Control and Cybernetics

vol. **29** (2000) No. 4

Grammars in genetic programming

by

W. Wieczorek[1] and Z. J. Czech[2]

[1]Institute of Computer Science, University of Silesia,
Sosnowiec, Poland
e-mail: wieczor@ultra.cto.us.edu.pl

[2]Institute of Computer Science, Silesia University of Technology,
Gliwice, Poland
e-mail: zjc@polsl.gliwice.pl

Abstract: The work consists of two parts. In the first part the idea of genetic programming is presented and the basic elements of a genetic programming system are described. In the second part, considering a selected example, we describe the results of investigations of the influence of program grammars on the efficiency of genetic programming.

Keywords: strongly typed genetic programming, genetic algorithms, grammars.

1. Introduction

Many problems of machine learning and artificial intelligence can be considered as problems of finding a computer program which produces the desired output data for the given input data. From this point of view a process of solving a problem reduces to searching a space of programs in order to find a proper one. The paradigm of genetic programming defines a way of searching the space such that the program solving a given problem is discovered with high probability.

Genetic programming, in addition to genetic algorithms (Holland, 1992), evolutionary programming (Fogel, 1995), evolution strategies (Schwefel, 1981), and classifier systems (Goldberg, 1989), belongs to the field of evolutionary computation, which mimics the evolutionary processes appearing in nature. Genetic programming developing in many directions uses the techniques known from genetic algorithms, such as the tournament selection (Angeline, 1994), coevolution (Angeline and Pollack, 1993), the steady-state population (Reynolds, 1992). Many variants of mutation are used, although Koza (1992) points out that mutation in genetic programming is applied sparingly. A vast amount of

Martin considered prefixed, postfixed and hybrid notations. Various criteria which can be applied to determine a program fitness are investigated, e.g. a criterion of program efficiency. The inclusion of information about the program size into the fitness measure was considered by Kinnear (1993) and Iba (1994). The separate group of research is devoted to a suitable choice of parameters of genetic programming. The method of determining a number of tests was proposed by Teller and Andre (1997). The interdependence between a population size and a number of generations was studied by Gathercole and Ross (1997).

In classical genetic programming the closure condition must be satisfied which means that all elements of a sought program, i.e. terminal symbols and functions, must be of the same type. Other solutions regarding the syntax of genetic programs were devised by Montana (1994) and Whigham (1995). Montana proposed a strongly typed genetic programming method in which the types of terminal symbols, functions and their arguments can be specified. An important aspect of such an approach is reducing the program space which is to be searched, since determining the types of functions and their arguments forbids semantically invalid programs. Whigham demonstrated an application of context-free grammars for defining and manipulating genetic programs. He also described a method of modifying the grammar productions in a course of an evolutionary process. An early work on using grammars in genetic algorithms was presented by Antonisse (1991). He proposed a general reformulation of the genetic algorithm that makes it appropriate to any problem representation that can be cast in a formal grammar.

The aim of this work is to verify the hypothesis that genetic programming is more effective if the programs are built and transformed according to some predefined syntactic rules established by taking into account the features of the problem to be solved. To this goal the classical programming system was compared with the system in which a "suitable" grammar was used.

This work consists of two parts. In the first part (Sections 2 and 3) the idea of genetic programming is presented and the basic elements of a genetic programming system are described. In the second part (Section 4), considering a selected example of the mine-infested area problem, we describe the results of investigations of the influence of program grammars on the efficiency of genetic programming. Section 5 concludes the work.

2. What is genetic programming?

Genetic programming makes possible to solve a problem without a tedious phase of constructing a program, which solves it, by a human. The genetic programming idea uses genetic algorithms (Holland, 1992) whose work is modeled upon the natural evolution of organisms. The evolution proceeds in accordance to the Darwinian principle of survival and reproduction of the fittest. A population in genetic algorithms is a set of problem solutions (individuals) usually repre-

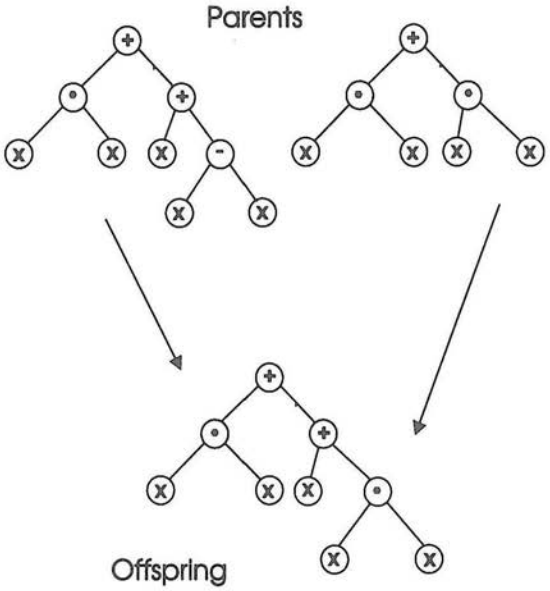
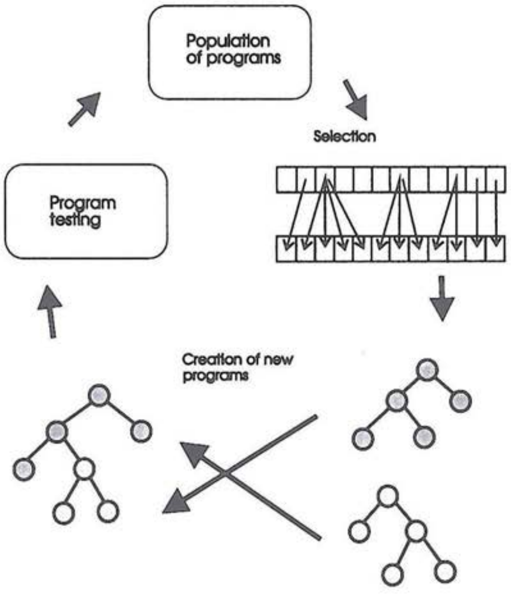


Figure 1. Crossover operation in genetic programming: as the result of crossing over $x^2 + (x + (x - x))$ and $x^2 + x^2$ we get $x^2 + (x + x^2)$



executed operations: the selection of best solutions and creation new ones out of them. While creating new solutions the operators of recombination known from genetic algorithms, such as crossover and mutation, are used. The new solutions replace other solutions in the population. In genetic programming the individuals of a population are computer programs. In order to illustrate the creation new programs from the two parent-programs they are represented as trees. New programs are built by removing a selected subtree from one tree and inserting it to another. The crossover operation is illustrated in Fig. 1 where the subtree representing $(x - x)$ in the left parent is replaced with the subtree x^2 coming from the right parent. The cycle of recurrent operations in genetic programming is the same as in genetic algorithms and is shown in Fig. 2.

3. Basic notions of genetic programming

3.1. Preparatory steps

There are six preparatory steps which must be accomplished before a searching process for a program to solve the problem can begin. These are as follows: (a) choice of terminal symbols constituting the set $T = \{t_1, t_2, \ldots, t_m\}$, (b) choice of functions constituting the set $F = \{f_1, f_2, \ldots, f_n\}$, (c) defining the fitness function, (d) defining the control parameters, (e) defining the termination criterion (Koza, 1992).

The terminal symbols, $t_i \in T$, and functions, $f_i \in F$, are the program components. For example, in Fig. 1 the internal nodes in the trees are arithmetic operations $+$, $-$, $*$. Each leaf in the trees must be a terminal symbol x. The choice of program components, i.e. the terminal symbols and functions, and a definition of the fitness function determine to a large extent the solution space which will be searched. The control parameters include the population size, the probabilities of crossover and mutation, the maximum tree size, etc.

3.2. Choice of terminal symbols and functions

A terminal symbol $t_i \in T$ can be a constant, for example $t_i = 3$, or a variable representing an input datum, or a measurement value coming from a gauge in an object under control. Every function $f_i \in F$ of a fixed arity can be an arithmetic operator ($+$, $-$, $*$, etc.), an arithmetic function (e.g. *sin*, *cos*, *exp*), a boolean operator (**and**, **or**, **not**), an alternative (**if-then-else**), an iterative operator (**while**), an arbitrarily defined function appropriate to the problem under consideration.

The crucial point in selecting the terminal symbols and functions is that using them one may express a solution to the problem. Furthermore, a closure condition is to be satisfied. We say that sets T and F satisfy the closure condition if every function from F accepts as its arguments the values returned by

3.3. Fitness function

The aim of the fitness function is to provide the basis for competition among individuals of a population. It is important that not only the correct solutions should obtain a high assessment (reward), but also every improvement of an individual should result in increasing of that reward.

There are several measures of fitness. One of them is raw fitness. Its definition depends on the problem of interest. For most problems raw fitness is defined as the sum of distances (errors) in all tests between the output result produced by a program for the test data and the expected value for that test. The raw fitness of an i-th program in a population in time t is defined as follows:

$$r(i,\, t) = \sum_{j=1}^{N} |W(i,\, j) - C(j)|$$

where $W(i,\, j)$ is the value returned by the i-th program for the j-th test, $C(j)$ is the correct answer for test j, and N is a number of tests. If the values returned by the programs are not numbers, but boolean values *true* or *false*, then the sum of distances is equivalent to the number of encountered errors. For certain problems raw fitness may not have an error form. For example, in problems of optimal control raw fitness may be the cost of particular control strategies (expressed as time, distance, profit in monetary units etc.). For other problems raw fitness may be a gained result, for example a number of points scored, an amount of food found, etc. (Koza, 1992).

4. The mine-infested area problem

Consider a rectangular area divided into $m \times n$ fields in which a fixed number, M, of invisible mines were placed. In a field with co-ordinates (initX, initY) an agent is positioned which may execute one of the following operations:

- Move to one of the eight neighbor fields (operations `GoN`, `GoS`, `GoW`, `GoE`, `GoNW`, `GoNE`, `GoSW`, `GoSE`).
- Open a field it stays on (operation `Open`). If this field contains a mine then the agent ends its work. Otherwise, as the result of opening the field a number from 0 to 8 indicating a number of mines around the field shows up.
- Determine the content of the field it stays on (operation `H`), or the content of an adjacent field (operations `N`, `S`, `W`, `E`, `NW`, `NE`, `SW`, `SE`).
- Determine the number of opened or unopened adjacent fields (operations `NumOp`, `NumIn`).

Solving the mine-infested area problem consists in finding a program which controls the agent. Such a program executed at most L times should open as large a part of the area as possible (surely, of at most $m \times n - M$ fields). The position (initX, initY). The execution $i+1$ begins from the position reached by the agent after execution i, $1 \leq i < L$. The agent's control terminates before L executions are completed if in the last execution a field with a mine was opened, or the given number of fields, K, $0 < K \leq m \times n - M$, was opened.

For the purpose of experiments the following values of parameters were fixed:

- $m = n = 5$ (defining the size of the mine-infested area, $m \times n = 5 \times 5$),
- $M = 5$ (the number of mines placed on the fields with co-ordinates $(1,0)$, $(2,1)$, $(3,1)$, $(2,2)$, $(1,3)$),
- initX $= 4$, initY $= 3$ (agent's initial position),
- $L = 25$ (the maximum number of program executions),
- $K = 18$ (the minimum number of fields which are to be opened).

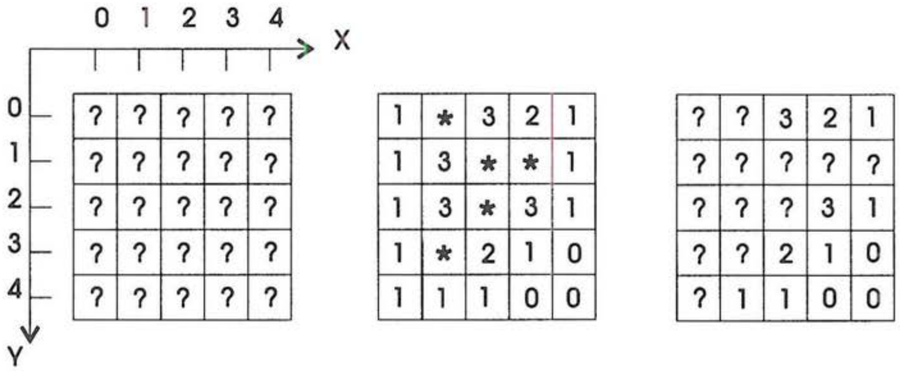


Figure 3. From left to right: the initial state of the area; the distribution of mines (indicated by asterisks) and numbers counting the mines around fields; a sample state of the area during the run of the program

Now let us define the elements of a system of genetic programming. Assume the following set of functions

$$F = \{\texttt{IfEq}, \texttt{Prog2}, \texttt{Prog3}\}$$

with four, two and three arguments, respectively. The function `IfEq` executes the subtrees represented by its first and second argument, `arg1` and `arg2`, respectively. Then it executes and returns a result of the execution of the third (if the execution result of `arg1` is equal to that of `arg2`) or fourth (if the execution result of `arg1` is not equal to that of `arg2`) argument, i.e. result of execution of `arg3` or `arg4`, respectively. The functions `Prog2` and `Prog3` execute in turn their arguments (`arg1`, `arg2` for `Prog2`, and `arg1`, `arg2`, `arg3` for `Prog3`) and

Let us specify the following set of terminal symbols:

$T =$ {NumOp, NumIn, Open, GoN, GoS, GoW, GoE, GoNW, GoNE, GoSW, GoSE, H, N, S, W, E, NW, NE, SW, SE, 0, 1, 2, 3, 4, 5, 6, 7, 8, ?, e}.

The Open operation returns the opened number. The operations from the „Go" group also return the content of a field the agent moved onto, in particular the value ?, if the field has not been opened yet. The symbol e denotes the edge of the area. It is returned, for example, by the operation NE when the current position of the agent is $(2, 0)$.

The function which evaluates a quality of generated programs counts the opened fields of the area. Hence, raw fitness value belongs to the range $[0, 20]$.

Having set the parameters: the probability of crossover equal to 0.7, the probability of mutation equal to 0.1 and the size of population equal to 200 we generated and evaluated[1] 7593 programs. A sample solution which was found is shown below.

```
(Prog3 (Prog2 (IfEq (Prog3 (Prog2 GoNE GoE) (IfEq
(IfEq NW Open 3 1) E Open Go E) (IfEq SE SE GoSW
NumIn)) (Prog2 (IfEq GoNW 4 GoNE GoNE) (Prog2 GoE
GoW)) (IfEq (IfEq 8 E 3 4) (IfEq S GoW NE 8) (Prog2
GoSW e) (Prog2 SW NumOp)) (Prog3 (Prog2 2 GoS) (Prog3
1 W SW) (Prog2 e S))) (IfEq (Prog3 (Prog2 6 GoNE)
(Prog2 5 NE) (Prog3 SE 5 GoSW)) (IfEq (IfEq GoSE 3
GoSE 6) (IfEq NE Open N 8) (IfEq NW Open 3 1) (Prog3 5
H NumIn)) (IfEq (IfEq 4 5 6 E) (IfEq 5 GoE GoW NE)
(Prog3 GoS GoSE 8) (Prog3 GoNW GoN 8)) (IfEq (IfEq E
GoSE 1 SE) (IfEq GoSW SE GoSE W) (Prog2 Open GoS) (IfEq
NE NumIn GoSE (IfEq 5 GoE GoW NE))))) (Prog3 (Prog2 (Prog3
(Prog3 GoNW GoN 8) (IfEq GoNE GoSE 1 5) (Prog2 H 5)) (IfEq
(IfEq NumOp 0 5 GoN) (Prog3 (Prog3 (Prog3 (Prog2 S NumOp)
(IfEq NW GoS GoNE 1) (IfEq GoW H ? GoW)) (IfEq (Prog3 E
1 GoS) (Prog3 N GoW 3) (Prog2 H S) (IfEq GoN GoW GoNW 8))
(Prog2 (Prog2 GoSE e) (IfEq GoSW N NE GoSW))) (Prog3 (IfEq
(Prog2 Open NumIn) (IfEq Open H GoNE GoW) (Prog3 SW E S)
(IfEq e 6 3 GoNW)) (IfEq (IfEq e Open 5 GoNE) (Prog2 GoSW
GoNE) (IfEq GoS GoN W NumIn) (IfEq GoS 7 2 4)) (IfEq (Prog2
NE GoW) E (Prog2 6 0) 2)) (Prog2 6 GoNE)) (Prog3 N Open
GoN) (Prog2 8 GoS))) (Prog3 (IfEq (Prog3 e GoSW GoN) (Prog2
7 NE) (IfEq SE H 3 5) (IfEq 7 NumIn 4 GoSW)) (Prog3 (Prog3
NW GoN GoSW) (Prog2 S NumOp) (IfEq GoSE SE NE NumIn)) (Prog2
(IfEq 7 GoNE SE 6) NumIn)) 5) 4)
```

[1]The evaluation consists in the execution of a program at least L times for the test area

Note that we stopped the genetic system when a given number of programs have been generated and evaluated, instead of executing a certain number of its cycles. This was because we used a genetic algorithm with a steady-state replacement (Syswerda, 1991). In such an algorithm a new individual is created in each generation, and it replaces another individual of the population. The most popular method of choosing an individual for replacement is based on a tournament selection. In this approach a set of r individuals is considered and the best individual (or the worst, while choosing a candidate for replacement) is selected.

When analyzing the program we may find in it some parts, e.g. `(Prog3 6 e 4)` or `(IfEq NumOp 5 NW 6)`, which do not make much sense from the semantic point of view, as the executions of arguments `6`, `e`, `4` and `5` are empty. However, such fragments may appear, as we need to satisfy the closure condition which says that every function and terminal symbol can be an argument of another function. In other words, the programs are generated according to a context free grammar with the following productions (nonterminal symbols are enclosed in angle parentheses):

Grammar 1

<program> → <F>
<F> → (`IfEq` <F> <F> <F> <F>) | (`Prog2` <F> <F>)
<F> → (`Prog3` <F> <F> <F>) | <T>
<T> → `NumOp` | `NumIn` | `Open` | `GoN` | `GoS` | `GoW` | `GoE` | `GoNW` | `GoNE` | `GoSW`
<T> → `GoSE` | `H` | `N` | `S` | `W` | `E` | `NW` | `NE` | `SW` | `SE` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `?` | `e`

In a conventionally written program we would distinguish in set T the symbols concerned with the "real activities" of the agent on the explored area (e.g. `GoN`, `Open`). Furthermore, we would determine the ones which provide the agent with the information enabling it to select a particular activity (e.g. `H`, `NumOp`, `7`). The latter symbols could be the building elements of expressions. Thus, let us define a grammar which from a point of view of semantics of the program, that we look for, is "more suitable":

Grammar 2

<program> → <instr>
<instr> → `GoN` | `GoS` | `GoW` | `GoE` | `GoNW` | `GoNE` | `GoSW` | `GoSE` | `Open`
<instr> → **if** <expr> = <expr> **then begin** <instr> **end else begin** <instr> **end**
<instr> → <instr> <instr>
<expr> → `H` | `N` | `S` | `W` | `E` | `NW` | `NE` | `SW` | `SE` | `NumOp` | `NumIn`
<expr> → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `?` | `e`

The fourth production of this grammar plays a similar role as the functions `Prog2` and `Prog3` in Grammar 1. Due to this production the constructed proalternative may contain arbitrarily long sequences of instructions. While manipulating the programs, i.e. in a course of their crossing and mutation, because the closure condition is not satisfied, we have to assure that for the subtrees being exchanged their dominating nodes are either all instructions or all expressions. To maintain the syntactic correctness of the generated programs we cannot replace instructions with expressions or expressions with instructions. For the parameters given above, the new grammar and the constraints mentioned before, an example of a genetic program we obtained is as follows:

```
GoN GoSW if H = N then begin GoW GoW GoSW Open GoNE
GoNE GoS GoSW GoW GoNE GoS GoN GoS end else begin if H
= NumIn then begin GoS GoN GoS GoN GoE GoN end else
begin if N = H then begin GoS if 7 = H then begin GoS
end else begin Open end end else begin GoSE end end
GoNE Open GoNW Open GoN GoE GoN end GoW Open GoE GoSE
Open GoSW GoE GoNE if N = NW then begin GoS GoSW GoSW
Open if S = H then begin GoN GoE GoN end else begin
GoN end end else begin GoSE GoE GoNE if NumIn = NW
then begin GoS GoSW GoNE GoW GoE GoSE Open GoSW if N =
NumIn then begin GoNE GoSW GoS end else begin GoNE GoE GoSW
GoNW GoNW GoW GoN GoW GoE GoE Open GoS GoSW GoSE GoNE GoW
GoE GoSE Open GoSW GoE GoNE if SE = NW then begin GoS GoSW
GoN if 1 = NumOp then begin GoW end else begin GoW end end
else begin GoSE GoS GoN GoS end GoNW GoSE GoSW Open GoS
GoNW if H = 6 then begin if E = 2 then begin GoSE Open Open
end else begin GoSW end if H = 6 then begin GoSE end else
begin GoE GoSW GoW GoNW GoW if NumOp = S then begin GoNW
end else begin GoNE GoNW GoW Open end GoSE GoS Open GoNW
GoNE end end else begin GoN GoW end end GoW GoE GoSW GoN
GoN GoW GoNE GoS GoN GoS end else begin GoN GoS end GoNW
GoSE GoNE GoS GoN GoS end GoW Open
```

In order to compare the efficiency of genetic programming for Grammar 1 and 2 both systems were executed 100 times. An execution was terminated when the solution to the problem under consideration was found. After each of 100 executions a number of generated and evaluated programs[2] was counted. The histogram depicting the number of executions of the systems (E) versus the number of generated and evaluated programs (P) which guaranteed finding the final solution is presented in Fig. 4. Note that the application of the more general Grammar 1 which results in the wider exploration of the space of programs has two aspects—positive and negative. The positive aspect follows from the fact that we may reach such the areas, unreachable for Grammar 2, whose searching causes a fast convergence to the proper solution. Considering those executions of the systems which succeeded before the 4,000 programs were generated (the first pair of bars in Fig. 4) it turns out that the system with Grammar 1 is slightly better. The negative aspect consists in penetrating those areas of program space which are worthless in terms of finding the final solution. If we compare the numbers of those executions which did not give a success before the 16,000 programs were generated, it is clear that the system with Grammar 1 is much worse (the black bar of the last pair in Fig. 4 is significantly shorter than the white one).

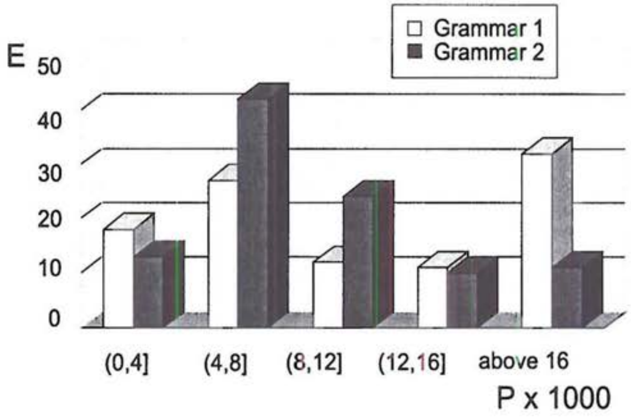


Figure 4. The number of executions of the systems versus the number of generated and evaluated programs

Yet another comparison of the genetic programming systems with Grammar 1 and 2 is shown in Fig. 5, which depicts the probability of success in finding the

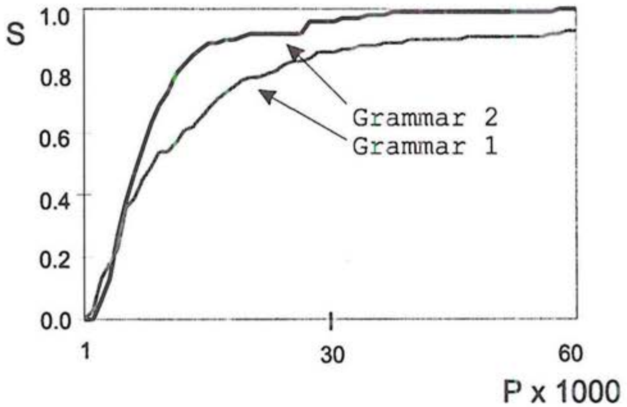


Figure 5. The probability of success versus the number of generated and evalu-

final solution (S) versus the number of generated and evaluated programs (P). Let the termination criterion of the genetic programming systems be the generation and evaluation of 15,000 programs. Based on the graph in Fig. 5 we can then expect that for 100 executions approximately 70 will end successfully in case of Grammar 1, and approximately 90 in case of Grammar 2. If we increase the number of programs to 60,000, the probabilities of success will increase to 0.9 for Grammar 1, and to 1.0 for Grammar 2. It also turned out that for Grammar 1 the number of generated programs even as large as 200,000 was not sufficient to achieve the probability of success equal to 1.0. Thus, it seems that application of a more precise grammar increases significantly the chances of finding the solution after generating an appropriate number of programs.

5. Conclusions

Genetic programming is a dynamically progressing research direction in a current development of evolutionary computation. In genetic programming we search through a space of computer programs in order to find the best one, i.e. the fittest. The search is conducted by creating a population of executable computer programs, in which programs compete with each other. Weak programs die out, whereas the strong ones reproduce. With regard to the selection of a suitable grammar for the generated programs we showed that for the problem of the mine-infested area, the more precise grammar of generated programs, the more effective genetic programming.

References

ANTONISSE, H.J. (1991) A grammar-based genetic algorithm. In *Foundations of Genetic Algorithms*, Morgan Kaufmann, 193–204.

ANGELINE, P.J. (1994) Genetic programming and emergent intelligence. In Kinnear, Jr., K.E., ed., *Advances in Genetic Programming*, MIT Press.

ANGELINE, P.J. and POLLACK, J.B. (1993) Competitive environments evolve better solutions for complex tasks. In Forrest, S., ed., *Proc. 5th International Conference on Genetic Programming*, ICGA-93, Morgan Kaufmann.

FOGEL, D.B. (1995) *Evolutionary computation: toward a new philosophy of machine intelligence.* IEEE Press, Piscataway, NJ, USA.

GATHERCOLE, C. and ROSS, P. (1997) Small populations over many generations can beat large populations over few generations in genetic programming. In Koza, J.R. et al., eds., *Genetic Programming 1997: Proc. 2nd Annual Conf.*, Stanford University, CA, USA, Morgan Kaufmann.

GOLDBERG, D.E. (1989) *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Publishing.

HOLLAND, J.H. (1992) *Adaptation in natural and artificial systems, an introductory analysis with applications to biology, control and artificial intelli-*

IBA, H., DE GARIS, H. and SATO, T. (1994) Genetic programming using a minimum description length principle. In Kinnear, Jr., K.E., ed., *Advances in Genetic Programming.* MIT Press.

KEITH, M.J. and MARTIN, M.C. (1994) Genetic programming in C++: implementation issues. In Kinnear, Jr., K.E., ed., *Advances in Genetic Programming.* MIT Press.

KINNEAR, JR., K.E. (1993) Generality and difficulty in genetic programming. In Forrest, S., ed., *Proc. 5th International Conference on Genetic Algorithms.* ICGA-93, Morgan Kaufman.

KOZA, J.R. (1992) *Genetic programming: on the programming of computers by means of natural selection.* A Bradford Book, The MIT Press.

MONTANA, D.J. (1994) Strongly Typed Genetic Programming. BBN technical report #7866, Cambridge, Mass., USA.

REYNOLDS, C.W. (1992) An evolved, vision-based behavioral model of coordinated group motion. In Meyer, Wilson, eds., *From Animals to Animats, Proc. of Simulation of Adaptive Behaviour.* MIT Press.

SCHWEFEL, H.-P. (1981) *Numerical optimization of computer models.* John Wiley, Chichester, UK.

SYSWERDA, G. (1991) A study of reproduction in generational and steady state genetic algorithms. In Rawlings, G.J.E., ed., *Foundations of Genetic Algorithms.* Morgan Kaufmann, Indiana University.

TELLER, A. and ANDRE, D. (1997) Automatically choosing the number of fitness cases: the rational allocation of trials. In Koza, J.R. et al., eds., *Genetic Programming 1997: Proc. 2nd Annual Conf.*, Stanford University, CA, USA, Morgan Kaufmann.

WHIGHAM, P.A. (1995) Grammatically-based genetic programming. In Rosca, J.P., ed., *Proc. of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Tahoe City, California, USA.