Contents

List of Figures

List of Tables

List of Algorithms

Chapter I

Improving the Search by Encoding Multiple Solutions in a Chromosome

Mihai Oltean [I.1],

We investigate the possibility of encoding multiple solutions of a problem in a single chromosome. The best solution encoded in an individual will represent (will provide the fitness of) that individual. In order to obtain some benefits the chromosome decoding process must have the same complexity as in the case of a single solution in a chromosome. Three Genetic Programming techniques are analyzed for this purpose: Multi Expression Programming, Linear Genetic Programming and Infix Form Genetic Programming. Numerical experiments show that encoding multiple solutions in a chromosome greatly improves the search process.

I.1 Introduction

Evolutionary Algorithms (EAs) [9, 10] are powerful tools used for solving difficult real-world problems.

This paper describes a new paradigm called *Multi Solution Programming* (MSP) that may be used for improving the search performed by the Evolutionary Algorithms. The main idea is to encode multiple solutions (more than one) in a single chromosome. The best solution encoded in a chromosome will represent (will provide the fitness of) that individual.

[I.1]Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş Bolyai University, Kogalniceanu 1, 3400 Cluj-Napoca, Romania, `moltean@cs.ubbcluj.ro`, `www.cs.ubbcluj.ro/~moltean`

I. IMPROVING THE SEARCH BY ENCODING MULTIPLE SOLUTIONS IN A CHROMOSOME

This special kind of encoding is useful when the complexity of the decoding process is similar to the complexity of the decoding process of chromosomes encoding a single solution of the problem being solved.

Note that the Multi Solution Programming is not a particular technique, but a paradigm intended to be used in conjunction with an Evolutionary Algorithm. MSP refers to a new way of encoding solutions in a chromosome.

GP techniques are very suitable for the MSP paradigm because the trees offer an implicit multiple solutions representation: each sub-tree may be considered as a potential solution of the problem. Three Genetic Programming (GP) [11, 12] variants are tested with the proposed model: Multi Expression Programming (MEP) [17, 18], Linear Genetic Programming (LGP) [4, 5, 6, 16] and Infix Form Genetic Programming (IFGP) [19].

Multi Expression Programming uses a chromosome encoding similar to the way in which C or *Pascal* compilers translate mathematical expressions in machine code. Note that MEP was originally designed [17] to encode multiple solutions in a chromosome. A MEP variant that encodes a single solution in a chromosome is proposed and tested in this paper.

A Linear Genetic Programming chromosome is a sequence of C language instructions. Each instruction has a destination variable and several source variables. One of the variables is usually chosen to provide the output of the program. A LGP variant encoding multiple solutions in a chromosome is proposed in this paper.

Infix Form Genetic Programming chromosomes are strings of integer encoding mathematical expressions in infix form. Each IFGP chromosome encodes multiple solutions of the problem being solved [19]. A IFGP variant encoding a single solution in a chromosome is proposed and tested in this paper.

All the solutions represented in a MSP individual should be decoded by traversing the chromosome only once. Partial results are stored by using Dynamic Programming.

Several numerical experiments with the considered GP techniques are performed by using 4 symbolic regression problems. For each test problem the relationships between the success rate and the population size and the code length are analyzed.

Results show that Multi Solutions Programming significantly improves the evolutionary search. For all considered test problems the evolutionary techniques encoding multiple solutions in a chromosome are significantly better than the similar techniques encoding a single solution in a chromosome.

The paper is organized as follows. Motivation for this research is presented in section I.2. Test problems used to asses the performance of the considered algorithms are given in section I.3. Multi Expression Programming and its counterpart Single Expression Programming are described in section I.4. Linear Genetic Programming and Multi-Solution Linear Genetic Programming are described in section I.5. Infix Form Genetic Programming and Single-Solution Infix Form Genetic Programming are described in section I.6. Conclusions and further work directions are given in section I.7.

I.2 MULTIPLE SOLUTION PROGRAMMING

This section tries to answer two fundamental questions:

- Why encoding multiple solutions in the same chromosome?

- How to efficiently encode multiple solutions in the same chromosome in order to obtain some benefits?

The answer for the first question is motivated by the No Free Lunch Theorems for Search [23]. There is neither practical nor theoretical evidence that one of the solutions encoded in a chromosome is better than the others. More than that, Wolpert and McReady [23] proved that we cannot use the search algorithm's behavior so far for a particular test function to predict its future behavior on that function.

The second question is more difficult than the first one. There is no general prescription on how to encode multiple solutions in the same chromosome. In most cases this involves creativity and imagination. However, some general suggestions can be given.

(i) In order to obtain some benefits from encoding more than one solution in a chromosome we have to spend a similar effort (computer time, memory etc) as in the case of encoding a single solution in a chromosome. For instance, if we have 10 solutions encoded in chromosome and the time needed to extract and decode these solutions is 10 times larger than the time needed to extract and process one solution we got nothing. In this case we can not talk about an useful encoding.

(ii) We have to be careful when we want to encode a multiple solutions in a variable length chromosome (for instance in a standard GP chromosome), because this kind of chromosome will tend to increase its size in order to encode more and more solutions. And this could lead to bloat [2, 15].

(iii) Usually encoding multiple solutions in a chromosome might require storing the partial results. Sometimes this can be achieved by using the Dynamic Programming [3] technique. This kind of model is employed by the techniques described in this paper.

I.3 TEST PROBLEMS AND METRIC OF PERFORMANCE

For assessing the performance of the considered algorithms we use several symbolic regression problems [11]. The aim of symbolic regression is to find a mathematical expression that satisfies a set of fitness cases.

The test problems used in the numerical experiments are:

$$F_1(x) = x^4 - x^3 + x^2 - x,$$

$$F_2(x) = x^4 + x^3 + x^2 + x,$$

$$F_3(x) = x^4 + 2 * x^3 + 3 * x^2 + 4 * x,$$

$$F_4(x) = x^6 - 2x^4 + x^2.$$

Test problem f_2 is also known as *quartic polynomial* and f_4 is known as *sextic polynomial* [11, 12].

For each function 20 fitness cases have been randomly generated with a uniform distribution over the [0, 10] interval.

For each test problem the relationships between the success rate, the population size and the code length are analyzed. The success rate is computed as:

$$Success\,rate = \frac{The\,number\,of\,successful\,runs}{The\,total\,number\,of\,runs}. \tag{I.1}$$

Roughly speaking the quality of a GP chromosome is the average distance between the expected output and the obtained output [11]. A run is considered successful if the fitness of the best individual in the last generation is less than 0.01.

I.4 MULTI EXPRESSION PROGRAMMING

In this section, *Multi Expression Programming* (MEP) [17] is briefly described.

I.4.1 MEP Representation

MEP genes are represented by substrings of a variable length. The number of genes per chromosome is constant. This number defines the length of the chromosome. Each gene encodes a terminal or a function symbol. A gene encoding a function includes pointers towards the function arguments. Function arguments always have indices of lower values than the position of that function in the chromosome.

This representation is similar to the way in which **C** and **Pascal** compilers translate mathematical expressions into machine code [1].

MEP representation ensures that no cycle arises while the chromosome is decoded (phenotypically transcripted). According to this representation scheme the first symbol of the chromosome must be a terminal symbol. In this way only syntactically correct programs (MEP individuals) are obtained.

Example

A representation where the numbers on the left positions stand for gene labels is employed here. Labels do not belong to the chromosome, they being provided only for explanation purposes.

For this example we use the set of functions $F = \{+, *\}$, and the set of terminals $T = \{a, b, c, d\}$. An example of chromosome using the sets F and T is given below:

1: a
2: b
3: + 1, 2
4: c
5: d
6: + 4, 5
7: * 3, 6

I.4.2   Decoding MEP Chromosomes and Fitness Assignment Process

In this section is described the way in which MEP individuals are translated into computer programs and the way in which the fitness of these programs is computed.

This translation is achieved by reading the chromosome top-down. A terminal symbol specifies a simple expression. A function symbol specifies a complex expression obtained by connecting the operands specified by the argument positions with the current function symbol.

For instance, genes 1, 2, 4 and 5 in the previous example encode simple expressions formed by a single terminal symbol. These expressions are:

$E_1 = a,$
$E_2 = b,$
$E_4 = c,$
$E_5 = d,$

Gene 3 indicates the operation + on the operands located at positions 1 and 2 of the chromosome. Therefore gene 3 encodes the expression:

$E_3 = a + b.$

Gene 6 indicates the operation + on the operands located at positions 4 and 5. Therefore gene 6 encodes the expression:

$E_6 = c + d.$

Gene 7 indicates the operation * on the operands located at position 3 and 6. Therefore gene 7 encodes the expression:

$E_7 = (a + b) * (c + d).$

E_7 is the expression encoded by the whole chromosome.

Each MEP chromosome is allowed to encode a number of expressions equal to the chromosome length. Each of these expressions is considered as being a potential solution of the problem.

The value of these expressions may be computed by reading the chromosome top down. Partial results are computed by dynamic programming and are stored in a conventional manner.

I. IMPROVING THE SEARCH BY ENCODING MULTIPLE SOLUTIONS IN A CHROMOSOME

Usually the chromosome fitness is defined as the fitness of the best expression encoded by that chromosome.

For instance, if we want to solve symbolic regression problems the fitness of each sub-expression E_i may be computed using the formula:

$$f(E_i) = \sum_{k=1}^{n} |o_{k,i} - w_k|, \tag{I.2}$$

where $o_{k,i}$ is the obtained result by the expression E_i for the fitness case k and w_k is the targeted result for the fitness case k. In this case the fitness needs to be minimized.

The fitness of an individual is set to be equal to the lowest fitness of the expressions encoded in chromosome:

$$f(C) = \min_i f(E_i). \tag{I.3}$$

When we have to deal with other problems we compute the fitness of each sub-expression encoded in the MEP chromosome and the fitness of the entire individual is given by the fitness of the best expression encoded in that chromosome.

I.4.3 Search Operators

Search operators used within MEP algorithm are crossover and mutation. Considered search operators preserve the chromosome structure. All offspring are syntactically correct expressions.

Crossover By crossover two parents are selected and are recombined. For instance, within the uniform recombination the offspring genes are taken randomly from one parent or another.

Example

Let us consider the two parents C_1 and C_2 given in Table I.1. The two offspring O_1 and O_2 are obtained by uniform recombination as shown in Table I.1.

Mutation Each symbol (terminal, function of function pointer) in the chromosome may be target of mutation operator. By mutation some symbols in the chromosome are changed. To preserve the consistency of the chromosome its first gene must encode a terminal symbol.

Example

Consider the chromosome C given in Table I.2. If the boldfaced symbols are selected for mutation an offspring O is obtained as shown in Table I.2.

6

Table I.1. MEP uniform recombination.

Parents		Offspring	
C_1	C_2	O_1	O_2
1: *b*	1: *a*	1: *a*	1: *b*
2: *** 1, 1**	2: *b*	2: *** 1, 1**	2: *b*
3: **+ 2, 1**	3: + 1, 2	3: **+ 2, 1**	3: + 1, 2
4: *a*	4: *c*	4: *c*	4: *a*
5: *** 3, 2**	5: *d*	5: *** 3, 2**	5: *d*
6: *a*	6: + 4, 5	6: + 4, 5	6: *a*
7: **- 1, 4**	7: * 3, 6	7: **- 1, 4**	7: * 3, 6

Table I.2. MEP mutation.

C	O
1: *a*	1: *a*
2: * 1, 1	2: * 1, 1
3: *b*	3: **+ 1, 2**
4: * 2, 2	4: * 2, 2
5: *b*	5: *b*
6: + **3**, 5	6: **+ 1**, 5
7: *a*	7: *a*

I.4.4   MEP Algorithm

In this paper we use a steady-state [22] as underlying mechanism for Multi Expression Programming. The algorithm starts by creating a random population of individuals. The following steps are repeated until a stop condition is reached. Two parents are selected using a selection procedure. The parents are recombined in order to obtain two offspring. The offspring are considered for mutation. The best offspring replaces the worst individual in the current population if the offspring is better than the worst individual.

The algorithm returns as its answer the best expression evolved along a fixed number of generations.

I.4.5   Single Expression Programming

The MEP variant encoding a single solution in a chromosome is called Single Expression Programming (SEP). The expression encoded in a SEP chromosome is given by its last gene.

Example

Consider again the chromosome given in section I.4.1.

1: a
2: b
3: + 1, 2
4: c
5: d
6: + 4, 5
7: * 3, 6

The SEP expression encoded by this chromosome is:

$$E = (a + b) * (c + d).$$

I.4.6   Numerical Experiments with MEP and SEP

Several numerical experiments with Multi Expression Programming and Single Expression Programming are performed in this section using the test problems described in section I.3.

The general parameters of the MEP and SEP algorithms are given in Table I.3. The same settings are used for Multi Expression Programming and for Single Expression Programming.

For all problems the relationship between the success rate and the chromosome length and the population size is analyzed. The success rate is computed as the number of successful runs over the total number of runs (see section I.3).

Table I.3. Parameters of the MEP and SEP algorithms for solving symbolic regression problems.

Parameter	Value
Number of generations	51
Crossover probability	0.9
Mutations	2 / chromosome
Function set	$F = \{+, -, *, /, \}$
Terminal set	Problem inputs
Selection	Binary Tournament

Experiment 1

In this experiment the relationship between the success rate and the chromosome length is analyzed. The population size was set to 50 individuals. Other parameters of the MEP and SEP algorithms are given in Table I.3. Results are depicted in Figure I.1.

Figure I.1 shows that Multi Expression Programming significantly outperforms Single Expression Programming for all the considered test problems and for all the considered parameter setting. More than that, large chromosomes are better for MEP than short chromosomes. This is due to the multi-solution ability: increasing the chromosome length leads to more solutions encoded in the same individual. The easiest problem is f_2. MEP success rate for this problem is over 90% when the number of instructions in a chromosome is larger than 10. The most difficult problem is f_3. For this problem and with the parameters given in Table I.3, the success rate of the MEP algorithm never increases over 70%. However, these results are very good compared to those obtained by SEP (the success rate never increases over 10% for the test problem f_3).

Experiment 2

In this experiment the relationship between the success rate and the population size is analyzed. The number of instructions in a MEP or SEP chromosome was set to 10. Other parameters for the MEP and SEP are given in Table I.3. Results are depicted in Figure I.2.

Figure I.2 shows that MEP performs better than SEP. Problem f_2 is the easiest one and problem f_3 is the most difficult one.

I.5 LINEAR GENETIC PROGRAMMING

Linear Genetic Programming (LGP) [4, 16] uses a specific linear representation of computer programs. Instead of the tree-based GP expressions [11] of a functional programming language (like **LISP**), programs of an imperative language (like **C**) are evolved.
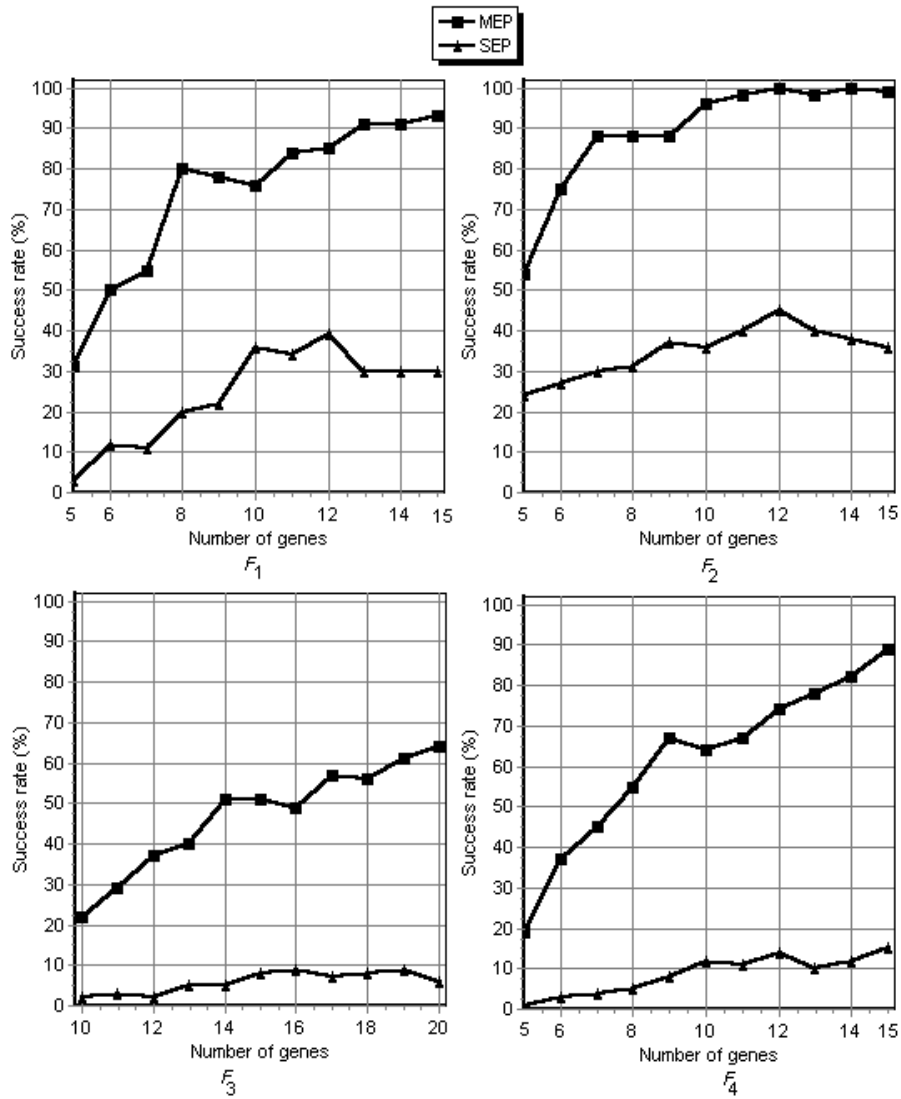
Figure I.1. The relationship between the success rate and the number of instructions in a chromosome. Results are averaged over 100 runs.

Figure I.2. The relationship between the success rate and the population size. Results are averaged over 100 runs.

I. IMPROVING THE SEARCH BY ENCODING MULTIPLE SOLUTIONS IN A CHROMOSOME

I.5.1 LGP Representation

A LGP individual is represented by a variable-length sequence of simple C language instructions. Instructions operate on one or two indexed variables (registers) r or on constants c from predefined sets. The result is assigned to a destination register, e.g. $r_i = r_j * c$.
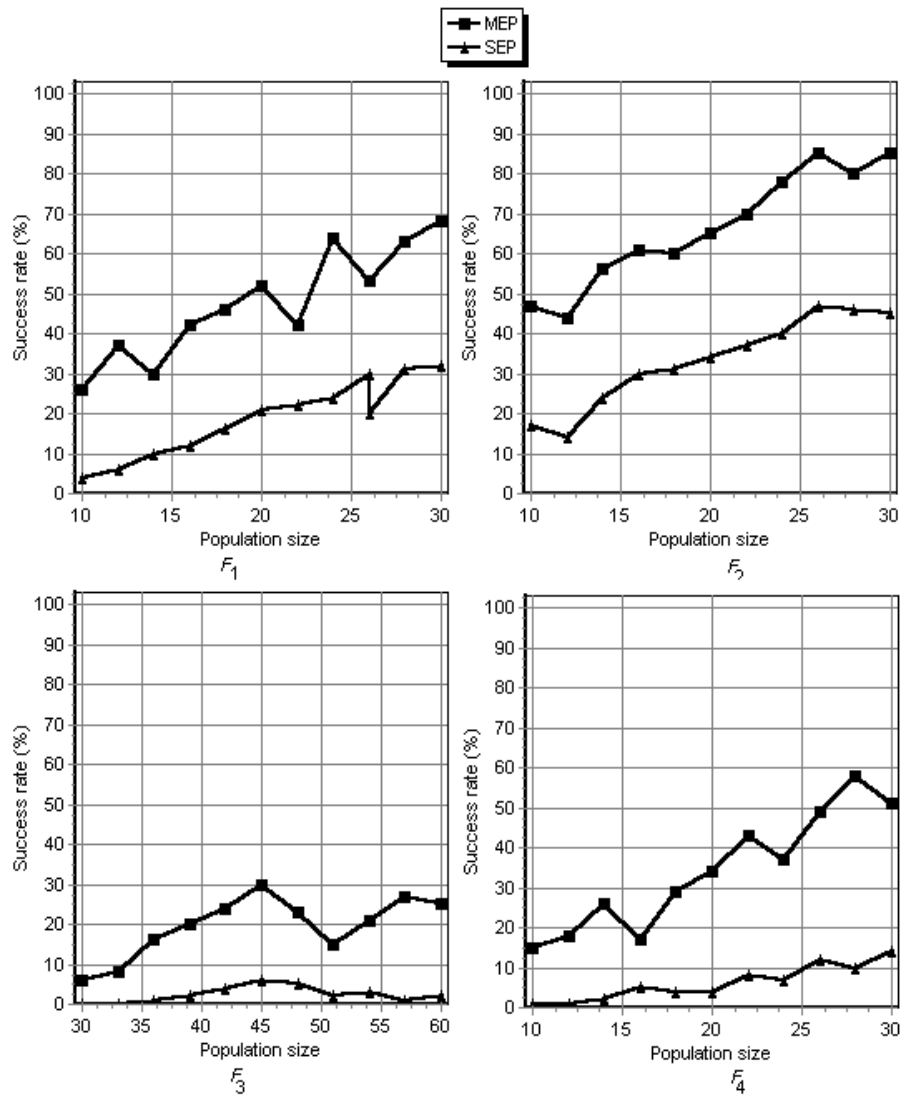
An example of the LGP program is the following one:

```
void LGP(double r[8])
{
    r[0] = r[5] + 73;
    r[7] = r[3] − 59;
    r[2] = r[5] + r[4];
    r[6] = r[7] * 25;
    r[1] = r[4] − 4;
    r[7] = r[6] * 2;
}
```

I.5.2 Decoding LGP Individuals

A linear genetic program can be turned into a functional representation by successive replacements of variables starting with the last effective instruction [4].

Usually one of the variables ($r[0]$) is chosen as the output of the program. This choice is made at the beginning of the program and is not changed during the search process. In what follows we will denote this LGP variant as Single-Solution Linear Genetic Programming (SS-LGP).

I.5.3 Genetic Operators

The variation operators used in conjunction with Linear Genetic Programming are crossover and mutation. Standard LGP crossover works by exchanging continuous sequences of instructions between parents [4].

Two types of standard LGP mutations are usually used: micro mutation and macro mutation. By micro mutation an operand or an operator of an instruction is changed [4].

Macro mutation inserts or deletes a random instruction [4].

Since we are interested more in multi-solutions paradigm rather than in variable length chromosomes we will use fixed length chromosomes in all experiments performed in this paper. Genetic operators used in numerical experiments are uniform crossover and micro mutation.

LGP uniform crossover LGP uniform crossover works between instructions. The offspring's genes (instructions) are taken with a 50% probability from the parents.

Example

12

Let us consider the two parents C_1 and C_2 given in Table I.4. The two offspring O_1 and O_2 are obtained by uniform recombination as shown in Table I.4.

Table I.4. LGP uniform recombination.

Parents		Offspring	
C_1	C_2	O_1	O_2
$r[5]=r[3]*r[2];$	$\boldsymbol{r[2]=r[0]+r[3]};$	$r[5]=r[3]*r[2];$	$\boldsymbol{r[2]=r[0]+r[3]};$
$r[3]=r[1]+6;$	$\boldsymbol{r[1]=r[2]*r[6]};$	$r[1]=r[2]*r[6];$	$r[3]=r[1]+6;$
$r[0]=r[4]*r[7];$	$\boldsymbol{r[4]=r[6]\text{-}4};$	$r[0]=r[4]*r[7];$	$\boldsymbol{r[4]=r[6]\text{-}4};$
$r[5]=r[4]\text{-}r[1];$	$\boldsymbol{r[6]=r[5]/r[2]};$	$r[5]=r[4]\text{-}r[1];$	$\boldsymbol{r[6]=r[5]/r[2]};$
$r[1]=r[6]*7;$	$\boldsymbol{r[2]=r[1]+7};$	$\boldsymbol{r[2]=r[1]+7};$	$r[1]=r[6]*7;$
$r[0]=r[0]+r[4];$	$\boldsymbol{r[1]=r[2]+r[4]};$	$\boldsymbol{r[1]=r[2]+r[4]};$	$r[0]=r[0]+r[4];$
$r[2]=r[3]/r[4];$	$\boldsymbol{r[0]=r[4]*3};$	$\boldsymbol{r[0]=r[4]*3};$	$r[2]=r[3]/r[4];$

LGP mutation LGP mutation works inside of a LGP instruction. By mutation each operand (source or destination) or operator is affected with a fixed mutation probability.

Example

Consider an individual C which is affected by mutation. An offspring O is obtained as shown in Table I.5 (modified variables are written in boldface):

Table I.5. LGP mutation.

C	O
$r[5] = r[3] * r[2];$	$r[5] = r[3] * r[2];$
$r[3] = r[1] + 6;$	$r[3] = \boldsymbol{r[6]} + \boldsymbol{r[0]};$
$r[0] = r[4] * r[7];$	$r[0] = r[4] + r[7];$
$r[5] = r[4] \text{-} r[1];$	$\boldsymbol{r[4]} = r[4] \text{-} r[1];$
$r[1] = r[6] * 7;$	$r[1] = r[6] * \boldsymbol{2};$
$r[0] = r[0] + r[4];$	$r[0] = r[0] + r[4];$
$r[2] = r[3] / r[4];$	$\boldsymbol{r[0]} = r[3] / r[4];$

I.5.4 LGP Algorithm

LGP uses a modified steady-state algorithm [4]. Initial population is randomly generated. The following steps are repeated until a termination criterion is reached: Four individuals are randomly selected from the current population. The best two of them are considered the winners of the tournament and they will act as parents.

I. IMPROVING THE SEARCH BY ENCODING MULTIPLE SOLUTIONS IN A CHROMOSOME

The parents are recombined and the offspring are mutated and then replace the losers of the tournament.

I.5.5 Multi Solution Linear Genetic Programming

The LGP structure is enriched as follows:

- *(i)* We allow as each destination variable to represent the output of the program. In the standard LGP only one variable is chosen to provide the output.

- *(ii)* We check for the program output after each instruction in chromosome. This is again different from the standard LGP where the output was checked after the execution of all instructions in a chromosome.

After each instruction, the value stored in the destination variable is considered as a potential solution of the problem. The best value stored in one of the destination variables is considered for fitness assignment purposes.

Example

Consider the chromosome C given below:

```
void LGP(double r[8])
{
    r[5] = r[3] * r[2];
    r[3] = r[1] + 6;
    r[0] = r[4] * r[7];
    r[6] = r[4] - r[1];
    r[1] = r[6] * 7;
    r[2] = r[3]/r[4];
}
```

Instead of encoding the output of the problem in a single variable (as in SS-LGP) we allow that each of the destination variables ($r[5]$, $r[3]$, $r[0]$, $r[6]$, $r[1]$ or $r[2]$) to store the program output. The best output stored in these variables will provide the fitness of the chromosome.

For instance, if we want to solve symbolic regression problems, the fitness of each destination variable $r[i]$ may be computed using the formula:

$$f(r[i]) = \sum_{k=1}^{n} |o_{k,i} - w_k|,$$

where $o_{k,i}$ is the result obtained in variable $r[i]$ for the fitness case k, w_k is the targeted result for the fitness case k and n is the number of fitness cases. For this problem the fitness needs to be minimized.

The fitness of an individual is set to be equal to the lowest fitness of the destination variables encoded in the chromosome:

$$f(C) = \min_i f(r[i]).$$

Thus, we have a Multi-Solution LGP program at two levels: first level is given by the possibility that each variable to represent the output of the program and the second level is given by the possibility of checking for the output at each instruction in the chromosome.

I.5.6 Numerical Experiments with LGP and MS-LGP

In this section several experiments with SS-LGP and MS-LGP are performed. For this purpose we use several well-known symbolic regression problems described in section I.3.

The general parameters of the LGP algorithms are given in Table I.6. The same settings are used for Multi-Solution LGP and for Single-Solution LGP.

Table I.6. Parameters of the LGP and MS-LGP algorithms for solving symbolic regression problems.

Parameter	Value
Number of generations	51
Crossover probability	0.9
Mutations	2 / chromosome
Function set	$F = \{+, -, *, /, \}$
Terminal set	Problem inputs + 4 supplementary registers
Selection	Binary Tournament

For all problems the relationship between the success rate, the chromosome length and the population size is analyzed. The success rate is computed as the number of successful runs over the total number of runs as described in section I.3.
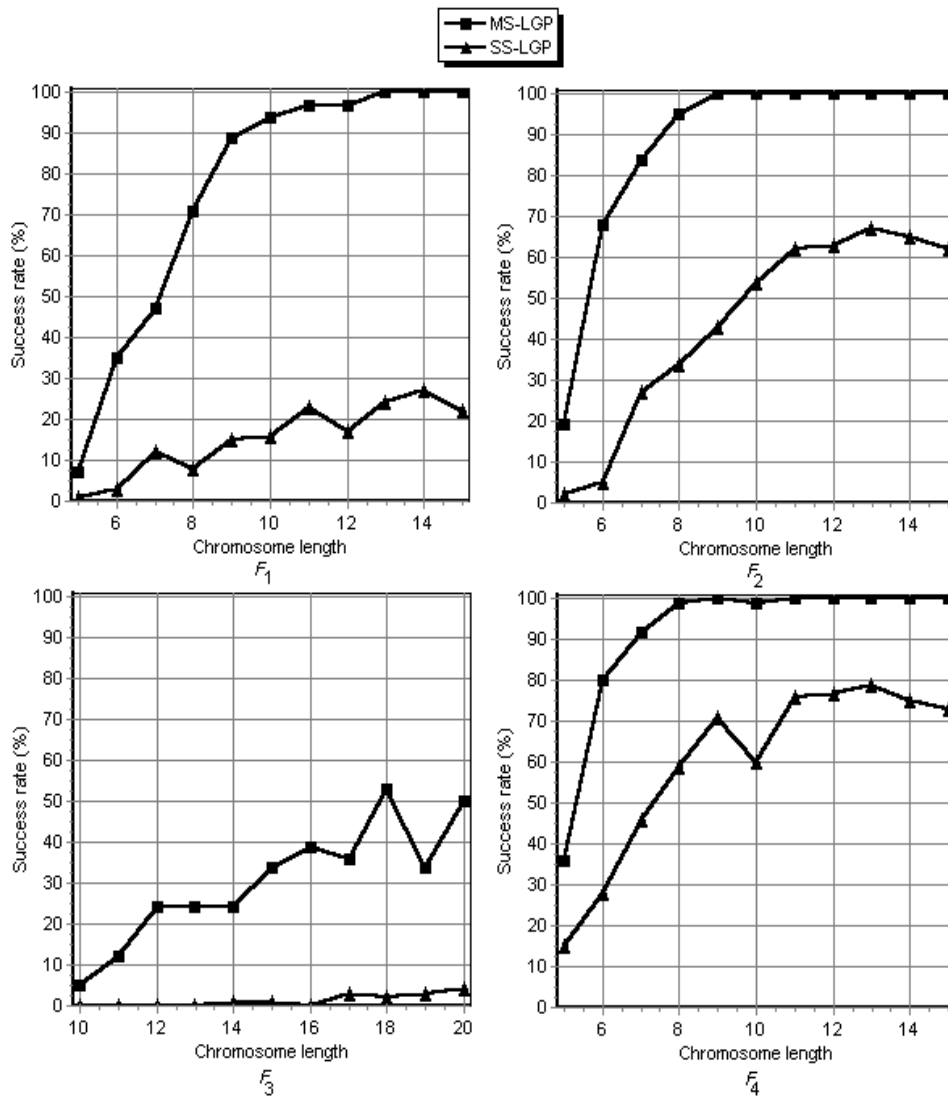
Experiment 1

In this experiment the relationship between the success rate and the chromosome length is analyzed. The population size was set to 50 individuals. Other parameters of the LGP are given in Table I.6. Results are depicted in Figure I.3.

Figure I.3 shows that Multi-Solution LGP significantly outperforms Single-Solution LGP for all the considered test problems and for all the considered parameter setting. As in the case of MEP larger chromosomes are better for MS-LGP than shorter ones. This is due to the multi-solution ability: increasing the chromosome length leads to more solutions encoded in the same individual. The most difficult problem is f_3. For this problem the success rate of SS-LGP never increases over 5%).

Experiment 2

Figure I.3. The relationship between the success rate and the number of instructions in a chromosome. Results are averaged over 100 runs.

In this experiment the relationship between the success rate and the population size is analyzed. The number of instructions in a LGP chromosome was set to 12. Other parameters for the LGP are given in Table I.6. Results are depicted in Figure I.4.
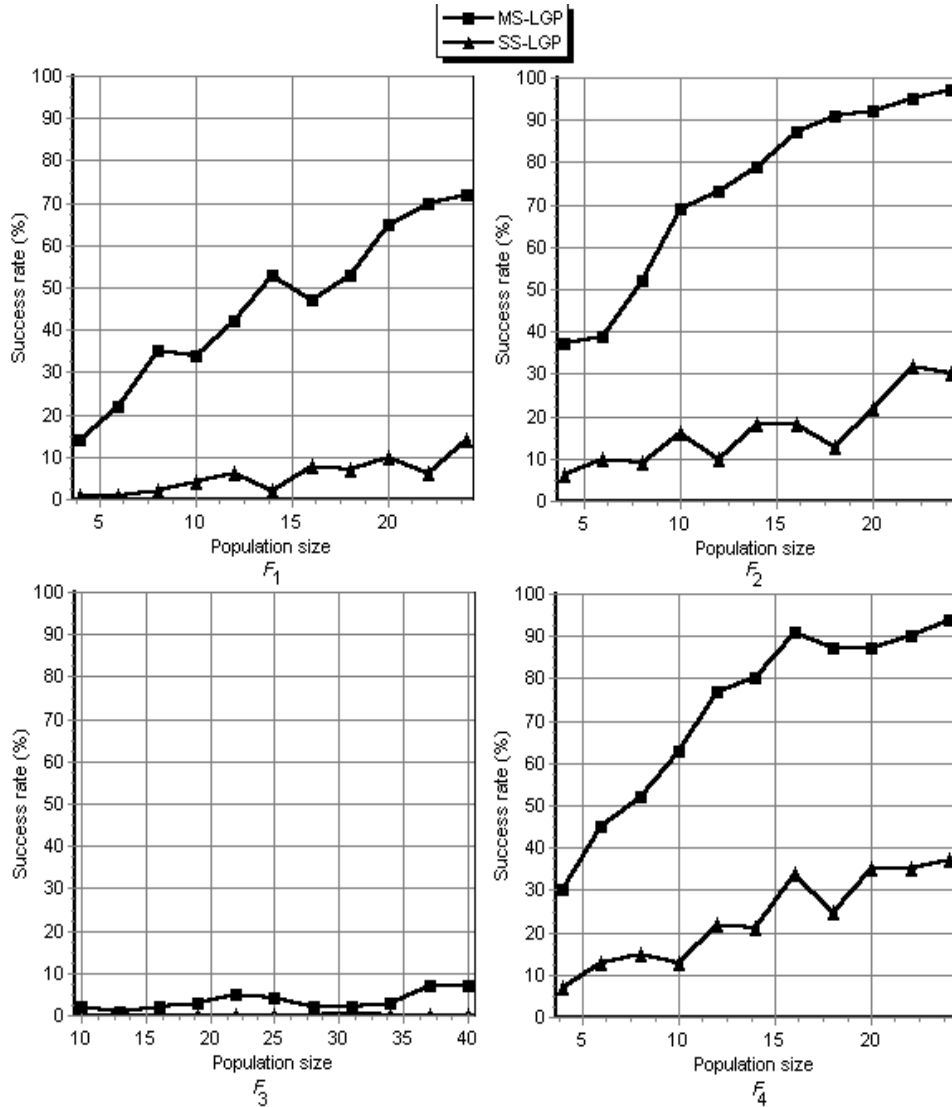
Figure I.4. The relationship between the success rate and the population size. Results are averaged over 100 runs.

Figure I.4 shows that Multi-Solution LGP performs better than Single-Solution LGP. For the test problem f_3 the success rate of SS-LGP is 0% (no run successfull).

I.6   Infix Form Genetic Programming

In this section *Infix Form Genetic Programming* (IFGP) [19] technique is described. IFGP uses linear chromosomes for solution representation (i.e. a chromosome is a string of genes). An IFGP chromosome usually encodes several solutions of a problem.

In [19] IFGP was used for solving several classification problems taken from [20]. The conclusion was that IFGP performs similar and sometimes even better that Linear Genetic Programming and Neural Networks [4, 20].

I.6.1   Prerequisite

We denote by F the set of function symbols (or operators) that may appear in a mathematical expression. F usually contains the binary operators $\{+, -, *, /\}$. By *Number_of_Operators* we denote the number of elements in F. A correct mathematical expression also contains some terminal symbols. The set of terminal symbols is denoted by T. The number of terminal symbols is denoted by *Number_of_Variables*.

Thus, the symbols that may appear in a mathematical expression are $T \cup F \cup \{'(', ')'\}$. The total number of symbols that may appear in a valid mathematical expression is denoted by *Number_of_Symbols*.

By C_i we denote the value on the ith gene in a IFGP chromosome and by G_i the symbol in the ith position in the mathematical expression encoded into an IFGP chromosome.

I.6.2   IFGP Individual Representation

In this section we describe how IFGP individuals are represented and how they are decoded in order to obtain a valid mathematical expression.

Each IFGP individual is a fixed size string of genes. Each gene is an integer number in the interval [0 .. *Number_Of_Symbols* - 1]. An IFGP individual can be transformed into a functional mathematical expression by replacing each gene with an effective symbol (a variable, an operator or a parenthesis).

Example

If we use the set of functions symbols $F = \{+, *, -, /\}$, and the set of terminals $T = \{a, b\}$, the following chromosome

C = 7, 3, 2, 2, 5

is a valid chromosome in IFGP system.

I.6.3   IFGP Decoding Process

We begin to decode this chromosome into a valid mathematical expression. In the first position (in a valid mathematical expression) we may have either a variable, or

an open parenthesis. That means that we have *Number_Of_Variables* + 1 possibilities to choose a correct symbol on the first position. We put these possibilities in order: the first possibility is to choose the variable x_1, the second possibility is to choose the variable x_2 ... the last possibility is to choose the closed parenthesis ')'. The actual value is given by the value of the first gene of the chromosome. Because the number stored in a chromosome gene may be larger than the number of possible correct symbols for the first position we take only the value of the first gene *modulo* number of possibilities for the first gene. Note that the *modulo* operator is used in a similar context by Grammatical Evolution [21].

Generally, when we compute the symbol stored in the ith position in expression we have to compute first how many symbols may be placed in that position. The number of possible symbols that may be placed in the current position depends on the symbol placed in the previous position. Thus:

(i) if the previous position contains a variable (x_i), then for the current position we may have either an operator or a closed parenthesis. The closed parenthesis is considered only if the number of open parentheses so far is larger than the number of closed parentheses so far.

(ii) if the previous position contains an operator, then for the current position we may have either a variable or an open parenthesis.

(iii) if the previous position contains an open parenthesis, then for the current position we may have either a variable or another open parenthesis.

(iv) if the previous position contains a closed parenthesis, then for the current position we may have either an operator or another closed parenthesis. The closed parenthesis is considered only if the number of open parentheses so far is larger than the number of closed parentheses.

Once we have computed the number of possibilities for the current position it is easy to determine the symbol that will be placed in that position: first we take the value of the corresponding gene modulo the number of possibilities for that position. Let p be that value ($p = C_i \ mod \ Number\_Of\_Possibilities$). The pth symbol from the permitted symbols for the current is placed in the current position in the mathematical expression. (Symbols that may appear into a mathematical expression are ordered arbitrarily. For instance we may use the following order: x_1, x_2, ..., +, -, *, /, '(', ')'.)

All chromosome genes are translated excepting the last one. The last gene is used by the correction mechanism (see below).

The obtained expression usually is syntactically correct. However, in some situations the obtained expression needs to be repaired. There are two cases when the expression needs to be corrected:

The last symbol is an operator (+, -, *, /) or an open parenthesis. In that case a terminal symbol (a variable) is added to the end of the expression. The added symbol is given by the last gene of the chromosome.

I. IMPROVING THE SEARCH BY ENCODING MULTIPLE SOLUTIONS IN A CHROMOSOME

The number of open parentheses is greater than the number of closed parentheses. In that case several closed parentheses are automatically added to the end in order to obtain a syntactically correct expression.

Remark. If the correction mechanism is not used, the last gene of the chromosome will not be used.

Example

Consider the chromosome $C = 7, 3, 2, 0, 5, 2$ and the set of terminal and function symbols previously defined ($T = \{a, b\}$, $F = \{+, -, *, /\}$).

For the first position we have 3 possible symbols (a, b and '('). Thus, the symbol in the position $C_0 \bmod 3 = 1$ in the array of possible symbols is placed in the current position in expression. The chosen symbol is b, because the index of the first symbol is considered to be 0.

For the second position we have 4 possibilities ($+$, $-$, $*$, $/$). The possibility of placing a closed parenthesis is ignored since the difference between the number of open parentheses and the number of closed parentheses is zero. Thus, the symbol '/' is placed in position 2.

For the third position we have 3 possibilities (a, b and '('). The symbol placed on that position is an open parenthesis '('.

In the fourth position we have 3 possibilities again (a, b and '('). The symbol placed on that position is the variable a.

For the last position we have 5 possibilities ($+$, $-$, $*$, $/$) and the closed parenthesis ')'. We choose the symbol on the position 5 **mod** $5 = 0$ in the array of possible symbols. Thus the symbol '+' is placed in that position.

The obtained expression is $E = b \ / \ (a+$.

It can be seen that the expression E it is not syntactically correct. For repairing it we add a terminal symbol to the end and then we add a closed parenthesis. Now we have obtained a correct expression:

$$E = b/(a + a).$$

The expression tree of E is depicted in Figure I.5.
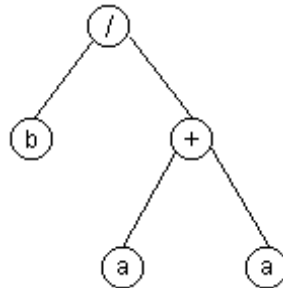
Figure I.5. The expression tree of $E = b \ / \ (a + a)$.

I.6.4 Fitness Assignment Process

In this section we describe how IFGP may be efficiently used for solving symbolic regression problems.

A GP chromosome usually stores a single solution of a problem and the fitness is normally computed using a set of fitness cases.

Instead of encoding a single solution, an IFGP individual is allowed to store multiple solutions of a problem. The fitness of each solution is computed in a conventional manner and the solution having the best fitness is chosen to represent the chromosome.

In the IFGP representation each sub-tree (sub-expression) is considered as a potential solution of a problem. For example, the previously obtained expression (see section I.7) contains 4 distinct solutions (sub-expressions):

$$E_1 = a,$$
$$E_2 = b,$$
$$E_3 = a + a,$$
$$E_4 = b/(a + a).$$

The fitness of an IFGP individual using the same formulas used for MEP in section I.4.2.

I.6.5 Search Operators

Search operators used within the IFGP model are recombination and mutation. These operators are similar to the genetic operators used in conjunction with binary encoding [8]. By recombination two parents exchange genetic material in order to obtain two offspring. In this paper only two-point recombination is used. Mutation operator is applied with a fixed mutation probability (p_m). By mutation a randomly generated value over the interval [0, *Number_of_Symbols*-1] is assigned to the target gene.

I.6.6 IFGP Algorithm

A steady-state [22] algorithm (similar to that used by MEP) is used as underlying mechanism for IFGP. The algorithm starts with a randomly chosen population of individuals. The following steps are repeated until a termination condition is reached. Two parents are chosen at each step using binary tournament selection [8]. The selected individuals are recombined with a fixed crossover probability p_c. By recombining two parents, two offspring are obtained. The offspring are mutated and the best of them replaces the worst individual in the current population (only if the offspring is better than the worst individual in population).

The algorithm returns as its answer the best expression evolved for a fixed number of generations.

I.6.7 Single Solution Infix Form Genetic Programming

As IFGP was originally designed to encode multiple solutions in a single chromosome [19] we modify the technique in order to encode a single solution / chromosome. The obtained variant is called Single-Solution IFGP (SS-IFGP). The expression encoded in a SS-IFGP chromosome is that representing the entire individual (the largest expression).

Example

Consider again the chromosome:

C = 7, 3, 2, 2, 5.

The SS-IFGP expression is:

$E = b/(a + a)$.

I.6.8 Numerical Experiments with IFGP and SS-IFGP

Several numerical experiments with Infix Form Genetic Programming and Single-Solution Infix Form Genetic Programming are performed in this section using the test problems described in section I.3.

The general parameters of the IFGP and SS-IFGP algorithms are given in Table I.7. The same settings are used for Infix Form Genetic Programming and for Single Solution Infix Form Genetic Programming.

Table I.7. Parameters of the IFGP and SS-IFGP algorithms for solving symbolic regression problems.

| Parameter	Value
Number of generations	51
Crossover probability	0.9
Mutations	2 / chromosome
Function set	$F = \{+, -, *, /\}$
Terminal set	Problem inputs
Selection	Binary Tournament

For all problems the relationship between the success rate and the chromosome length and the population size is analyzed. The success rate is computed as the number of successful runs over the total number of runs.

Experiment 1

In this experiment the relationship between the success rate and the chromosome length is analyzed. The population size was set to 50 individuals. Other parameters

of the IFGP and SS-IFGP algorithms are given in Table I.7. Results are depicted in Figure I.6.
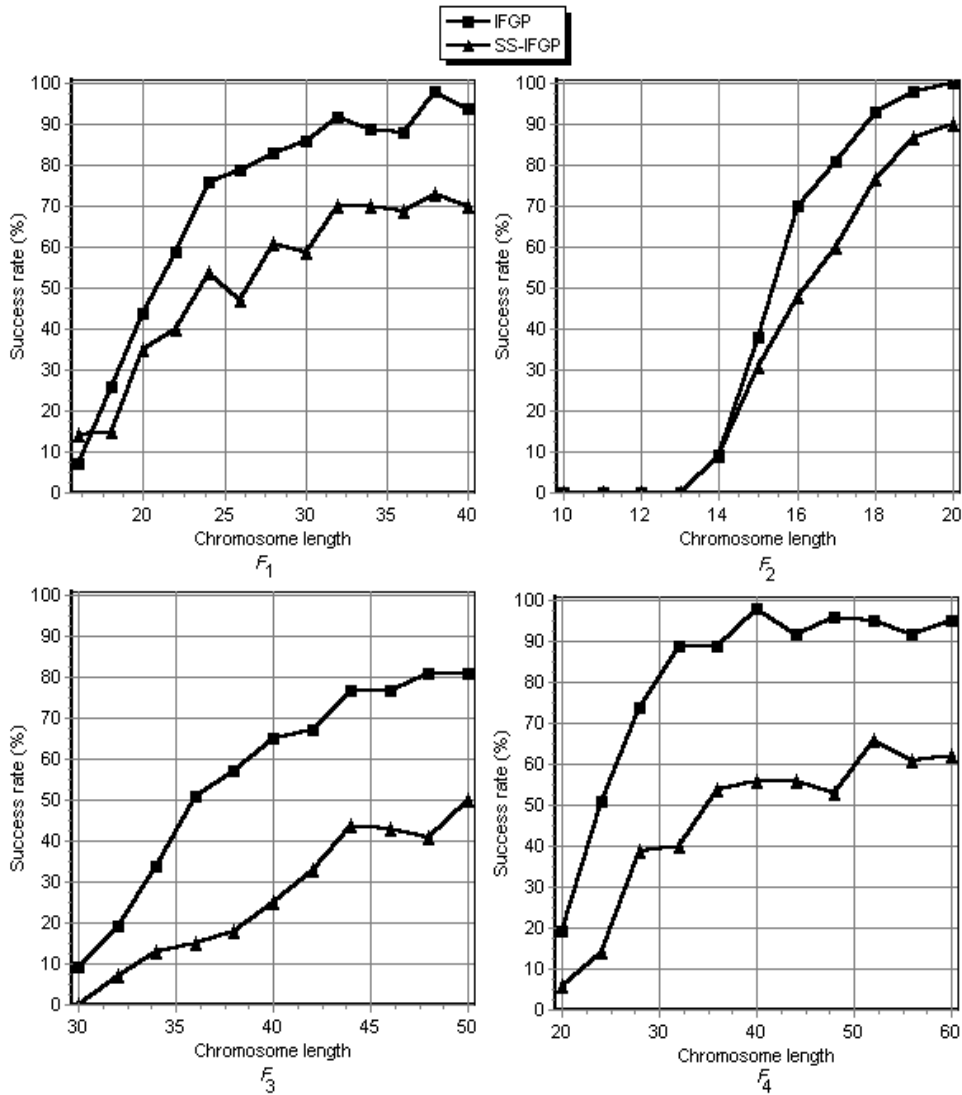
Figure I.6. The relationship between the success rate and the number of symbols in a chromosome. Results are averaged over 100 runs.

Figure I.6 shows that Infix Form Genetic Programming outperforms Single-Solution Infix Form Genetic Programming for the considered test problems. However the differences between MS-IFGP and SS-IFGP are not so significant as in the case of MEP and LGP.
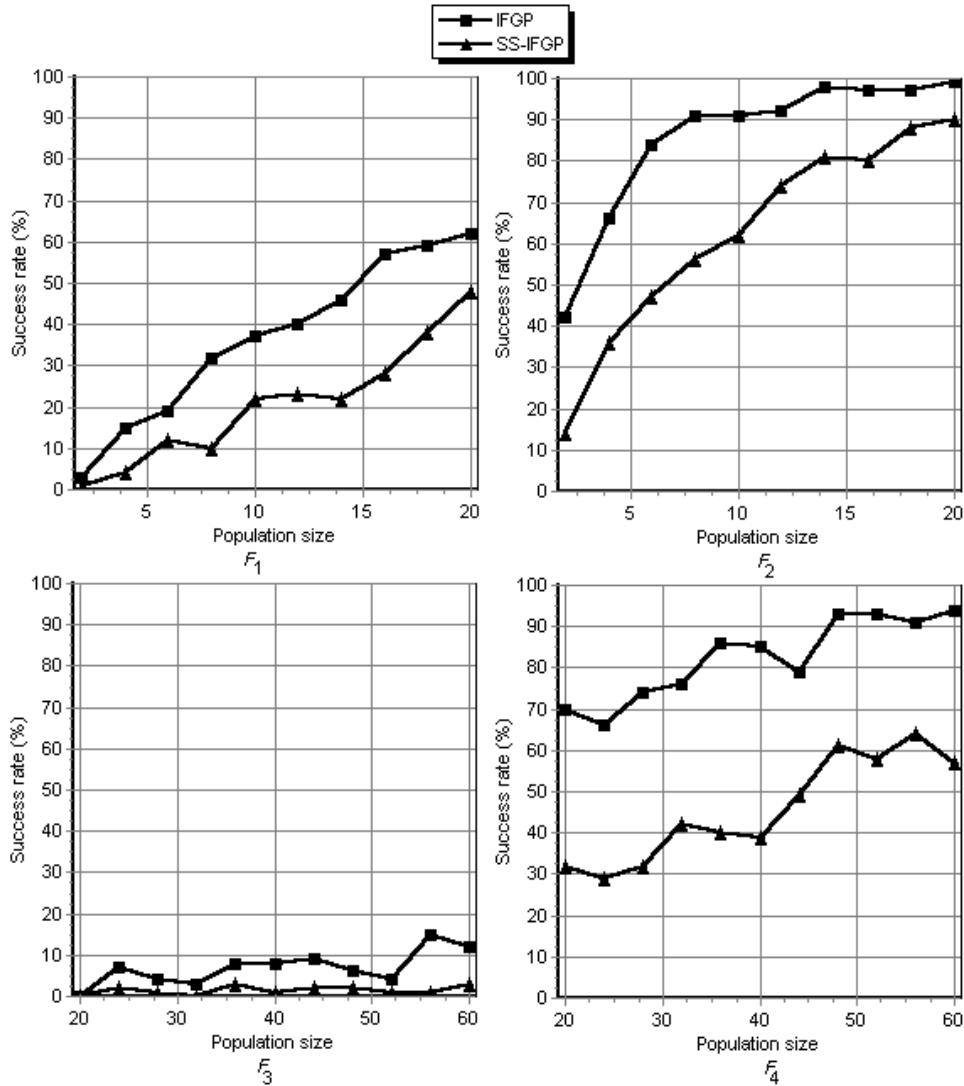
Experiment 2

In this experiment the relationship between the success rate and the population

size is analyzed. The number of symbols in a IFGP or SS-IFGP chromosome was set to 30. Other parameters for the IFGP and SS-IFGP are given in Table I.7. Results are depicted in Figure I.7.

Figure I.7. The relationship between the success rate and the population size. Results are averaged over 100 runs.

Figure I.7 shows that Infix Form Genetic Programming performs better than Single-Solution Infix Form Genetic Programming. Problem f_2 is the easiest one (the IFGP success is 100 % when the population size is 20) when the chromosome length is 20 and problem f_3 is the most difficult one.

I.7 Conclusions and Further Work

The ability of encoding multiple solutions in a single chromosome has been analyzed in this paper for 3 GP techniques: Multi Expression Programming, Linear Genetic Programming and Infix Form Genetic Programming. It has been show how to efficiently decode the considered chromosomes by traversing them only once.

Numerical experiments have shown that Multi-Solution Programming significantly improves the evolutionary search for all the considered test problems. There are several reasons for which Multi Solution Programming performs better than Single Solution Programming:

- MSP chromosomes act like variable-length chromosomes even if they are stored as fixed-length chromosomes. The variable-length chromosomes are better than fixed-length chromosomes because they can easily store expressions of various complexities,

- MSP algorithms perform more function evaluations than their SSP counterparts. However the complexity of decoding individuals is the same for both MSP and SSP techniques.

The multi-solution ability will be investigated within other evolutionary models.

Bibliography

[1] A. Aho, R. Sethi and J. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 1986.

[2] W. Banzhaf and W.B. Langdon, Some Considerations on the Reason for Bloat, Genetic Programming and Evolvable Machines, Vol. 3, pp. 81-91, 2002.

[3] R. Bellman, Dynamic Programming, Princeton, Princeton University Press, New Jersey, 1957.

[4] M. Brameier and W. Banzhaf, A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining, IEEE Transactions on Evolutionary Computation, Vol. 5, pp. 17-26, IEEE Press, NY, 2001.

[5] M. Brameier and W. Banzhaf, Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming, The 4th European Conference on Genetic Programming, E. Lutton, J. Foster, J. Miller, C. Ryan and A. Tettamanzi (Editors), pp. 38-50, Springer-Berlin, 2002.

[6] M. Brameier and W. Banzhaf, Evolving Teams of Predictors with Linear Genetic Programming, Genetic Programming and Evolvable Machines, Vol. 2, pp. 381-407, 2001.

[7] T.H. Cormen, C.E. Leiserson and R.R. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.

[8] D. Dumitrescu, B. Lazzerini, L. Jain, A. Dumitrescu, Evolutionary Computation, CRC Press, Boca Raton, FL, 2000.

[9] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989.

[10] J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975.

[11] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, 1992.

[12] J.R. Koza, Genetic Programming II: Automatic Discovery of Reusable Subprograms, MIT Press, Cambridge, MA, 1994.

[13] J.R. Koza, (et al.), Genetic Programming III: Darwinian Invention and Problem Solving, Morgan Kaufmann, San Francisco, CA, 1999.

[14] W.B. Langdon, and R. Poli, Genetic Programming Bloat with Dynamic Fitness, First European Workshop on Genetic Programming, W. Banzhaf, R. Poli, M. Schoenauer and T. C. Fogarty (Editors), pp. 96-112, Springer-Verlag, Berlin, 1998.

[15] S. Luke and L. Panait, Fighting Bloat with Nonparametric Parsimony Pressure, in Proceedings of the Seventh International Conference on Parallel Problem Solving from Nature, J. J. Merelo Guervos, P. Adamidis, H-G. Beyer, J. L. Fernandez-Villacanas Martin, and H-P. Schwefel (Editors), Springer-Verlag, Berlin, 2002.

[16] P. Nordin, A Compiling Genetic Programming System that Directly Manipulates the Machine Code, K. E. Kinnear, Jr. (Editor), Advances in Genetic Programming, pp. 311-331, MIT Press, 1994.

[17] M. Oltean and C. Groşan, Evolving Evolutionary Algorithms using Multi Expression Programming, The 7th European Conference on Artificial Life, W. Banzhaf (et. al), (Editors), LNCS 2801, pp. 651-658, Springer-Verlag, Berlin, 2003.

[18] M. Oltean, Solving Even-Parity Problems with Multi Expression Programming, Te 5th International Workshop on Frontiers in Evolutionary Algorithm, K. Chen (et. al), (Editors), North Carolina, pp. 315-318, 2003.

[19] M. Oltean and C. Groşan, Solving Classification Problems using Infix Form Genetic Programming, The 5th International Symposium on Intelligent Data Analysis, M. Berthold (et al), (Editors), LNCS 2810, pp. 242-252, Springer, Berlin, 2003.

[20] L. Prechelt, PROBEN1 - A Set of Neural Network Problems and Benchmarking Rules, Technical Report 21, University of Karlsruhe, 1994.

[21] C. Ryan, J. J. Collins, and M. O'Neill, Grammatical Evolution: Evolving Programs for an Arbitrary Language, First European Workshop on Genetic Programming, W. Banzhaf, R. Poli, M. Schoenauer and T. C. Fogarty (Editors), Springer-Verlag, Berlin, 1998.

[22] G. Syswerda, Uniform Crossover in Genetic Algorithms, Schaffer, J.D., (Editor), Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 2-9, Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[23] D.H. Wolpert and W.G. McReady, No Free Lunch Theorems for Search, technical report SFI-TR-05-010, Santa Fe Institute, 1995.

[24] D.H. Wolpert and W.G. McReady, No Free Lunch Theorems for Optimisation, IEEE Transaction on Evolutionary Computation, Vol. 1, pp. 67-82, 1997.