Juraj SPALEK[*]
Michal GREGOR[**]

Adaptive Approaches to Parameter Control in Genetic Algorithms and Genetic Programming

Abstract

The paper concerns the application of Genetic Algorithms and Genetic Programming to complex tasks such as automated design of control systems, where the space of solutions is non-trivial and may contain discontinuities. Several adaptive mechanisms for control of the search algorithm's parameters are proposed, investigated and compared to each other. It is shown that the proposed mechanisms are useful in preventing the search from getting trapped in local extremes of the fitness landscape.

Introduction

Genetic Algorithms represent a well-known optimization method recognized in particular for its flexibility in representation of solutions. Genetic Programming applies the theory of Genetic Algorithms to evolving computer programs, usually represented by syntactic trees.
There is a multitude of research papers that aim to improve convergence and robustness of both algorithms. Some of these concentrate on parameter control, that is to say on setting and modifying various parameters of the search algorithm.
This paper proposes several adaptive mechanisms, which aim to decrease the probability that the search will become trapped in local maxima by various techniques. They are all based on detecting that the search has become trapped by observing how average fitness of the population changes in time.

Genetic Algorithms

Genetic algorithms represent one of the several computational techniques based on simulation of evolution, a process based on the principle of *natural selection*, that is, on the *survival of the fittest*. The genetic algorithm operates on a population of individuals. The individuals represent various solutions of a specific problem. The main principle of the algorithm is as shown in figure 1.

---

* Prof. Ing. Juraj Spalek, PhD. – Department of Control and Information Systems, Faculty of Electrical Engineering, University of Žilina, Univerzitná 1, 010 26 Žilina, Slovak Republic, juraj.spalek@fel.uniza.sk
** Bc. Michal Gregor – Department of Control and Information Systems, Faculty of Electrical Engineering, University of Žilina, Univerzitná 1, 010 26 Žilina, Slovak Republic, o.m.gregor@gmail.com

The first step is to generate the initial population – this typically involves generating a group of random individuals. The next step is to perform evaluation of those individuals, which enables the algorithm to compare the individuals to each other and, furthermore, to introduce the survival of the fittest: the individuals with the best scores (also known as *fitness* in the GA terminology) are the most likely[**] to participate in *reproduction*, that is, in forming the next generation. This is analogous to the natural selection process, in which the fitter individuals have greater chance to survive and reproduce.

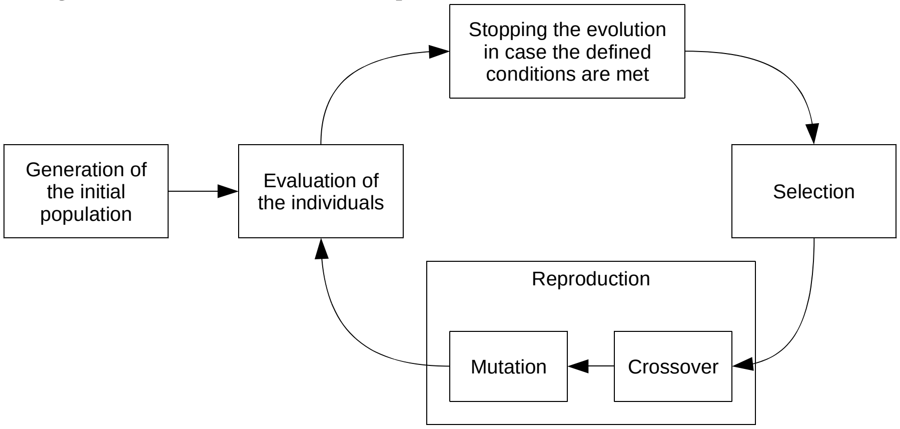
Stopping the evolution in case the defined conditions are met
Generation of the initial population
Evaluation of the individuals
Selection
Reproduction
Mutation
Crossover

Fig. 3. The general principle of genetic algorithms

Figure 1 also shows that the process of forming the next generation typically involves two main genetic operators – crossover and mutation. Mutation represents a random modification of the genetic code of a single individual.
In crossover, however, several (usually two) individuals exchange parts of their genome. Therefore, if we choose mostly the highly fit individuals for reproduction, crossover provides a mechanism which may produce an offspring that combines their good properties (and thus achieves greater fitness that any of the parents).
The process of evolution runs iteratively until certain conditions are met (like achieving a predefined level of (maximum or average) fitness, or reaching the maximum number of generations[††]).
The individual phases will not be covered in detail here, see [1], [2], or [3]. However, the next section will present some information concerning fitness scaling as this concept will be utilized in the following sections.

Fitness Scaling

There is a well known problem associated with the fitness-proportionate selection methods. As [3] says, when the evolution starts, the fitness variance in population is usually high and a small number of individuals are much fitter than the others. Those individuals are consequently

** However, we usually refrain from directly choosing the best *n* individuals as that tends to reduce diversity, which leads to premature convergence and to getting trapped in a local extreme.
†† The latter is usually monitored in every implementation so as to prevent an infinite loop in case the algorithm does not converge.

much more likely to be selected than any of the others and so their offspring quickly multiplies, which leads to premature convergence and non-optimal results.
On the other hand, later in the search, when all individuals are very similar and the fitness variance is therefore low, the evolution becomes extremely slow as there are virtually no fitness differences to explore.
To address these problems a fitness scaling function can be applied – that is, the original fitness function f will be wrapped into a scaling function f_s:

$$f_s : F \rightarrow F \cdot \tag{7}$$

The scaling function wraps the original fitness function and the selection algorithm uses the scaled values:

$$Scaled\ \ fitness = f_s\left(f\left(x\right)\right), \tag{8}$$

where $x \in I$ represents an individual.
There are several widely used types of fitness scaling functions – [4] lists 3 basic categories:

1. linear,
2. sigma truncation,
3. power law.

Linear Scaling

A fitness function with linear scaling then has the following definition [4]:

$$f_{linear}(x) = a + b.f(x), \tag{9}$$

where $f(x)$ is the raw fitness and a, b are user-defined constants – article [4] experiments with $a = max\{f(x)\}$ and $b = -min\{f(x)\}/N$, where N is the number of individuals. In [5] author presents a way to derive relationships for a, b, which provide linear scaling that preserves the average fitness.

Sigma Truncation Scaling

For a fitness function with sigma truncation scaling, source [6] provides the following definition:

$$f_{sigma}(x) = 1 + \frac{f(x) - \mu_f}{\sigma_f}, \tag{10}$$

where μ_f and σ_f are the mean fitness and the standard deviation – respectively – of fitness for the current generation.

Power Law Scaling

Source [5] provides the following definition of fitness function scaled using the power law scaling:

$$f_{power}(x) = f(x)^k \, , \tag{11}$$

where k is a problem-dependent exponent that may require to be changed during the run. [5] also states that a value of $k = 1.005$ has been successfully used in machine-vision applications.

Boltzmann Scaling

There are also several special scaling methods, such as the Boltzmann scaling [6], definition of which is as follows:

$$f_{Boltzmann}(x) = \frac{\exp(f(x)/T)}{mean[\exp(f(x)/T)]} , \tag{12}$$

where T represents a *temperature* parameter, which gradually reduces over time (with an increasing number of generations).

Scaling the Fitness Function to Satisfy the Requirements

Certain selection methods also impose requirements on the range of the fitness function, the most obvious example being the fitness roulette selection, where fitness values must be greater than or equal to zero (see (11)). The most apparent way to achieve this is to use the following scaling, which could be considered a special case of linear scaling:

$$f_s(x) = \begin{cases} f(x) - min\{f(x)\} & min\{f(x)\} < 0 \\ f(x) & min\{f(x)\} \geq 0 \end{cases} \tag{13}$$

The minimum can be evaluated over the current generation, or over the current and n previous generations in which case the subtraction of the minimum is referred to as *fitness windowing* [7].

Adaptive Genetic Algorithms

In some applications based on the theory of genetic algorithms, the optimization task may be so difficult – with a complex space including a great number of local optima in which the search process can be get trapped – that additional techniques may be required to find the global optimum. Genetic programming presented in the next section does in a multitude of tasks serve as an especially good example of the problem, as it evolves computer programs and it is obvious that two very similar computer programs may produce drastically different results and thus the space of solutions is highly complex..

Among the approaches that aim to prevent getting trapped in a local optimum are the adaptive schemes that observe various parameters of the algorithm or the search process itself and using the observed values adapt some of the parameters. The approaches to parameter setting can basically be divided into the following categories [8], [9]:

4. static parameter control,
5. dynamic parameter control,
6. adaptive parameter control,
7. self-adaptive parameter control.

Static Parameter Control

The common feature of approaches falling into this category is that the setting they provide remains constant for the entire duration of the evolutionary process. There are many works analysing the problem of finding optimum settings for parameters like mutation probability and crossover probability. Some of these are listed in [8], e.g. the work of Mühlenbeinm which proposes the following formula for the mutation probability:

$$p_m = 1/L\,, \tag{14}$$

where L is the length of the bit string.

Dynamic Parameter Control

As stated in [9] dynamic parameter approaches typically prescribe a deterministically decreasing schedule over a number of generations and provides a formula for mutation probability derived by Fogarty:

$$p_m(t) = \frac{1}{240} + \frac{0.11375}{2^t}\,, \tag{15}$$

where t is the generation counter.
Papers [8] and [9] both refer to a more general expression derived by Hesser and Männer:

$$p_m(t) = \sqrt{\frac{\alpha}{\beta}} \times \frac{\exp\left(\frac{-\lambda t}{2}\right)}{\lambda\sqrt{L}}\,, \tag{16}$$

where α, β, γ are constants, λ is the population size and t is the generation counter and L is again the length of the bit string.

Adaptive Parameter Control

Adaptive parameter control techniques monitor the search process itself and provide feedback. Some examples can be found in [10], which starts with a simple expression for the mutation and crossover probabilities. Crossover probability is expressed as follows:

$$p_c = \frac{k_1}{f_{max} - \bar{f}}, \tag{17}$$

where k_1 is a constant and f_{max}, $\bar{f}$ are the current generation maximum and average fitness values respectively.
A similar formula is proposed for mutation probability:

$$p_m = \frac{k_2}{f_{max} - \bar{f}}, \tag{18}$$

where k_2 is a constant.
It is further concluded in [10] that these expression do not depend on the fitness value of any particular solution, which means that the crossover and mutation probabilities will be the same for both – individuals with low and high fitness values. Another version of these formulas is derived that reflects these concerns [10]:

$$p_c = \begin{cases} k_1 \dfrac{f_{max} - f'}{f_{max} - \bar{f}} & f' > \bar{f} \\ \\ k_3 & f' \leq \bar{f} \end{cases} \tag{19}$$

$$p_m = \begin{cases} k_2 \dfrac{f_{max} - f}{f_{max} - \bar{f}} & f > \bar{f} \\ \\ k_4 & f \leq \bar{f} \end{cases} \tag{20}$$

where f is the fitness value of the individual to be mutated, f' is the larger of the fitness values of the individuals to be crossed and k_3 and k_4 are constants. It is required that k_1 and k_2 be less than 1.0 in order to constrain p_c and p_m to the range of $\langle 0,1 \rangle$. The $p_c = k_3 \quad f' \leq \bar{f}$ and $p_m = k_4 \quad f \leq \bar{f}$ expressions are to prevent crossover and mutation probabilities from exceeding 1.0 for suboptimal solutions.
Authors of [10] also observe that p_c and p_m are zero for the solution with maximum fitness and that $p_c = k_1$ for $f' = \bar{f}$, while $p_m = k_2$ for $f = \bar{f}$. For further details and for information concerning setting the values of the constants refer to [10]. Some discussion concerning this approach is also provided in section 0.

Self-adaptive Parameter Control

When using the self-adaptive parameter control approach, parameters such as mutation rate and crossover probability of each individual are part of its genome and are evolved with it. As stated in [9], the idea behind this is that a good parameter value will provide an evolutionary advantage to the individual. For further reference see [8] or [9].

Genetic Programming

Genetic programming (GP) is a technique introduced by John Koza (see *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [11]). It utilizes the previously outlined concepts to evolve computer programs. The main idea of Genetic Programming revolves around the way in which the individuals are represented, that is to say around the syntactic trees (also known as parse trees). The problem will be analysed more specifically in the following sections.

Representation

It is obvious, that simple text-based representation of a programme is not especially suitable for genetic algorithms as using a naïve implementation of crossover and mutation over the text-based code would lead to syntactically incorrect programs.
The solution proposed by John Koza is to represent a program using a parse tree (see Fig. 2 and 3 for an instance), which is analogous to LISP S-expressions [1]. The syntactic tree is a graph with two types of nodes – non-terminals, which represent functions, and terminals, which represent variables and constants.
Figures 2 and 3 show examples of such trees with Fig. 2 displaying a tree that codes the expression $x.y + \ln x$ and Fig. 3 displaying a tree with more general mechanisms like conditional execution, assignment and return.

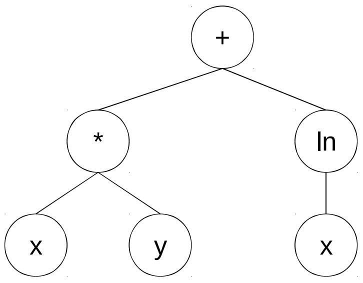

Fig. 4. A simple example of a syntactic tree

The program in Fig. 3 shows one of the possible ways to return values. The root node called PRG (the name is taken over from [1], where a PRG functor is used to express that several void-returning functors are called in a sequence) is a functor with an arbitrary number of inputs of type *void*, while the last input is of a pre-set type, which is identical to the return type of the program. That way after all processing is done by the void input subtrees, the result can be collected using the last input and returned.

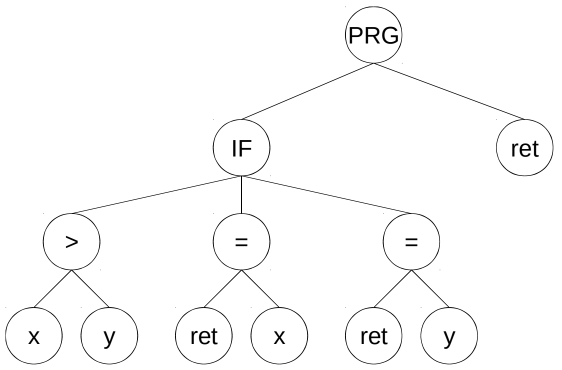

Fig. 5 A more complex parse tree

The program from figure 3 can be rewritten into the following C++ code (Listing 1):

Listing 1 Code expressed by Fig. 3

```
if(x > y) ret = x;
else ret = y;
return ret;
```

The representation proposed by Koza has one important property, known as the *closure property*, which requires that any valid tree generated from a set of terminals:

$$T = \{t_{1,}t_{2,}...,t_n\}, \tag{21}$$

and a set of non-terminals:

$$NT = \{t_{1,}t_{2,}...,t_m\}, \tag{22}$$

represents a valid program, which states that any non-terminal should be able to handle as an argument any data type and value returned from a terminal of non-terminal [12].
In contrast to this approach, several researches focus on the so-called *strongly typed genetic programming* [12], where nodes are allowed to have different incompatible return and argument types. In this case, type constraints have to be enforced, which introduces several fundamental differences. The most notable aspect is that when generating, crossing or mutating a tree care has to be taken to ensure that the return type of the node used as an input is compatible with the data type of the input itself.
The closure property can still be enforced in strongly typed genetic programming using dynamic typing. Non-terminals can be built so that they accept an argument of any type, but throw an exception if type id of the argument is not as expected.

The Artificial Ant Problem

The artificial ant problem described by John Koza in [11] is essentially a trail-following task. The actor – an artificial ant – is supposed to navigate in an environment following an irregular path consisting of pieces of food which it collects. The ant has very limited sensing capabilities – it only sees a single tile right in front of it. John Koza successfully solves the problem by applying Genetic Programming‡‡.
This constraint, although a reasonable one – with many line-following agents this is in fact the case – makes the task of navigating along a non-trivial path rather difficult. It seems that even a human is generally unable to navigate the ant correctly when only seeing a single tile in front of the actor although this has not been tested on a wide range of subjects.
Concerning the application of GP to the problem, Koza uses the following set of terminals [11]:

$$T = \{MOVE, \quad RIGHT, \quad LEFT\}, \tag{23}$$

and the following non-terminals:

$$F = \{IF\text{-}FOOD\text{-}AHEAD, \quad PROGN2, \quad PROGN3\}. \tag{24}$$

The meaning of most of these is straight-forward – MOVE moves the actor forward by a single step, RIGHT and LEFT turn the actor in the respective directions. IF-FOOD-AHEAD is a functor with two arguments – the first is the then part and is executed if there is a piece of food in front of the actor, while the other is the else part. PROGN2 and PROGN3 are functors with 2 and 3 arguments respectively. PROGN represents a sequence of steps to be executed unconditionally, that is, PROGN2 and PROGN3 both execute each of its sub-trees.

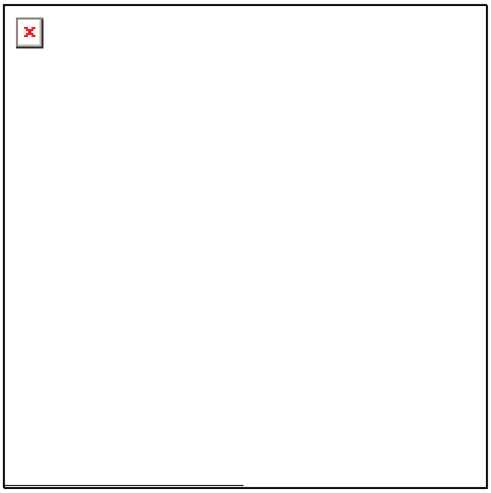

Fig. 6. The Santa Fe trail

‡‡ See [11] for detailed information about the solution.

The evaluation is based on simulation and the fitness is equal to the amount of food collected by the actor. It would normally be necessary to run several simulations for every individual to make sure that the solution works in general and not only on the single path on which it was tested. To avoid this Koza uses a trail known as the Santa Fe trail[§§] (Fig. 4), which is presumed to be sufficiently representative of the general trail following problem [11].

Mode of Execution and Operators Used

It is also necessary to mention the mode of execution used by Koza – the program generated by the evolutionary search is executed as fully as possible and then re-executed [11]. Both [11] and [1] limit the number of steps that a solution is allowed to perform to 400 so as to prevent running indefinitely for unfit individuals. The population size is set to 500 individuals and the maximum number of generations to 50 for both [11] and [1].

In our work we have set some additional requirements concerning the form of the solution – the evolved controller should, when executed, return the action that the ant is to execute next instead of calling functors that directly execute the action and wait for its completion. The set of terminals contains persistent variables and the controller has access to a pre-set number of its previous inputs and outputs.

Controllers based on such mode of execution seem to be much more difficult to evolve than those originally proposed by Koza. The search usually gets trapped in a local maximum from which it is often unable escape.

Let us provide the reader with some brief information concerning the terminals and non-terminals used in our work. The following components were utilized:

1. *VariableFunctor<NavAction>* – a terminal that acts as a variable of type NavAction (NavAction is an enumerated type representing the action that an actor can take like stay, forward, turn around, turn left, turn right).
2. *ConstFactory<NavAction>* – a factory that creates constant terminals of type NavAction.
3. *ConstFactory<TileType>* – a factory that creates constant terminals of type TileType (an enumerated type that represents various types of tiles in the map).
4. *ConstFunctor<void>* and *NumericConstFactory<bool>* – auxiliary terminals of type void and bool.
5. *IfAssign* – a non-terminal with 5 sub-nodes; the first is of type bool and expresses the condition. If the condition is true, value from sub-node 3 is assigned to variable from sub-node 2; if false value from sub-node 5 is assigned to variable from sub-node 4. Values and variables are of type NavAction.
6. *CompareFunctor<NavAction>* and *CompareFunctor<TileType>* – non-terminals that returns true if both of their inputs are equal and false if not.
7. Logic functors: *And, Or, Not.*
8. *PrgReturnFunctor(NavAction, N)* – a non-terminal used primarily as root functor of the tree – it has N sub-nodes returning void and one sub-node (the last one) returning NavAction. All sub-nodes are executed one by one and the return value of the last one is returned by the PrgReturnFunctor.

§§ It contains single gaps, double gaps, single, double and triple gaps at corners [11], etc.

Adaptive Value-switching of Mutation Rate

Motivation

Most of the existing parameter setting mechanisms, as presented in the previous section, either focus on setting GA-specific parameters such as length of the bit string (e.g. rule (8)), or are not adaptive (e.g. (8), (9) and (10)). The AGA adaptive mechanism described in [10] (formulas (13) and (14)) seems more fit to the task because it implements certain form of convergence detection based on comparison of the maximum and average fitness values. However this approach does little to solve the problem of getting trapped in a local optimum as the method does not discern between local and global optima.

Furthermore – as mentioned hereinbefore – equations (13) and (14) assign the best individual zero crossover and mutation probabilities, while assigning high probabilities to less fit individuals. The reasoning behind this is that the less fit individuals can safely be disrupted by high mutation rates and recombined by crossover (thus employing the solutions with sub-average fitness to search the space [10]), while the highly fit individuals should be preserved.

However, such approach has a very obvious downside which the authors do not seem to address – the highly fit individuals obviously contain the most excellent genetic material available and by disallowing mutation and crossover for these individuals the genetic code they carry becomes isolated and is not used to generate new solutions.

Description of the AVSMR Mechanism

The idea that the most fit solutions should survive crossover and mutation unmodified is valid, yet that feature can be enforced by using elitism[***]. Keeping that in mind we propose a different adaptation scheme – called AVSMR (Adaptive Value-switching of Mutation Rate) - in order to address the other issues. The main idea is that the mutation probability should be increased to a high value when the search has become trapped in an extreme so as to provide the search process with new genetic material some of which may previously have been unavailable. To determine whether the search has become trapped the adaptive mechanism observes the change of average fitness in time.

To describe the solution in more detail – the algorithm works with 2 values of mutation probability – the normal value and the high value. The algorithm switches from the normal value to the high value once the trigger criterion activates.

The trigger criterion itself is based on a measure that we will herein term a *delta sum*:

$$\Delta S_i = \alpha . \Delta S_{i-1} + \frac{\bar{f}_i - \bar{f}_{i-1}}{\bar{f}_i}, \tag{25}$$

where ΔS_i is the delta sum in generation i and $\bar{f}_i$ is the average fitness in generation *i* and α is the feedback coefficient (the experiments have been carried out for $\alpha = 0.4$).

If the delta sum is lower than a pre-set value for a predefined number of generations, that is to say the increase of average fitness in the last few generations is low, indicating that the search

*** The best individual is copied to the next generation unmodified.

has become trapped[†††] – the mutation probability is set to its high value so as to provide the search with new genetic material. As mentioned before, when used in conjunction with elitism it is guaranteed that the best solution is not destroyed by the high mutation probability.
The mutation probability is reset back to its normal value when at least one of the following conditions is true:

1. the average fitness increases enough to produce a sufficiently large delta sum;
2. the maximum fitness increases;
3. mutation has been set to its high value for at least n generations.

The *n*-generation limit is to ensure that the activation does not go on indefinitely (with the high mutation probability it is not very likely that the average fitness will increase enough to satisfy the first condition and maximum fitness may not increase as well).
It has been observed that average fitness typically decreases when the criterion activates because the search process is to a large extent disrupted by the high mutation probability. However after the *n*-generation limit forces the mutation rate back to its normal value, average fitness tends to increase rapidly, thus usually moving away from the local extreme.

Experimental Results

Several experiments have been carried out (the specific settings are attached in Appendix **Błąd! Nie można odnaleźć źródła odwołania.**) – Fig. 5 shows performance of the search algorithm with the AGA adaptive mechanism proposed in [10] with constants set according to recommendations. It also shows performance of the search algorithm without any adaptive mechanism and with the adaptive mechanism proposed in this paper. The maximum fitness value achieved is shown for each of the 5 runs displayed.

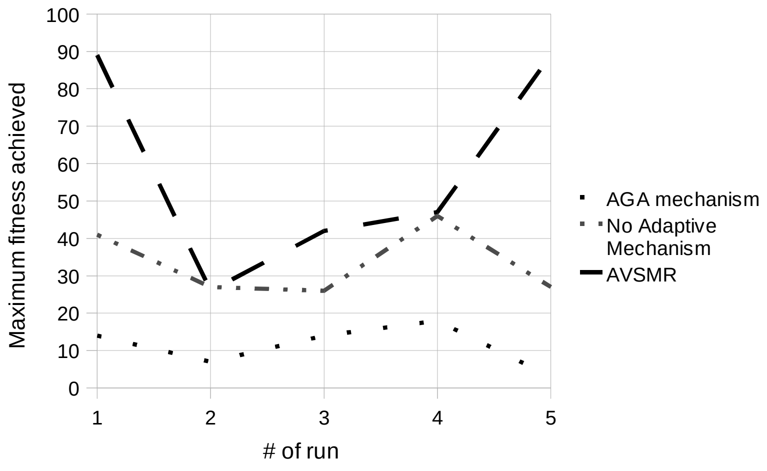

Fig. 7. Comparison of the AGA Adaptive Mechanism and AVSMR

††† This may also indicate convergence to the global maximum, it is, however, hardly possible to tell global and local maxima apart unless the algorithm is provided with additional problem-specific data.

As shown, search achieves suboptimal results when running with no adaptive mechanism. This can be ascribed to its inability to escape from local extremes. With no adaptive mechanism the search has not found the global optimum (fitness = 89) in any of the 5 runs.
As expected, the AGA mechanism has caused further deterioration and its results are even worse than those produced in the previous case.
The Value-switching adaptive mechanism proposed in this work improves the process of search – in 2 of the runs the global optimum is found, yet in certain cases not even the high mutation rate is guaranteed to help the search escape from the local maximum (runs 2, 3, 4).

Further Suggestions

It has been shown that the adaptive mechanism described in this work is able to effect considerable improvements and that it is able to some extent prevent getting trapped in local maxima. Further experiments should now be carried out in order to ascertain that the principle is valid for a wider range of tasks.
It has also become apparent that even with the high mutation rates it is not always guaranteed that the search will indeed escape from the local maximum. Value-switching, or piecewise continuous relationships for other parameters could perhaps help to alleviate the problem – this issue requires further investigation.

The Simple Flood Mechanism

Seeing that the AVSMR mechanism described in the previous section is helpful in controlling the search process by helping it to escape from local extremes, yet not completely reliable and not always effective. To address these issues, we have developed another adaptive scheme supposed to provide even greater level of introducing new genetic material into the process.

Simple Flood Mechanism

The principle is very straight-forward – once a trapping is detected – a relatively small part of the population is selected – these individuals survive. The rest of the population is destroyed and replaced by newly generated individuals. This method is superior to AVSMR in that a large part of the population is guaranteed to be replaced and the newly generated individuals are generated in the same way that the initial population was.
The trigger criterion has been modified for this task. The first requirement is that the criterion only activates for a single generation at a time as it would probably be useless and possibly even counterproductive to activate the flood mechanism for several successive generations.
The new trigger criterion is still based on the average fitness $\bar{f}_i$ (where i is the number of generation). The criterion stores average fitness $\bar{f}_i$ for N generations ($N-1$ previous generations and the current one; $N=7$ generations was used in the experiments). The mechanism cannot activate before the $\bar{f}_i$ for at least N generations has actually been collected. Once that is true, the mechanism activates if the following holds:

$$\sum_{i=j}^{j-(N-2)} \bar{f}_i - \bar{f}_{i-1} < \Theta, \qquad (26)$$

where j is the number of current generation and Θ is an activation threshold. It is also possible to interpret the threshold as a relative parameter in which case we can rewrite the equation as follows:

$$\frac{\sum_{i=j}^{j-(N-2)} \bar{f}_i - \bar{f}_{i-1}}{\bar{f}_j} < \Theta \cdot \qquad (27)$$

All experiments were carried out using (21).
It is also important to note that once the mechanism activates, the array storing the previous value of average fitness is cleared so it is guaranteed that the mechanism does not activate for the next N generations.
Although the approach seems straight-forward and similar in concept to AVSMR, experimental results point out an important issue. As obvious from Fig. 6, the results achieved by the Simple Flood Mechanism are significantly worse than those produced by the AVSMR – they are in fact worse than those produced by the system when using no adaptive mechanism.

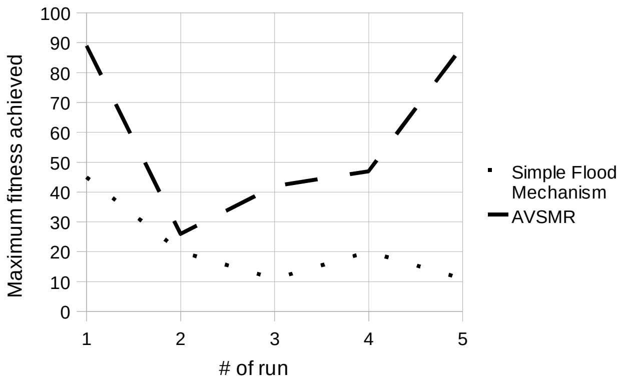

Fig. 8. Comparison of AVSMR and the Simple Flood Mechanism

The reason behind this is very simple – although we do introduce new genetic material into the process, the newly generated individuals will generally have very low fitness (usually 0, 3, or 4 at most). Therefore if we apply fitness-proportionate selection to these in the next generation, almost every newly generated individual will be discarded. The survivors on the other hand will now dominate the population. This is especially true later in the evolutionary process when fitness score of the best individual will tend to be vastly greater than that of any randomly generated individual. At this point the next generation will be formed almost exclusively by the best individual, which will almost in every case aggravate the problem of getting trapped in a local extreme instead of solving it.

Flood Mechanism with Low-pressure Scaling and the New-Blood Mechanism

There are several ways to alleviate the problem that the Simple Flood Mechanism faces. The objective is – in any case – to create such scheme in which the newly generated individuals mate with the survivors so as to make use of their potentially useful code.

This paper proposes two different ways to achieve this:

1. apply a fitness scaling function with low selection pressure to the GA for several generations following the flood – this mechanism will be referred to as *Flood Mechanism with Low-pressure Scaling (FMLPS)*;
2. once the mechanism activates create only such mating pairs in which at least one individual is newly generated – this mechanism will be referred to as the *New Blood Mechanism*.

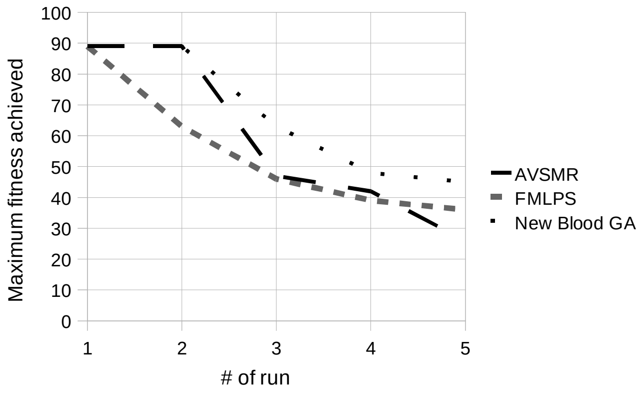

Fig. 9. Comparison of the AVSMR, FMLPS the New Blood Mechanism

The experimental results are shown in Fig. 7. FMLPS uses power scaling of 0.3 as the low-pressure scaling. To make the comparison easier, the values are now ordered by fitness rather than by the number of run. This shows that AVSMR is still superior to FMLPS (although FMLPS is – in contrast to the Simple Flood Mechanism – significantly better than vanilla GP). The New Blood GA on the other hand is definitely superior to AVSMR – although it still gets trapped in local extremes, the maximum fitness values achieved are greater than those achieved by the AVSMR.

The influence that some of the parameters such as the number of survivors, or the selection pressure applied by the low-pressure scaling have on the process of search should be subjected to a more systematic investigation. Combining the proposed adaptive mechanisms with some of the concepts introduced by the AGA mechanism could also prove useful – e.g. instead of decreasing the selection pressure using a scaling function the spread of the best individual's copies through the population immediately after the flood could be inhibited by techniques similar to those utilized in AGA.

Conclusion

It is well known that search processes based on genetic algorithms and genetic programming are prone to getting trapped in local maxima when exploring highly complex spaces. As shown in the paper, search process based on the standard genetic programming approach fails to find the global optimum when applied to the modified version of the artificial ant problem.

This paper investigates the problem and proposes several adaptive mechanism, which should help the search process to escape from local extremes. As shown, the results are considerably better than those of the standard Genetic Programming approach.

Although the results are better, even the proposed algorithms cannot always guarantee that the process will indeed escape from every local maximum it encounters. This stems mainly from the high order of stochasticity that the algorithm is subject to as well as from the size of the searched space.
Related techniques such as adaptive value-switching, or piecewise continuous relationships for other parameters of the search algorithm might provide further improvements. The influence that some of the flood mechanism related parameters (such as the number of survivors, or the selection pressure applied by the low-pressure scaling) have on the process of search may also provide an interesting area for further investigation.

REFERENCES

1. HYNEK, J.: *Genetické algoritmy a genetické programování*. Grada Publishing, a. s. Praha, 2008. ISBN 978-80-7300-218-3 [In Czech.]
2. ALBA, E., COTTA, C.: *Evolutionary Algorithms*. 2004. http://www.lcc.uma.es/%7Eccottap/papers/eas.pdf
3. MITCHELL, M.: *An Introduction to Genetic Algorithms*. A Bradford Book The MIT Press. Cambridge, Massachusetts, 1999. ISBN 0−262−13316−4
4. SADJADI, F. A.: *Comparison of fitness scaling functions in genetic algorithms with applications to optical processing*. Optical Information Systems II, Proceedings of SPIE: Vol. 5557, 2004.
5. BANERJEE, A.: *Fitness Scaling*. [quot. 11-20-2010]. http://www.cse.unr.edu/~banerjee/scaling.htm
6. LAROSE, D. T.: *Data Mining Methods and Models*. John Wiley & Sons. New Jersey, 2006. ISBN 978-04-7166-656-1
7. BUSETTI, F.: *Genetic algorithms overview*. 2001. http://www.vit.ac.in/academicresearch/res701/RES701DUMP%5CEvolutionary%20Algorithms%5Cgaweb.pdf
8. EIBEN, Á. E., ROBERT, H., MICHALEWICZ, Z.: *Parameter Control in Evolutionary Algorithms*. IEEE Transactions of Evolutionary Computation: 3, 1999. http://www.gpa.etsmtl.ca/cours/sys843/pdf/Eiben1999.pdf
9. THIERENS, D.: *Adaptive mutation rate control schemes in genetic algorithms*. Proceedings of the 2002 IEEE World Congress on Computational Intelligence: Congress on Evolutionary Computation, 2002.

http://dynamics.org/~altenber/UH_ICS/EC_REFS/GP_REFS/CEC/2002/GP_WCCI_2002/7315.PDF

10. SRINIVAS, M., PATNAIK, L. M.: *Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms*. IEEE Transactions on Systems, Man and Cybernetics: 24, 1994. http://eprints.iisc.ernet.in/archive/00006971/02/adaptive.pdf
11. KOZA, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press. Cambridge, Massachusetts, 1998. ISBN 0-262-11170-5
12. MONTANA, D. J.: *Strongly Typed Genetic Programming*. Evolutionary Computation: 3, 1995. http://vishnu.bbn.com/papers/stgp.pdf

[1] Selected Experimental Data

Maximum number of generations: 150.
Elitism: 3.
Population size: 500.
Maximum depth: 10.
Tree generator used: GrownTreeGenerator.
Functors used:

1. VariableFunctor<NavAction>,
2. ConstFactory<NavAction>(ACT_STAY, ACT_TURN_AROUND),
3. ConstFactory<TileType>(tile_empty, tile_wall),
4. CompareFunctor<NavAction>,
5. CompareFunctor<TileType>,
6. ConstFunctor<void>,
7. NumericConstFactory<bool>(0, 1),
8. IfAssign,
9. Logic functors: And, Or, Not,
10. PrgReturnFunctor(NavAction, 10).

Formal parameters used:

1. 3 x TileType (tile in front of the actor now and in past 2 turns).
2. 3 x NavAction (previous outputs of the program).

Selection method: FitnessRoulette.

1. **No Adaptive Mechanism**

Crossover probability: 1.0.
Mutation probability: 0.2.
Results:

# of run	max. fitness achieved
1	41
2	27
3	26
4	46
5	27

2. **The AGA Adaptive Mechanism**

Crossover probability: variable.
Mutation probability: variable.
Additional information: Uses the AGA mechanism with k1 = 1.0, k2 = 0.5, k3 = 1.0, k4 = 0.5.
Results:

# of run	max. fitness achieved
1	14
2	7
3	14
4	18
5	3

3. **The AVSMR Adaptive Mechanism**

Fitness scaling: none.
Crossover probability: 1.0.
Mutation probability: Basic mutation probability of 0.2; can be increased to 0.8 by the adaptive mechanism.
Additional information: Uses the AVSMR mechanism.

Results:

# of run	max. fitness achieved
1	89
2	26
3	42
4	47
5	89

4. The FMLPS Adaptive Mechanism

Crossover probability: 1.0.
Mutation probability: 0.2.
Additional information: Uses the FMLPS mechanism with 20 survivors, power scaling of 0.3, $N = 7$; $\Theta = 0.01$.
Results:

# of run	max. fitness achieved
1	89
2	39
3	63
4	36
5	46

5. The New Blood Adaptive Mechanism

Crossover probability: 1.0.
Mutation probability: 0.2.
Additional information: Uses the New Blood mechanism with 20 survivors, $N = 7$; $\Theta = 0.01$.
Results:

# of run	max. fitness achieved
1	89
2	48
3	89
4	45
5	63