.

[8] ZELINKA I., OPLATKOVÁ Z., *Boolean Parity Function Synthesis by Means of Arbitrarry Evolutionary Algorithms - Comparative Study*, In: 8th World Multicon-ference on Systemics, Cybernetics and Informatics (SCI 2004), Orlando, USA, in July 18-21, 2004.

[9] ZELINKA I., *Analytic programming by Means of Soma Algorithm. Mendel '02*, In: Proc. 8th International Conference on Soft Computing Mendel'02, Brno, Czech Republic, 2002, 93-101., ISBN 80-214-2135-5.

[10] VAŘACHA, Pavel. *Syntéza neuronových sítí metodou symbolické regrese* . [s.l.], 2006. 83 s. UTB Zlín. Diplomová práce.

[11] NAGEL, Ch., et al. *Professional C# 2005 with .NET 3.0* [s.l.]: Wrox Press, 2007. 1798 s. ISBN 9780470124727.

[12] KAČMÁŘ, Dalibor. *Programujeme .NET aplikace : ve Visual Studiu .NET*. Praha: Computer Press, 2001. 330s. ISBN 80-7226-569-5.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AP	Analytické programování
BNF	Backus-Naurova forma
CF	cost function – účelová funkce
CV	cost value – hodnota účelové funkce
CR	práh křížení
D	dimenze
DE	diferenciální evoluce
EA	evoluční algoritmus
GE	Gramatická evoluce
GFS	základní množina AP
GP	Genetické programování
F	mutační konstanta
LISP	programovací jazyk
NP	velikost populace
PRT	perturbace
SOMA	Samoorganizující se Migrační Algoritmus

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: KNIHOVNY

Zdrojový kód knihovny MojeOperace.dll

```
using System;
using System.Collections.Generic;
using System.Text;
using PlugInterf;

public class Plus : IMathDual
{
    public double operace(double x, double y)
    {
        return x + y;
    }
    public string znacka()
    {
        return "+";
    }
}
public class Minus : IMathDual
{
    public double operace(double x, double y)
    {
        return x - y;
    }
    public string znacka()
    {
        return "-";
    }
}
public class Krat : IMathDual
{
    public double operace(double x, double y)
    {
        return x * y;
    }
    public string znacka()
    {
        return "*";
    }
}

public class Deleno : IMathDual
```

```
{
    public double operace(double x, double y)
    {
        if (y != 0)
        {
            return x / y;
        }
        else
        {
            return 1.7E308;
        }
    }
    public string znacka()
    {
        return "/";
    }
}
public class Sin : IMathPrim
{
    public double operace(double x)
    {
        return Math.Sin(x);

    }
}
public class Cos : IMathPrim
{
    public double operace(double x)
    {
        return Math.Cos(x);

    }
}
public class Tan : IMathPrim
{
    public double operace(double x)
    {
        return Math.Tan(x);

    }
}

public class Abs : IMathPrim
```

```
{
    public double operace(double x)
    {
        return Math.Abs(x);

    }
}
public class Log10 : IMathPrim
{
    public double operace(double x)
    {
        return Math.Log10(x);

    }
}
public class Xna2 : IMathPrim
{
    public double operace(double x)
    {
        return x*x;

    }
}
public class Nasobek3 : IMathPrim
{
    public double operace(double x)
    {
        return x * 3;
    }
}

public class Soucet : IMath
{
    public double operace(double[] pole)
    {
        double vysledek = 0;
        for (int i = 0; i < pole.Length; i++)
        {
            vysledek += pole[i];
        }
        return vysledek;
    }
}
```

Zdrojový kód knihovny LogickeOperace.dll.

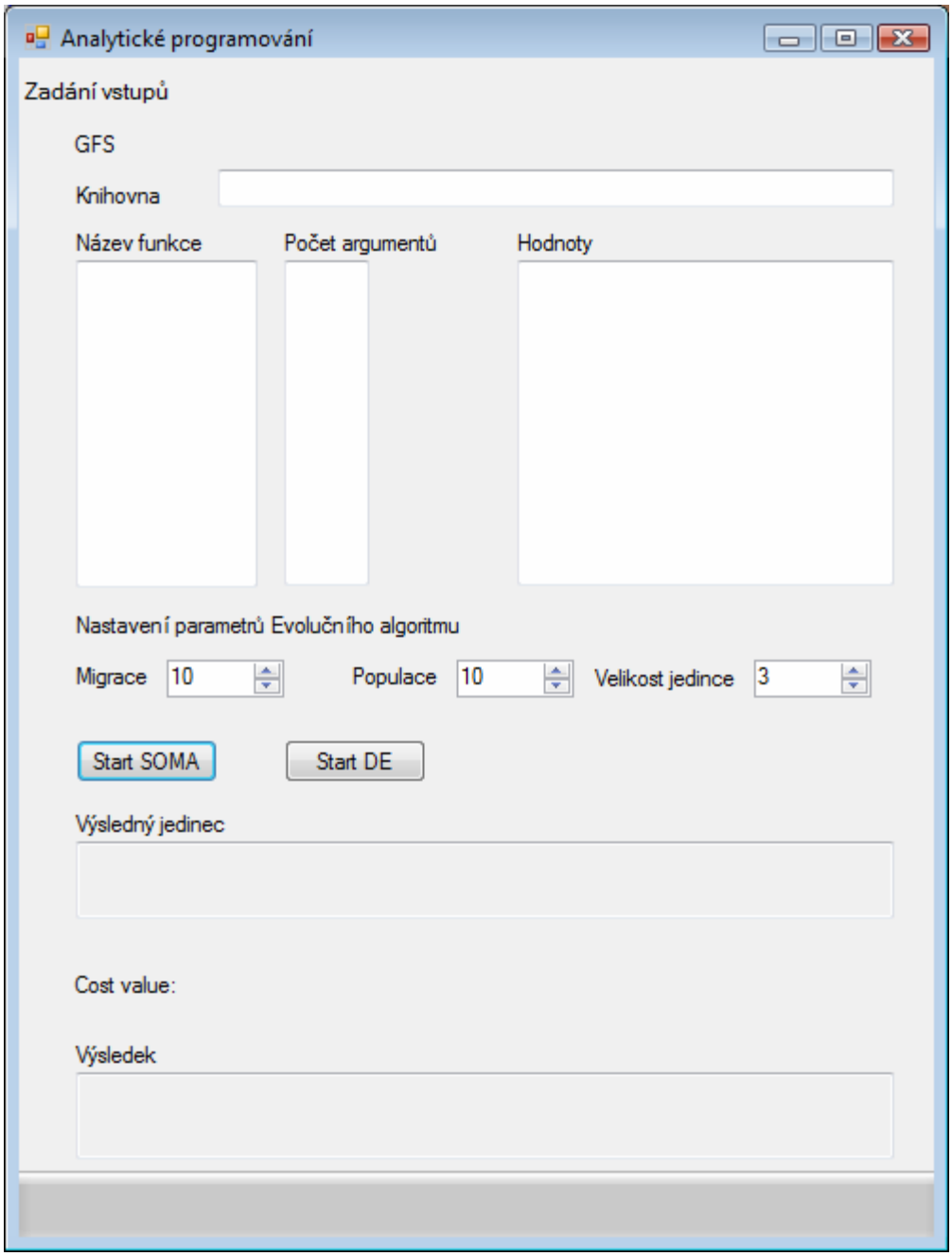
```
using System;
using System.Collections.Generic;
using System.Text;
using PlugInterf;

public class And:IMathDual
{
    public double operace(double x, double y)
    {
        return x * y;
    }
    public string znacka()
    {
        return "and";
    }
}
public class Or : IMathDual
{
    public double operace(double x, double y)
    {
        double v = x + y;
        if (v == 2)
        {
            v = 1;
        }
        return v ;
    }
    public string znacka()
    {
        return "or";
    }
}
```

```
public class Nand : IMathDual
{
    public double operace(double x, double y)
    {
```

```
        double v= x * y;
        if (v == 1) v = 0;
        else v = 1;
        return v;
    }
    public string znacka()
    {
        return "nand";
    }
}
public class Nor : IMathDual
{
    public double operace(double x, double y)
    {
        double v= x + y;
        if (v == 2)
        {
            v = 1;
        }
        if (v == 1) v = 0;
        else v = 1;
        return v;
    }
    public string znacka()
    {
        return "nor";
    }
}
```

PŘÍLOHA PII: POPIS PROGRAMU

Po spuštění aplikace ze zobrazí následující formulář:



Je nutné zadat následující vstupy:

Knihovna: název knihovny např. LogickeOperace.dll. Knihovna musí implementovat rozhranní, které je definováno v knihovně PlugInterf.dll a musí být nakopírována ve stejném adresáři jako spouštěná aplikace.

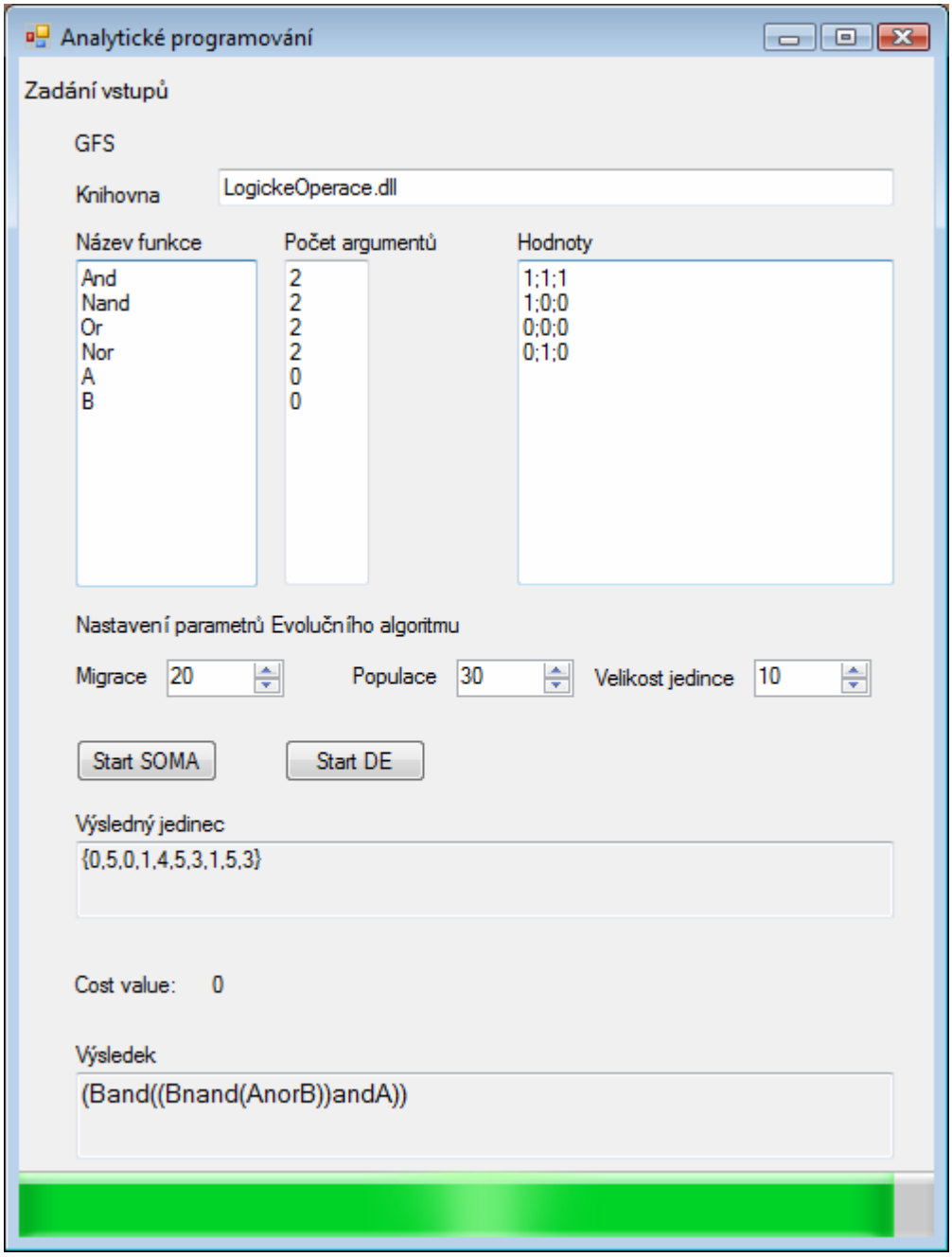
Název funkce: názvy jednotlivých funkcí a terminálů, které mají patřit do GFS. V případě funkcí se jedná o názvy tříd z dané knihovny.

Počet argumentů: Každé funkci musíme přiřadit počet jejích argumentů. U terminálů je to 0.

Hodnoty: vyplňují se hodnoty proměnných. Pokud máme v GFS dvě proměnné např x a y tak na jednom řádku jsou tři hodnoty oddělené středníkem. Poslední hodnota je očekávaná hodnota pro bod se souřadnicemi x a y.

Nastavení evolučních algoritmů : nastavení počtu migrací, velikost populace a velikost jedince.

Tlačítka Start SOMA a Start DE spouští příslušný evoluční algoritmus. Po ukončení výpočtu je zobrazen výsledek.


Analytické programování
Zadání vstupů
GFS
Knihovna
LogickeOperace.dll
Název funkce
Počet argumentů
Hodnoty
And
Nand
Or
Nor
A
B
2
2
2
2
0
0
1;1;1
1;0;0
0;0;0
0;1;0
Nastavení parametrů Evolučního algoritmu
Migrace
20
Populace
30
Velikost jedince
10
Start SOMA
Start DE
Výsledný jedinec
{0,5,0,1,4,5,3,1,5,3}
Cost value:
0
Výsledek
(Band((Bnand(AnorB))andA))