Hindawi Publishing Corporation
Modelling and Simulation in Engineering
Volume 2009, Article ID 845080, 12 pages
doi:10.1155/2009/845080

Research Article

Investigation on Evolutionary Synthesis of Movement Commands

Zuzana Oplatková and Ivan Zelinka

Faculty of Applied Informatics, Tomas Bata University in Zlín, Nad Stranemi 4511, 762 72 Zlín, Czech Republic

Correspondence should be addressed to Zuzana Oplatková, oplatkova@fai.utb.cz

Received 27 February 2008; Revised 24 November 2008; Accepted 3 February 2009

Recommended by Gaby Neumann

This paper deals with usage of an alternative tool for symbolic regression—analytic programming which is able to solve various problems from the symbolic domain, as well as genetic programming and grammatical evolution. This paper describes a setting of an optimal trajectory for a robot (originally designed as an artificial ant on Santa Fe trail) solved by means of analytic programming. Firstly, main principles of analytic programming are described and explained. The second part shows how analytic programming was used for the application of finding a suitable trajectory step by step. Because analytic programming needs evolutionary algorithms for its run, three evolutionary algorithms were used—self-organizing migrating algorithm, differential evolution, and simulated annealing—to show that anyone can be used. The total number of simulations was 150 and results show that the first two used algorithms were more successful than not so robust simulated annealing.

1. Introduction

The term "symbolic regression" represents a process during which measured data is fitted and a suitable mathematical formula is obtained in an analytical way. This process is well known for mathematicians. It is used when a mathematical model of unknown data is needed. For long time, symbolic regression was a domain of humans but in the last few decades, computers have gone to foreground of interest in this field. Firstly, the idea of symbolic regression done by means of computer was proposed by Koza in genetic programming (GP) [1–3]. The other two approaches are grammatical evolution (GE) developed by Ryan et al. [4–6] and here described analytic programming (AP) designed in [7–9].

Genetic programming was the first tool for symbolic synthesis of the so-called programs done by means of computer instead of humans. The main idea comes from genetic algorithms (GAs) [10], which Koza uses in his GP. The ability to solve very difficult problems was proved many times, and hence, GP today can be applied, for example, to synthesize highly sophisticated electronic circuits, robot trajectory, biochemistry problems, and many others [2].

The other tool is GE which was developed in the last decade of 20th century by Conor Ryan. Gramatical evolution has one advantage compared to GP which is the ability to use arbitrary programming language not only LISP as in the case of the cannonical version of GP. In contrast to other evolutionary algorithms, GE was used with a few search strategies with a binary representation of the populations [5], as well as with other algorithms like those in [11, 12]. Other 2 interesting investigations using symbolic regression were carried out by Johnson [13] working on artificial immune systems and probabilistic incremental program evolution (PIPE), the work in [14] generates functional programs from an adaptive probability distribution over all possible programs.

This contribution demonstrates the use of a method which is independent of computer platform, programming language, and can use any evolutionary algorithm (as demonstrated in [7–9]) to find an optimal solution of the required task.

2. Analytic Programming

2.1. Description. Basic principles of the AP were developed in 2001. Until that time, mainly GP and GE existed. GP uses genetic algorithms while AP can be used with any evolutionary algorithm, independently of individual

representation. To avoid any confusion based on the use of names according to the used algorithm, the name Analytic programming was chosen, because AP stands for synthesis of analytical solution by means of evolutionary algorithms [7–9].

AP was inspired, in general, by numerical methods in Hilbert spaces and by GP. Principles of AP [9] are somewhere between these two philosophies. From GP, an idea of evolutionary creation of symbolic solutions is taken into AP while from Hilbert spaces, an idea of synthesis of more complicated functions from elementary functions is adopted into AP. Analytic programming as well as GP is based on the set of functions, operators, and so-called terminals, which are usually constants or independent variables like

(i) functions: sin, tan, And, Or, and so forth,

(ii) operators: +, −, ∗, /, dt, and so forth,

(iii) terminals: 2.73, 3.14, t, and so forth.

All these "mathematical" objects create a set which AP tries to synthesize the appropriate solution from. The set of mathematical objects are functions, operators, and so-called terminals (usually constants or independent variables). All these objects are mixed together as shown in Figure 1and consist of functions with different number of arguments. Because of the variability of the content of this set, it is called for article purposes general functional set (GFS). The structure of GFS is nested, that is, it is created by subsets of functions according to the number of their arguments. The content of GFS is dependent only on the user. Various functions and terminals can be mixed together. For example, GFS_{all} is a set of all functions, operators, and terminals, GFS_{3arg} is a subset containing functions with only three arguments, GFS_{0arg} represents only terminals, and so forth.

This nested structure is necessary that the main principle of AP can work without any difficulties. The core of AP is based on discrete set handling, proposed in [15, 16] (see Figure 2). Discrete set handling (DSH) shows itself as a universal interface between EA and symbolically solved problem. That is why AP can be used almost by any evolutionary algorithm.

Briefly said, DSH works with integer indexes which represent numerical or nonnumerical expressions (operators, functions, etc.) in a discrete set. This index then serves like a pointer into a discrete set. Based on that, appropriate objects are chosen for cost function evaluation [16]. During an evolutionary process, only indexes are used for all evolutionary operations. Objects from the discrete set are used (by means of integer index) only in cost function, whereas according to integer index, a symbolic structure is synthesized and consequently evaluated.

2.2. Mapping Method in AP. The nested structure presence in GFS is vitally important for AP. It is used to avoid synthesis of pathological programs, that is, programs containing functions without arguments, and so forth. Performance of AP is, of course, improved if functions of GFS are expertly chosen based on experiencies with solved problem.

The important part of the AP is a sequence of mathematical operations which are used for program synthesis. These operations are used to transform an individual of a population into a suitable program. Mathematically said, it is mapping from an individual domain into a program domain. This mapping consists of two main parts. The first part is called discrete set handling (DSH) and the second one is security procedures which do not allow synthesizing pathological programs.

Discrete set handling proposed in [15, 16] is used to create an integer index, which is used in the evolutionary process like an alternative individual handled in EA by method of integer handling. The method of DSH, when used, allows handling arbitrary objects including nonnumeric objects like linguistic terms (hot, cold, dark, etc.), logic terms (true, false), or other user defined functions. In the AP, DSH is used to map an individual into GFS and together with security procedures (SP) creates the aforementioned mapping which transforms arbitrary individual into a program. Individuals in the population consist of integer parameters, that is, an individual is an integer index pointing into GFS.

Analytic programming is basically a series of function mapping. Figure 3 demonstrated an artificial example of how a final function is created from an integer individual. Number 1 in the position of the first parameter of integer index means that the operator "+" from GFS_{all} is used. Because the operator "+" has to have at least two arguments, next two index pointers 6 (sin from GFS_{all}) and 7 (cos from GFS_{all}) are dedicated to this operator as its arguments. Both functions, sin and cos, are one-argument functions so the next unused pointers 8 (tan from GFS_{all}) and 9 (t from GFS_{all}) are dedicated to sin and cos function. Because as an argument of cos variable t is used, this part of resulting function is closed (t is zero-argument) in its AP development. One-argument function tan remains, and because there is one unused pointer 9 tan is mapped on "t" which is on the 9th position in GFS.

To avoid synthesis of pathological functions a few security "tricks" are used in AP. The first one is that GFS consists of subsets containing functions with the same number of arguments. Existence of this nested structure is used in the special security subroutine which is measuring how far the end of individual is, and according to it, objects from different subsets are selected to avoid pathological function synthesis. Precisely, if more arguments are desired than possible (the end of the individual is near), function will be replaced by other function with the same index pointer from subset with lower number of arguments. For example, it may happen that the last argument for one argument function will not be a terminal (zero-argument function). If pointer is bigger than length of subset, that is, the pointer is 5 and is used GFS_{0arg}, then the element is selected according to $element = pointer\_value \bmod number\_of\_elements\_in\_GFS_{0arg}$. In this example, case-selected element would be variable t (see GFS_{0arg} in Figure 1).

GFS needs to be constructed not only from clear mathematical functions as demonstrated but also from other user-defined functions, which can be used, for example, logical

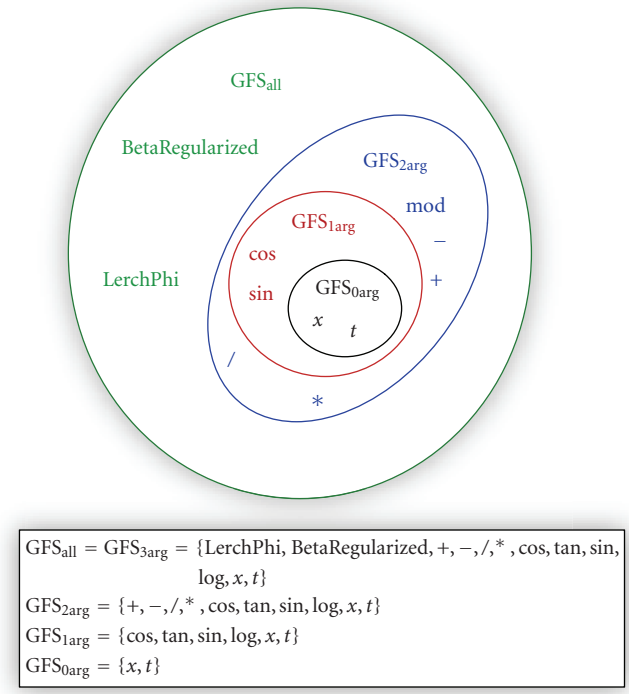

Figure 1: General function set (GFS).

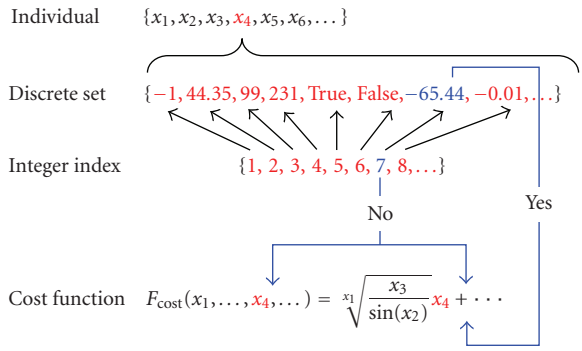

Figure 2: Discrete set handling.

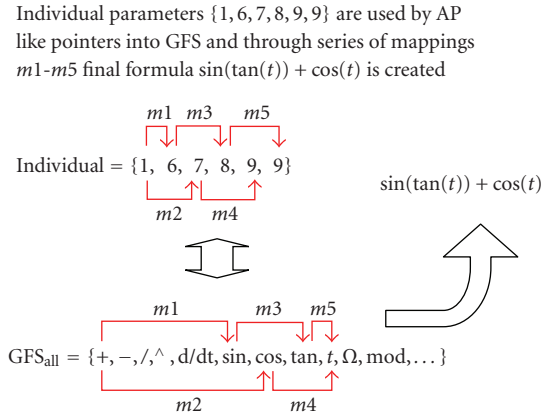

Figure 3: Main principles of AP.

functions, functions which represent elements of electrical circuits, or robot movement commands.

2.3. Versions of AP. Today, AP exists in three versions: AP$_{basic}$, AP$_{meta}$, and AP$_{nf}$. In all three versions, the same sets of functions, terminals, and so forth, as Koza use in GP [1–3] are necessary for the program synthesis. AP$_{basic}$ works as described earlier and the formulas do not contain any constants. The second version (AP$_{meta}$) is modified in the sense of constant estimation. For example, when Koza uses randomly generated constants in the so-called sextic problem [3], AP uses only one (K), which is inserted into the formula at various places by evolutionary processing. The function can look as follows:

$$\frac{x - K}{\pi^K}. \tag{1}$$

When the program is synthesized, then all "K" are indexed so that $K_1, K_2, \ldots, K_n$ are obtained from (2), and then all K_n are estimated by second evolutionary algorithm, and the result is in (3):

$$\frac{x - K_1}{\pi^{K_2}}, \tag{2}$$

$$\frac{x - 1.289}{\pi^{-112}}. \tag{3}$$

Because EA "works under" EA (i.e., EA$_{master}$▶ program▶ K indexing▶ EA$_{slave}$▶ estimation of K_n), this version is called AP with metaevolution—AP$_{meta}$. As this

version was quite time-consuming, another modification of AP_{meta} was done extending the second version by estimation of K. It is done by suitable methods of nonlinear fitting (AP_{nf}). This method has shown the most promising performance when unknown constants are present.

2.4. Security Procedures. Security procedures (SPs) are in the AP as well as in GP, used to avoid various critical situations. In the case that AP security procedures were not developed for AP purposes after all, but they are mostly integrated parts of AP. However sometimes they have to be defined as a part of cost function, based on kind of situation (e.g., situation 2, 3, and 4, etc., see what follows). Critical situations are like

(1) pathological function (e.g., without arguments, self-looped),

(2) functions with imaginary or real part (if not expected),

(3) infinity in functions (e.g., dividing by 0),

(4) "frozen" functions (e.g., extremely long time to get a cost value: hours).

Simply as an SP can be regarded here mapping from an integer individual to the program which is checked for how far the end of the individual is, and based on this information, a sequence of mapping is redirected into a subset with lower number of arguments. This satisfies that no pathological function will be generated. Another activities of SP are integrated part of cost function to satisfy items 2–4, and so forth.

2.5. Similarities and Differences. Because AP was partly inspired by GP, then between AP, GP, and GE are some differences as well as some logical similarities. A few of the most important ones are as follows.

I. Similarity

(i) Synthesized programs: AP as well as G0P and GE is able to do symbolic regression in general point of view. It means that output of AP is according to all important simulations [7–9] similar to programs from GP and GE (see http://www.fai.utb.cz/people/zelinka/ap).

(ii) Functional set: AP_{basic} operates in principle on the same set of terminals and functions as GP or GE.

II. Differences

(i) AP_{meta} or AP_{nf} use universal constant K (difference) which is indexed after program synthesis.

(ii) Individual coding: coding of an individual is different. Analytic programming uses an integer index instead of direct representation as in canonical GP. Grammatical evolution uses binary representation of an individual, which is consequently converted into integers for mapping into programs by means of BNF [4].

(iii) Individual mapping: AP uses discrete set handling, [13] while GP in its fundamental form uses direct representation in Lisp [1] and GE uses grammar-Backus-Naur form (BNF) [4].

(iv) Constant handling: GP uses a randomly generated subset of numbers, constants, GE utilises user-determined constants and AP uses only one constant K for AP_{meta} and AP_{nf}, which is estimated by other EA or by nonlinear fitting.

(v) Security procedures: to guarantee synthesis of non-pathological functions, procedures are used in AP which redirect the flow of mapping into subsets of a whole set of functions and terminals according to the distance to the end of the individual. If a pathological function is synthesized in GP, then synthesis is repeated. In the case of GE, when the end of an individual is reached, then mapping continues from the individual beginning, which is not the case of AP. It is designed so that a nonpathological program is synthesized before the end of the individual is reached (maximally when the end is reached).

2.6. Selected Solved Problems. During AP development and research simulations, a lot of various kinds of programs have been synthesized. In (2) a mathematical formula is shown to demonstrate complexity of synthesized formulas, which were randomly generated amongst 1000 formulas to check if the final structure is free of pathologies (i.e., if all functions have the right number of arguments, etc.). In this case, no attention was paid to mathematical reasonability of the following test programs based on clear mathematical functions. In what mentioned earlier, a different approach to the symbolic regression called analytic programming was described. Based on its results and structure, it can be stated that AP seems to be a universal candidate for symbolic regression by means of different search strategies. Problems on which AP was utilised were selected from test and theory problems domain as well as from real-life problems and are shown in following examples.

(i) Random synthesis of function from GFS, 1000 times repeated: the aim of this simulation was to check if pathological function can be generated by AP. In this simulation, randomly generated individuals were created and consequently transformed into programs and checked for their internal structure. No pathological program was identified [7].

(ii) $\sin(t)$ approximation was repeated 100 times. Here AP was used to synthesize the program function $\sin(x)$ fitting [7].

(iii) $||\cos(t)| + \sin(t)|$ approximation was repeated 100 times, the same as in the previous example. Main aim was again fitting of dataset generated by a given formula [7].

(iv) Solving of ordinary differential equations (ODE): $u''(t) = \cos(t)$, $u(0) = 1$, $u(\pi) = -1$, $u'(0) = 0$, $u'(\pi) = 0$, was repeated 100 times, in that case AP was looking for suitable function, which would solve this case of ODE [7].

(v) Solving of ODE: $((4 + x)u''(x))'' + 600u(x) = 5000(x - x2)$, $u(0) = 0$, $u(1) = 0$, $u''(0) = 0$, $u''(1) = 0$, was repeated 5 times (due to longer time of simulation in the Mathematica environment). Again as in the previous case, AP was used to synthesize a suitable function-solution of this kind of ODE. This ODE was used from and represents a civil-engineering problem in reality [7].

(vi) Boolean even and symmetry problems according to [1] for comparative reasons [9].

(vii) Sextic and Quintic problems [8].

(viii) Simple neural network synthesis by means of AP: a simple few layered NN synthesis was tested by AP [17].

Such elementary objects are usually simple mathematical operators (+, −, ∗, ...), simple functions (sin, cos, And, Nor, etc.), user-defined functions, and so forth. Output of symbolic regression is a more complex "object" (formula, function, command, etc.), solving a given problem like data fitting of so-called Sextic and Quintic problem described by (4), [2, 8], randomly synthesized function (5) [8], as well as Boolean problems of parity and symmetry solution (basically logical circuits synthesis) (6) [2, 9]. However, (4)–(6) mentioned here are just only a few samples of numerous repeated experiments done by AP and are used to demonstrate how complex structures can be produced by symbolic regression in general sense for different problems:

$$x\left(K_1 + \frac{(x^2 K_3)}{K_4(K_5 + K_6)}\right) * (-1 + K_2 + 2x(-x - K_7)), \tag{4}$$

$$\sqrt{t}\left(\frac{1}{\log(t)}\right)^{\sec^{-1}(1.28)} \log^{\sec^{-1}(1.28)}(\sinh(\sec(\cos(1)))), \tag{5}$$

```
Nor[(Nand[Nand[B||B, B&&A], B])&&C&&A&&B,
Nor[(!C&&B&&A||!A&&C&&B||!C&&!B&&!A)&&
(!C&&B&&A||!A&&C&&B||!C&&!B&&!A)||                (6)
A && (!C&&B&&A||!A&&C&&B||!C&&!B&&!A),
(C||!C&&B&&A||!A&&C&&B||!C&&!B&&!A)&&A]].
```

The rest of this article is an investigation on evolutionary synthesis of robot commands, which is well known in genetic programming as a Santa Fe trail for an artificial ant.

3. Problem Design

3.1. Santa Fe Description. The Santa Fe trail, demonstrated in Figure 4, was chosen from [18] to make a comparative study with the same problem which was solved by Koza in genetic programming [1].

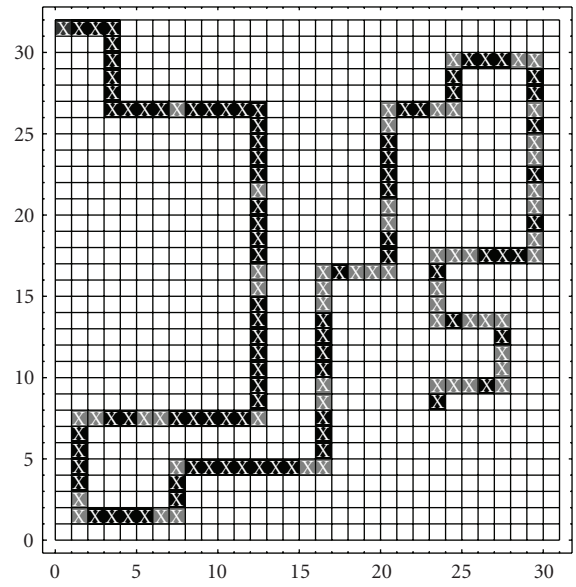

Figure 4: Santa Fe trail.

The aim of the task is that an artificial ant should go through defined trail and eat all food which is there. From a simple point of view, it can be looked at it as on robot movements on some trail. Robot trajectory is, of course, very complex task but the more complex behaviour can be added later in further simulations.

The Santa Fe trail is defined as a 32×31field where food is set out. In Figure 4, a black field is food for the ant. The gray one is basically the same as a white field but, for clarity, was used the gray color. The gray fields represent obstacles (fields without food on the road) for the ant. If there would not be these holes, the ant could go directly through the way. It would be enough to go and see before ant if there is food. If yes, ant would go straight and eat the bait. If not, it would turn around and see where food is, and the cycle would repeat till the ant would eat the last bait.

In the real world, robots have obstacles in their moving. Therefore, also in this case, such approach was chosen. The first problem which ant has to overcome is the simple hole (position (8,27) in Figure 4). Second one is the two holes in the line (positions (13,16) and (13,17)), or three holes ((17,15), (17,16), (17,17)). Next problem is the holes in the corners: one (position (13,8)) , two ((1,8), (2,8)), and three holes ((17,15), (17,16), (17,17)).

3.2. Set of Functions. The set of functions used for movements of the ant is as follows. As a set of variables GFS_{0arg}, that is, in the case of this article there are functions which provide movements of an ant, without any argument which could be add during the process of evolution.

The set consist of

(i) GFS_{0arg} = {Left,Right, Move},

where

GFS_{0arg}: a set of variables and terminals, zero argument functions GFS_{0arg},

Left: function for turning around in the anticlockwise direction,

Right: function for turning around in the clockwise direction,

Move: function for moving straight and if bait is in the field where the ant is moved, it is eaten.

This set of functions is not enough to make successfully a desired task. More functions are necessary, then a GFS_2 and a GFS_3 were set up:

(ii) GFS_2 = {IfFoodAhead, Prog2},

(iii) GFS_3 = {Prog3},

where the number in GFS means the arity of the functions inside, that is, the number of arguments which are needed to be evaluated correctly. Arguments are added to those functions during evolution process, as mentioned earlier in the description of AP.

IfFoodAhead is a decision function: the ant controls the field in front of it, and if there is food, the function in the field for truth argument is executed; otherwise, function in false position is performed.

Prog2 and *Prog3* are the same function in the principle. They do 2 or 3 functions in the same time. These two functions were originally defined also in Koza's approach but in AP, it is necessary because of the structure of generating the program.

3.3. Fitness Function. The aim of the ant is to eat all food on the way. There are 89 baits. This is so called raw fitness, and the value of cost function (7) is calculated as a difference between raw fitness and a number of baits eaten by an ant [1], which went through the grid according to just generated way:

$$\mathrm{CV} = 89 - \mathrm{Number\_of\_Food}, \tag{7}$$

where Number_of_Food is number of eaten baits by an ant according to synthesized way.

The aim is to find such formula whose cost value is equal to zero. To obtain an appropriate solution, two constraints should be set up into a cost function. One is a limitation concerned to the number of steps. It is not desired to the ant to go field by field in the grid. A requirement to the fastest and the most effective way is desired. Then a limit of steps was equal to 600. According to the original assignment, 400 steps should be sufficient, but as the work in [19], Koza's optimal solution was as in (8). However, as simple solution showed, 545 steps are necessary for an ant to eat all food in the Santa Fe trail.

IfFoodAhead[Move, Prog3[Left, Prog2[IfFoodAhead [Move, Right], Prog2[Right, Prog2[Left, Right]]], Prog2[IfFoodAhead[Move, Left], Move]]]. (8)

Table 1: Setting of SOMA.

Parameter	Value
PathLength	3
Step	0.22
PRT	0.21
PopSize	200
Migrations	50
MinDiv	-0.1
Individual length	50

Table 2: Setting of DE.

Parameter	Value
NP	200
F	0.8
CR	0.2
Generations	700
Individual length	50

Functionality of (8) can be described in follows. If bait is in front of the ant, it moves on the field and eats the food. If there is nothing, it does the following 3 commands. If food is in front of the ant it moves and eats the food, if not it turns twice right. Next Prog2(Left, Right) is not necessary there, this is the reason why all program takes 545 steps instead of 404 in the case of no Prog2(Left, Right). Then next control of food in front of ant is again, if yes ant moves and eats the food. If not it turns left to the original direction as it was at the beginning of the program. If the cycle is somewhere interrupt (e.g., in the case of truth in the first function IfFoodAhead), the cycle is repeated still from the beginning until all food is not eaten or constrained steps are not reached.

The second constraint could be concerned to the length of the list of commands for an ant. The longer can cause the more steps to reach all food is eaten. In this preliminary study, this constraint was not set up, but in further studies, a penalization concerned this constraint will be surely used.

4. Used Evolutionary Algorithm

In this paper, self-organizing migrating algorithm (SOMA), differential evolution (DE), and simulated annealing (SA) were used as an evolutionary algorithm. For detailed information, see [15, 20, 21].

4.1. Differential Evolution (DE). Differential evolution is a population-based optimization method that works on real-number-coded individuals [20]. For each individual $\vec{x}_{i,G}$ in the current generation G, DE generates a new trial individual $\vec{x}'_{i,G}$ by adding the weighted difference between two randomly selected individuals $\vec{x}_{r1,G}$ and $\vec{x}_{r2,G}$ to a third randomly selected individual $\vec{x}_{r3,G}$. The resulting individual $\vec{x}'_{i,G}$ is crossed-over with the original individual $\vec{x}_{i,G}$. The fitness of the resulting individual, referred to as a perturbated vector $\vec{u}_{i,G+1}$, is then compared to the fitness of $\vec{x}_{i,G}$. If the fitness of

Table 3: Setting of SA.

Parameter	Value
T	10 000
$T_{\min}$	0.000 01
α	0.986
MaxIter	1 500
MaxIterTemp	93
Individual length	50

Table 4: Cost function evaluation for SOMA, DE, and SA.

	Cost function evaluation		
	SOMA	DE	SA
Minimum	3 396	4 030	2 697
Maximum	134 114	136 011	98 241
Average	61 966	66 620	50 142

Table 5: Number of commands.

	Number of leaves (commands)		
	SOMA	DE	SA
Minimum	11	11	15
Maximum	50	50	50
Average	32	32	26

Table 6: Number of steps.

	Number of steps		
	SOMA	DE	SA
Minimum	396	367	406
Maximum	606	604	605
Average	547	540	535

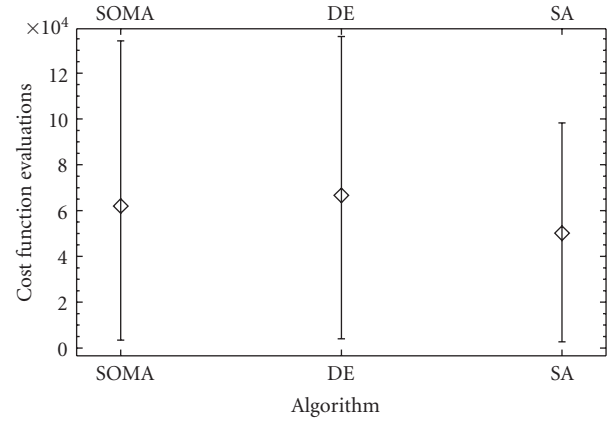

Figure 5: Graphical representation of minimal, maximal, and average values of cost function evaluation for SOMA, DE, and SA.

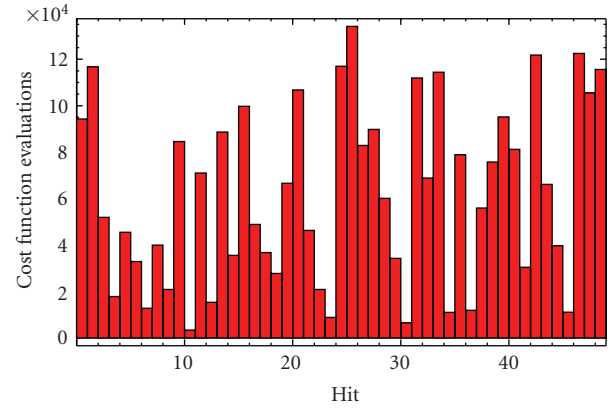

Figure 6: Histogram of SOMA algorithm.

$\vec{u}_{i,G+1}$ is greater than the fitness of $\vec{x}_{i,G}$, $\vec{x}_{i,G}$ is replaced with $\vec{u}_{i,G+1}$, otherwise $\vec{x}_{i,G}$ remains in the population as $\vec{x}_{i,G+1}$.

Differential evolution is robust, fast, and effective with global optimization ability. It does not require that the objective function is differentiable, and it works with noisy, epistatic, and time-dependent objective functions.

4.2. Self-Organizing Migrating Algorithm (SOMA). SOMA is a stochastic optimization algorithm that is modeled on the social behaviour of cooperating individuals [15] It was chosen because it has been proven that the algorithm has the ability to converge towards the global optimum [15]. SOMA works on a population of candidate solutions in loops called *migration loops*. The population is initialized randomly distributed over the search space at the beginning of the search. In each loop, the population is evaluated and the solution with the highest fitness becomes the leader L. Apart from the leader, in one migration loop, all individuals will traverse the input space in the direction of the leader. Mutation, the random perturbation of individuals, is an important operation for evolutionary strategies (ESs). It ensures the diversity among the individuals, and it also provides the means to restore lost information in a population. Mutation is different in SOMA compared with other ES strategies. SOMA uses a parameter called PRT to achieve perturbation. This parameter has the same effect for SOMA as mutation has for GA.

The novelty of this approach is that the PRT Vector is created before an individual starts its journey over the search space. The PRT Vector defines the final movement of an active individual in search space.

The randomly generated binary perturbation vector controls the allowed dimensions for an individual. If an element of the perturbation vector is set to zero, then the individual is not allowed to change its position in the corresponding dimension.

An individual will travel a certain distance (called the PathLength) towards the leader in n steps of defined length. If the PathLength is chosen to be greater than one, then the individual will overshoot the leader. This path is perturbed randomly.

4.3. Simulated Annealing (SA). Simulated annealing is one of older algorithm compared to SOMA and DE. It was introduced by Kirkpatrick et al. for the first time [21]. An inspiration for developing this algorithm was annealing of

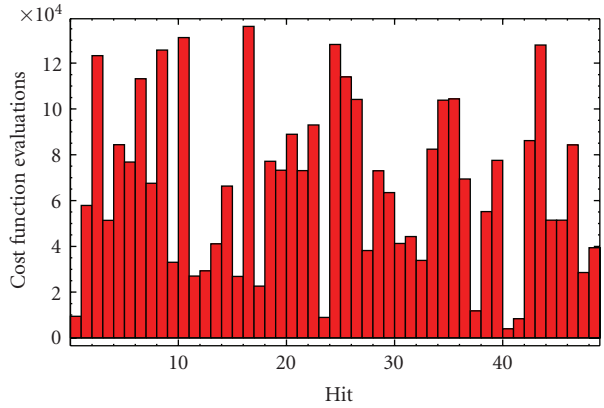

Figure 7: Histogram of DE algorithm.

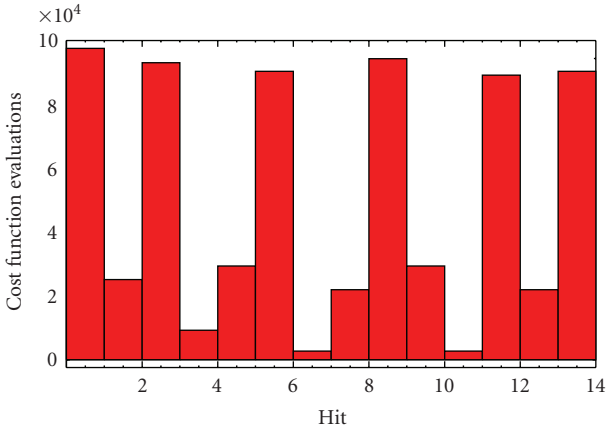

Figure 8: Histogram of SA algorithm.

metal. In the process, metal is heated up to temperature near the melting point and then it is cooled very slowly. The purpose is to eliminate unstable particles. In other words, particles are moved towards an optimum energy state. Metal is then in more uniform crystalline structure.

This approach was used in the case of simulated annealing including terms. It starts off from a randomly selected point. Then, a certain number of points (depends on user) are generated in the neighbourhood. The point with the best cost value is selected to be the middle of new neighbourhood (start point for a new loop). However, it is possible to accept also worse value of cost function. The acceptance is based on a probability which decreases with the number of iterations. In the case that the best cost value is in the start point, this one is chosen for the next loop. This approach is basic and some other improvements were done during research in this algorithm.

5. Experimental Results

The main idea is to show that SOMA, DE, and SA are able to solve such problems of symbolic regression—setting a trajectory—under analytic programming.

50 simulations were carried out for each algorithm (i.e., 150 simulations in total). SOMA and DE have almost all

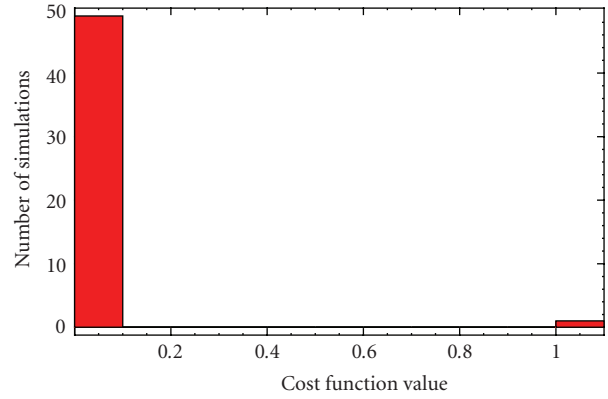

Figure 9: Histogram of SOMA algorithm: the number of cases in specific intervals of cost function values.

simulations with positive results; only one case in both algorithms did not reach the extreme. SA was not so successful, only 14 positive results. To show that AP is able to work with arbitrary evolutionary algorithms, we suppose to carry simulations out with genetic algorithms (GAs) and other algorithms, and also parallel computing is intended in this field. Data from all simulations were processed and vizualised in [20, 22].

In simulations made for the purposes of this article, following setting was used to run SOMA, DE, and SA according to Tables 1, 2, and 3, and explanation of each parameter symbol can be found in [15] (SOMA), [20] (DE), and [21] (SA).

Firstly, the results show values of cost function evaluations. This parameter shows good performance of analytic programming. As can be seen in Table 4, the lowest number of cost function evaluations equal 2697 for SA and 3396 for SOMA. DE was also not so far with its 4030 cost function evaluations.

Figure 5 shows the same as Table 4, but in a graphical way, where the diamond means the average value. As can be seen, SA had the lowest average value. However, this might be caused by only 14 cases which were included in the chart while SOMA and DE had 49 positive cases.

Second indicator depicts histogram of successful hits and the number of cost function evaluations for each hit (see Figures 6, 7, and 8). Negative results are not included.

Another creation of histograms can be made from the point of view of number of cases (axe y) which appeared in some interval of cost function values (axe x). This approach can be seen in Figures 9, 10, and 11. Here are all solutions, also bad ones which are represented by higher value than zero.

Next point, which we were interested in, was a number of commands for the ant and number of steps required to eat all baits (Tables 5 and 6). In Table 6, DE found a route which is overcome in the least number of steps can be seen. Sorted lists of pairs, commands and steps, are seen for all 3 algorithms in Table 7. As it is shown, it can be stated that the smallest number of commands does not have to cause

Table 7: Sorted numbers of steps and commands for all algorithms.

SOMA				DE				SA			
Sorted by steps		Sorted by commands		Sorted by steps		Sorted by steps		Sorted by commands		Sorted by steps	
396	49	594	11	367	49	599	11	406	25	577	15
399	36	596	11	387	49	592	12	406	25	592	16
409	21	568	14	390	50	564	13	409	23	605	16
409	22	594	14	409	18	542	14	503	22	592	17
409	23	594	14	409	18	568	14	503	22	537	19
421	37	577	15	409	50	577	14	537	19	503	22
456	50	544	16	421	50	581	14	577	15	503	22
489	17	590	16	475	16	581	14	577	49	409	23
521	50	594	16	496	50	583	15	592	16	406	25
532	50	606	16	509	21	594	15	592	17	406	25
533	20	489	17	516	46	475	16	592	50	594	34
533	27	544	17	517	49	533	16	594	34	594	34
537	34	583	17	519	49	409	18	594	34	577	49
540	27	576	18	525	38	409	18	605	16	592	50
542	27	533	20	533	16	533	18				
544	16	550	20	533	18	568	18				
544	17	409	21	533	20	584	19				
548	30	589	21	533	32	604	19				
548	50	409	22	541	49	533	20				
550	20	409	23	542	14	550	20				
551	43	559	24	550	20	509	21				
551	50	584	24	551	50	581	22				
559	24	583	26	557	31	596	23				
562	50	533	27	562	29	562	29				
568	14	540	27	564	13	557	31				
572	34	542	27	568	14	533	32				
574	27	574	27	568	18	525	38				
576	18	548	30	572	50	599	42				
577	15	537	34	573	49	516	46				
581	49	572	34	577	14	581	47				
581	50	399	36	581	14	367	49				
583	17	421	37	581	14	387	49				
583	26	551	43	581	22	517	49				
584	24	603	47	581	47	519	49				
589	21	396	49	583	15	541	49				
590	16	581	49	584	19	573	49				
592	50	596	49	588	50	589	49				
594	11	604	49	589	49	591	49				
594	14	606	49	591	49	595	49				
594	14	456	50	592	12	597	49				
594	16	521	50	594	15	601	49				
594	50	532	50	595	49	390	50				
596	11	548	50	595	50	409	50				
596	49	551	50	596	23	421	50				
601	50	562	50	597	49	496	50				
603	47	581	50	599	11	551	50				
604	49	592	50	599	42	572	50				
606	16	594	50	601	49	588	50				
606	49	601	50	604	19	595	50				

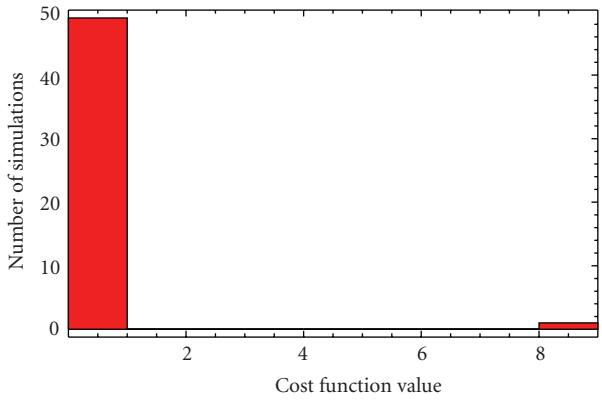

Figure 10: Histogram of DE algorithm: the number of cases in specific intervals of cost function values.

the smallest number of steps. Vice versa, the small number of steps does not mean the small set of commands.

Figure 12 depicts that the ant went through all fields; the white "X" shows fields which were attended by the ant. The notation (9) contains a set of rules for the ant how to go successfully through the trail. In (10), the whole description of the route can be seen where Ea, So, We, and No mean east, south, west, and north (which cardinal point the ant is turned into). The numbers in brackets are positions on the grid:

IfFoodAhead[Move, IfFoodAhead[Move, Prog2[Prog2 [Right, IfFoodAhead[Prog2[IfFoodAhead[IfFoodAhead [Move, Move], Move], Move], Prog3[IfFoodAhead[Move, IfFoodAhead[Prog3[Right, Right, Prog2[Left, Prog2[IfFoodAhead[Prog2[Prog2[Left, Move], Right], IfFoodAhead[Move, Left]], Prog2[IfFoodAhead[Move, Move], Prog2[IfFoodAhead[Move, Right], Right]]]]], Left]], Left, IfFoodAhead[Move, Right]]]], Move]]] (9)

{{32, 1},{32, 2},{32, 3},{32, 4},{So},{31, 4},{30, 4},{29, 4},{28, 4},{27, 4},{We},{So},{Ea},{27, 5},{27, 6},{27, 7},{So},{Ea},{No},{Ea},{27, 8},{27, 9},{27, 10},{27, 11},{27, 12},{27, 13},{So},{26, 13},{25, 13},{24, 13},{23, 13},{We},{So},{Ea},{So},{22, 13},{21, 13},{20, 13},{19, 13},{18, 13},{We},{So},{Ea},{So},{17, 13},{We},{So},{Ea},{So},{16, 13},{15, 13},{14, 13},{13, 13},{12, 13},{11, 13},{10, 13},{9, 13},{We},{So},{Ea},{So},{8, 13},{We},{8, 12},{8, 11},{8, 10},{8, 9},{8, 8},{No},{We},{So},{We},{8, 7},{No},{We},{So},{We},{8, 6},{8, 5},{8, 4},{No},{We},{So},{We},{8, 3},{No},{We},{So},{We},{8, 2},{No},{We},{So},{7, 2},{6, 2},{5, 2},{4, 2},{We},{So},{Ea},{So},{3, 2},{We},{So},{Ea},{So},{2, 2},{We},{So},{Ea},{2, 3},{2, 4},{2, 5},{2, 6},{So},{Ea},{No},{Ea},{2, 7},{So},{Ea},{No},{Ea},{2, 8},{So},{Ea},{No},{3, 8},{4, 8},{Ea},{No},{We},{No},{5, 8},{Ea},{5, 9},{5, 10},{5, 11},{5, 12},{5, 13},{5, 14},{5, 15},{So},{Ea},{No},{Ea},{5, 16},{So},{Ea},{No},{Ea},{5, 17},{So},{Ea},{No},{6, 17},{7, 17},{8, 17},{Ea},{No},{We},{No},{9, 17},{Ea},{No},{We},{No},{10, 17},{11, 17},{12, 17},{13, 17},{14, 17},{Ea},{No},{We},{No},{15, 17},{Ea},{No},{We},{No},{16, 17},{Ea},{No},{We},{No},{17, 17},{Ea},{17, 18},{17, 19},{17, 20},{So},{Ea},{No},{Ea},{17, 21},{So},{Ea},{No},{18, 21},{19, 21},{Ea},{No},{We},{No},{20, 21},{Ea},{No},{We},{No},{21, 21},{22, 21},{23, 21},{24, 21},{25, 21},{Ea},{No},{We},{No},{26, 21},{Ea},{No},{We},{No},{27, 21},{Ea},{27, 22},{27, 23},{27, 24},{So},{Ea},{No},{Ea},{27, 25},{So},{Ea},{No},{28, 25},{29, 25},{Ea},{No},{We},{No},{30, 25},{Ea},{30, 26},{30, 27},{30, 28},{So},{Ea},{No},{Ea},{30, 29},{So},{Ea},{No},{Ea},{30, 30},{So},{29, 30},{28, 30},{27, 30},{26, 30},{We},{So},{Ea},{So},{25, 30},{We},{So},{Ea},{So},{24, 30},{23, 30},{We},{So},{Ea},{So},{22, 30},{We},{So},{Ea},{So},{21, 30},{20, 30},{We},{So},{Ea},{So},{19, 30},{We},{So},{Ea},{So},{18, 30},{We},{18, 29},{18, 28},{18, 27},{No},{We},{So},{We},{18, 26},{No},{We},{So},{We},{18, 25},{No},{We},{So},{We},{18, 24},{No},{We},{So},{17, 24},{16, 24},{We},{So},{Ea},{So},{15, 24},{We},{So},{Ea},{So},{14, 24},{We},{So},{Ea},{14, 25},{14, 26},{So},{Ea},{No},{Ea},{14, 27},{So},{Ea},{No},{Ea},{14, 28},{So},{13, 28},{12, 28},{11, 28},{We},{So},{Ea},{So},{10, 28},{We},{10, 27},{10, 26},{10, 25},{No},{We},{So},{We},{10, 24},{No},{We},{So},{9, 24},{8, 24}}. (10)

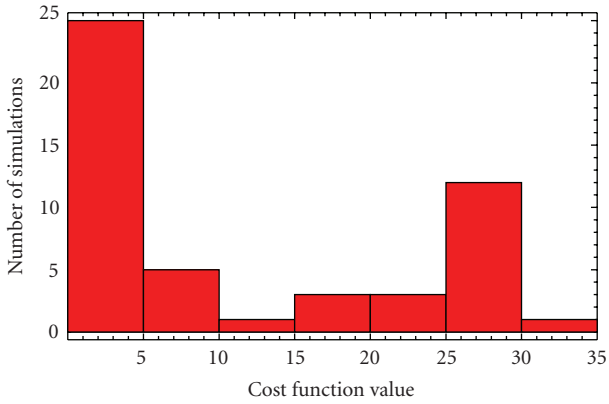

Figure 11: Histogram of SA algorithm: the number of cases in specific intervals of cost function values.

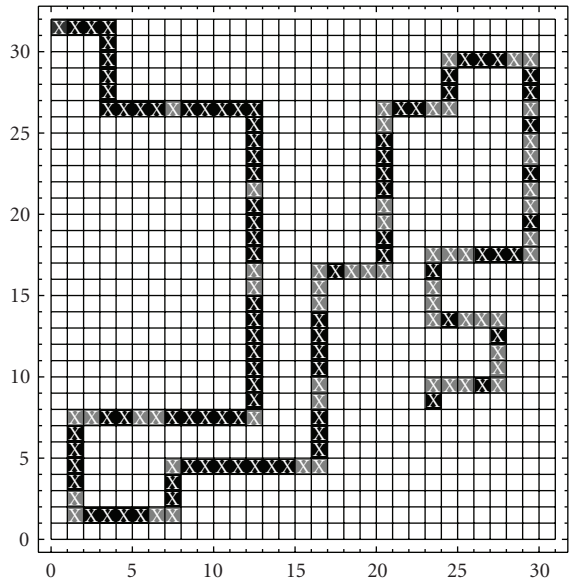

Figure 12: Santa Fe Trail overcome by ant found by DE.

6. Conclusions

This contribution deals with an alternative algorithm for symbolic regression. This study shows that this algorithm is suitable not only for mathematical regression but also for setting of optimal trajectory for artificial ant which can be replaced by robots in real world, in industry.

In comparison with standard GP, it can be stated on the basic aforementioned results that AP can solve this kind of problems in shorter times as cost function evaluations are counted.

The aim of this study was not to show that AP is better or worse than GP (or GE when compared), but that AP is also a powerful tool for symbolic regression with support of different evolutionary algorithms.

The main object of this paper was to show that symbolic regression done by AP is able to solve also cases where linguistic terms as, for example, commands for movement of artificial ant or robots in real world are. Here, simulations for 3 algorithms: SOMA, DE, and SA were carried out. As the figures showed, SOMA and DE were more successful in positive results than SA was. This proved that a good performance of AP depends on a choice of suitable robust and powerful evolutionary algorithms.

During simulations carried in this problem following results were reached:

(I) 50 simulations for each algorithm means 150 in total for all 3 algorithms.

(II) Positive results:

(i) 49 from 50 simulations for SOMA,

(ii) 49 from 50 for DE,

(iii) and 14 from 50 for SA,

which accomplished the required tasks thus analytic programming is able to solve such kind of problems in symbolic regression. This result also says that the basic version of simulated annealing used here is not so powerful tool as other two evolutionary algorithms are. It is supposed that the cost function is very complicated with quite a lot of local optima and, therefore, the simulated annealing was not so successful as SOMA or DE were.

(III) Solutions which fulfil conditions which were laid down by Koza [1], concerned to the number of steps, were found (2 by SOMA and 3 by DE). It means 5 solutions were successful under the 400 steps. Moreover, 17 (SOMA) + 20 (DE) + 6 (SA), in total 43 from 150 were successful under the 545 steps which was introduced by Koza [1, 22] as an optimal one.

Future research is key activity in this field. The following steps are to finished simulations with GA and other evolutionary algorithms and to try some other class of problems to show that analytic programming is powerful tool as genetic programming or grammatical evolution are.

Acknowledgments

This work was supported by Grant no. MSM 7088352101 of the Ministry of Education of the Czech Republic and by grants of the Grant Agency of the Czech Republic GACR 102/09/1680.

References

[1] J. R. Koza, *Genetic Programming*, MIT Press, Cambridge, Mass, USA, 1998.

[2] J. R. Koza, F. H. Bennet, D. Andre, and M. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Francisco, Calif, USA, 1999.

[3] http://www.genetic-programming.org.

[4] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2003.

[5] J. O'Sullivan and C. Ryan, "An investigation into the use of different search strategies with grammatical evolution," in *Proceedings of the 5th European Conference on Genetic Programming (EuroGP '02)*, pp. 268–277, Springer, Kinsale, Ireland, April 2002.

[6] http://www.grammatical-evolution.org.

[7] I. Zelinka, "Analytic programming by means of SOMA algorithm," in *Proceedings of the 8th International Conference on Soft Computing (Mendel '02)*, pp. 93–101, Brno, Czech Republic, June 2002.

[8] I. Zelinka and Z. Oplatkova, "Analytic programming—comparative study," in *Proceedings of the 2nd International Conference on Computational Intelligence, Robotics, and Autonomous Systems (CIRAS '03)*, Singapore, December 2003.

[9] I. Zelinka, Z. Oplatkova, and L. Nolle, "Boolean symmetry function synthesis by means of arbitrary evolutionary algorithms-comparative study," *International Journal of Simulation Systems, Science and Technology*, vol. 6, no. 9, pp. 44–56, 2005.

[10] L. Davis, *Handbook of Genetic Algorithms*, International Thomson Computer Press, Boston, Mass, USA, 1996.

[11] M. O'Neill and A. Brabazon, "Grammatical differential evolution," in *Proceedings of the International Conference on Artificial Intelligence (ICAI '06)*, pp. 231–236, CSEA Press, Las Vegas, Nev, USA, June 2006.

[12] M. O'Neill, F. Leahy, and A. Brabazon, "Grammatical swarm: a variable-length particle swarm algorithm," in *Swarm Intelligent Systems*, pp. 59–74, Springer, New York, NY, USA, 2006.

[13] C. G. Johnson, "Artificial immune system programming for symbolic regression," in *Proceedings of the 6th European Conference on Genetic Programming (EuroGP '03)*, C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds., vol. 2610 of *Lecture Notes in Computer Science*, pp. 345–353, Essex, UK, April 2003.

[14] R. Salustowicz and J. Schmidhuber, "Probabilistic incremental program evolution," *Evolutionary Computation*, vol. 5, no. 2, pp. 123–141, 1997.

[15] I. Zelinka, "SOMA-self organizing migrating algorithm," in *New Optimization Techniques in Engineering*, B. V. Babu and G. Onwubolu, Eds., Springer, New York, NY, USA, 2004.

[16] J. Lampinen and I. Zelinka, "Mechanical engineering design optimization by differential evolution," in *New Ideas in Optimization*, vol. 1, pp. 127–146, McGraw-Hill, Boston, Mass, USA, 1999.

[17] I. Zelinka, P. Varacha, and Z. Oplatkova, "Evolutionary synthesis of neural network," in *Proceedings of the 12th International Conference on Softcomputing (Mendel '06)*, pp. 25–31, Brno, Czech Republic, May-June 2006.

[18] Z. Oplatková, "Optimal trajectory of robots using symbolic regression," in *Proceedings of the 56th International Astronautical Congress*, Fukuoka, Japan, October 2005, paper no. IAC-05-C1.4.07.

[19] V. a kol. Mařík, *Artificial Intelligence IV*, Academia, Prague, Czech Republic, 2004.

[20] K. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*, Natural Computing Series, Springer, New York, NY, USA, 1st edition, 2005.

[21] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[22] Z. Oplatková and I. Zelinka, "Investigation on artificial ant using analytic programming," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pp. 949–950, Seattle, Wash, USA, July 2006.