Applied Soft Computing xxx (2008) xxx–xxx

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

An autonomous GP-based system for regression and classification problems

Mihai Oltean *, Laura Dioşan

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Kogalniceanu 1, 400084 Cluj-Napoca, Romania

ARTICLE INFO

Article history:
Received 14 November 2006
Received in revised form 27 February 2008
Accepted 11 March 2008
Available online xxx

PACS:
01.30. –y

Keywords:
Genetic Programming
Adaptive strategies
Autonomous systems
Symbolic regression
Classification

ABSTRACT

The aim of this research is to develop an autonomous system for solving data analysis problems. The system, called Genetic Programming-Autonomous Solver (GP-AS) contains most of the features required by an autonomous software: it decides if it knows or not how to solve a particular problem, it can construct solutions for new problems, it can store the created solutions for later use, it can improve the existing solutions in the idle-time it can efficiently manage the computer resources for fast running speed and it can detect and handle failure cases. The generator of solutions for new problems is based on an adaptive variant of Genetic Programming. We have tested this part by solving some well-known problems in the field of symbolic regression and classification. Numerical experiments show that the GP-AS system is able to perform very well on the considered test problems being able to successfully compete with standard GP having manually set parameters.

1. Introduction

Autonomous systems [5] are of high interest due to their ability to perform various tasks without relying on human interference. A computer program is autonomous if the user does not have to change its parameters when a new problem has to be solved. For achieving this goal the following features must be implemented:

- The ability to perform well under significant uncertainties in the system and environment for extended periods of time.
- The ability to recognize if it knows how to solve a problem or not.
- The ability to create new solutions for new problems.
- The ability to learn from previous experience.
- The ability to use the computer resources in a efficient manner.
- The ability to identify failure conditions.
- The ability to improve the existing solutions during the idle-time.

The purpose of this research is to build a system meeting the previously described criteria. We have limited our attention to symbolic regression and classification problems due to several reasons:

- These problems are of great interest because they arise in many real-world applications.
- The input and output has a well-defined structure, which is easy to handle: arrays of symbols.

Our system, called Genetic Programming-Autonomous Solver (GP-AS) consists of six main parts: a Decision Maker, a Trainer, a Solver Repository, a Repository Manager, an Idle-time Manager and a Failure Manager. When a problem is presented to the system, the Decision Maker will decide which Solver will try to solve that problem by sending a request to the Repository Manager which in turn will query its database. If no suitable Solver is found, the Decision Maker will activate the Trainer, which will try to train a Solver for that problem. This new Solver will be added to the Repository for later use. The training of new Solvers is performed by using examples which are requested from the user. When no request is sent to the system it will try to improve the existing solutions by calling Idle-time Manager. Another component is able to detect possible failure of the systems and to act accordingly.

* Corresponding author.
E-mail addresses: moltean@cs.ubbcluj.ro (M. Oltean), lauras@cs.ubbcluj.ro (L. Dioşan).

1568-4946/$ – see front matter © 2008 Elsevier B.V.. All rights reserved.
doi:10.1016/j.asoc.2008.03.008

The Trainer has been used for solving several interesting and difficult problems: even-parity and other 22 real-world problem taken from PROBEN1 [41].

It is difficult to compare the GP-AS system with some other problem solvers because the experimental conditions are different. Comparisons between GP and other techniques have been previously performed by other authors [10,17]. Here we have performed a raw comparison for the numerical experiments required to evolve Solvers. The results obtained by GP-AS are generally worse then those obtained by the systems that use a fixed population size and a fixed maximal tree height. There are still few cases where GP-AS Trainer performs better than standard GP. However, the comparison is not fair because experimental conditions are different. The GP-AS system uses an adaptive mechanism for population size and chromosome size and this means that it does not know which are the optimal values of these parameters for a given problem. This is different from other systems where several experimental trials have been performed in order to find this information.

The paper is organized as follows: related work is reviewed in Section 2. The extension of Genetic Programming, used for training Solvers, is described in Section 3. The way in which multiple outputs can be easily handled by GP is minutely discussed in Section 3.2. Fitness assignment in the case of regression (classification) problems is described in Sections 3.3.1 and 3.3.2. The proposed GP-AS system is presented in Section 4. The structure of the input required by the GP-AS system is thoroughly discussed in Section 4.1. Decision Maker is unveiled in Section 4.2. The Trainer and its underlying algorithm are presented in Section 4.3. The Idle-time and Failure Managers are described in Sections 4.6 and 4.7. Several numerical experiments used for solving problems are performed in Section 5 where we also discuss their results. Conclusions and future research directions are outlined in Section 6.

2. Related work

Developing automated problem solvers is one of the central themes of mathematics and computer science.

The source of inspiration in most of these approaches was the nature and the human brain. In this section, we will make a brief review of existing work in the field of general problem solvers and adaptive techniques.

2.1. Evolutionary adaptive models

The main engine of our system is based on an adaptive GP mechanism. This is why we start by reviewing some relevant work in this field.

In the early stages, evolutionary algorithms have been seen as problems solvers that exhibit the same performance over a wide range of problem without too much user interference [19,23]. The modern view say that there is no guarantee that an algorithm (having the same parameter settings) will perform similarly well on multiple problems [19,53]. This is why many techniques for adapting the parameters were proposed.

According to [3], adaptive evolutionary computations are distinguished by their dynamic manipulation of selected parameters or operators during the course of evolving a problem solution. Adaptive ECs have an advantage over standard ECs in that they are more reactive to the unanticipated particulars of the problem and, in some formulations, can dynamically acquire information about regularities in the problem and exploit them.

Mainly, there are two taxonomy schemes [3,19] which group adaptive computations into distinct classes—distinguishing by the type of adaptation (i.e., how the parameter is changed), and by the level of adaptation (i.e., where the changes occur).

In [3] the adaptive evolutionary computations have been divided into algorithms with absolute update rules (which compute a predetermined function over a set of generations or populations and use the changes in this heuristic to determine when and how to modify the algorithm's adaptive parameters) and empirical update rules (which use the same variation and selection process of evolving the problem solutions to also modify the adaptive parameters).

Both classes of adaptive evolutionary algorithms can be further subdivided based on the level the adaptive parameters operate on. Angeline distinguished between population, individual, and component-level adaptive parameters [3].

The classification scheme of Eiben et al. [19] has extended and broaden the concepts introduced by Angeline in [3]. Adaptation schemes are again classified firstly by the type of adaptation and secondly – as in [3]– by the level of adaptation. Considering the different levels of adaptation, a fourth level, environment level adaptation was introduced in order to take into account the cases where the responses of the environment are not static. Concerning the adaptation type, in [19] the algorithms are first divided into static (i.e., no changes of the parameters occur) and dynamic algorithms. Based on the mechanism of adaptation three subclasses are distinguished: deterministic, adaptive, and finally self-adaptive algorithms. The latter comprise the same class of algorithms as in [3].

The first proposals to adjust the control parameters of a computation automatically date back to the early days of evolutionary computation. In 1967, Reed et al. [43] have experimented with the evolution of probabilistic strategies playing a simplified poker game. Also in 1967, Rosenberg [45] has proposed to adapt crossover probabilities. Concerning genetic algorithms, Bagley [8] has considered incorporating the control parameters into the representation of an individual. Although Bagley's suggestion is one of the earliest proposals of applying classical self-adaptive methods, self-adaptation as usually used in ES appeared relatively late in genetic algorithms. In 1987, Schaffer and Morishima [47] introduced the self-adaptive punctuated crossover adapting the number and location of crossover points. Some years later, a first method to self-adapt the mutation operator was suggested by Back [7,6]. He has proposed a self-adaptive mutation rate in genetic algorithms similar to evolution strategies.

Some previous studies have investigated adaptive and self-adaptive representations in genetic programs. The adaptive representation genetic program [44] is a population-level adaptive genetic program that uses statistics gathered over all sub-trees occurring in the population to determine more advantageous crossover points and preserve high-fitness sub-trees. The genetic library builder (GLiB) [1,2] is an individual-level self-adaptive genetic program that co-evolves a hierarchical representation for each individual in the population.

Automatically defined functions (ADFs) [28] is an individual-level self-adaptive genetic program where each individual adapts its definitions for a predetermined set of subroutines. Koza and Andre [29] have extended this method to allow the number and interface of an individual's subroutines to adapt as well. Teller [52] has investigated a self-adaptive crossover scheme for a variant of genetic programming and Iba and de Garis [26] has described an adaptive crossover operation that uses a variety of absolute update

Please cite this article in press as: M. Oltean, L. Dioşan, An autonomous GP-based system for regression and classification problems, Appl. Soft Comput. J. (2008), doi:10.1016/j.asoc.2008.03.008

rules. In [4] Angeline has investigated two self-adaptive crossover operators for genetic programming inspired by the self-adaptive mutation operators for finite state machines (described in [21]): selective self-adaptive crossover (SSAC) and self-adaptive multi-crossover (SAMC).

In [24] the authors have proposed an evolutionary algorithm whose representation of solutions is changed during the search process. Each chromosome is an array of integers represented over some alphabet (numerical base). If no improvements of the solution occur for a fixed number of generations, the alphabet is changed to a random one.

The ARGOT system [46] adaptively manipulates the interpretation of the representation of a genetic algorithm based on levels of allele convergence and variance.

Bach [6] has investigated the self-adaptation of parameters for mutating fixed-length binary strings in a genetic algorithm.

Davis [15] has presented a genetic algorithm that adapts the probability of applying its various reproductive operators based on offspring performance over an adaptation window.

In the system proposed by Spears [49] an extra bit is added to each individual in the population. The percentage of these bits having a value of 1, taken over the entire population, determines the relative probability of applying uniform cross-over over single point crossover during the current reproduction cycle.

The auto-adaptive microGP [14] internally tunes both the number of consecutive random mutations and the activation probabilities of all genetic operators. The decision is taken based on the famous 1/5 rule of Rechenberg [42].

An adaptive variant of multi expression programming has been presented in [37].

PushPop system [50] has introduced auto constructive evolution, which represents a more radical form of self-adaptation. In most previous work, the algorithms for reproduction and diversification have been essentially fixed, with only the numerical parameters (such as mutation rates) subject to adaptation. By contrast, in an auto constructive evolution system the individuals in the population are entirely responsible for constructing their own offspring.

The fitness function of a GP algorithm has been also adapted during the search process. In [18] to each sample point (for a symbolic regression problem) was attached a weight. The fitness was computed as the weighted sum of the differences between the actual output and the expected output for each training data. In [20] a similar technique was applied to graph coloring problem. In this case, a weight was attached to each un-colored node.

In [30] has been proposed a novel means of automating the process of discovering successful local search strategies for EAs where the search strategies (themselves encoded as individuals of a GP population) are co-evolved alongside the population of potential solutions. The representation of local search operators can include features such as the acceptance strategy, the maximum number of neighborhood members to be sampled, the number of iterations for which the meme should be run, a decision function that will tell the meme whether it is worth or not to be applied on a particular individual and the move operator itself in which the meme will be based.

2.2. Adapting the strategy

An effective way to boost the performance is to switch between multiple algorithms depending on the current stage of the optimization process.

In [12] the melt undergoing solidification by applying an external magnetic field is controlled by an automatic procedure incorporating six popular optimization algorithms: genetic algorithm, a quasi-Newton method of Pschenichny–Danilin, modified AQ1 Nelder–Mead simplex method, sequential quadratic programming, Davidon–Fletcher–Powell gradient search algorithm, and differential evolution. Each of these algorithms provides a unique approach to optimization with varying degrees of convergence, reliability, and robustness at different cycles during the iterative optimization procedure. There are some heuristic rules to automatically switch between different optimization algorithms.

Another hybrid algorithm is presented in [13] for solving a problem that consists of a solidifying thermosolutal flow in a square cavity subjected to variable thermal and magnetic boundary conditions. Different heuristics such as Broyden–Fletcher–Goldfarb–Shanno (BFGS) quasi-Newton, the particle swarm optimization, and the differential evolution are used for this purpose. When a certain percent of the particles find a minimum, the algorithm switches automatically to the differential evolution method and the particles are forced to breed. If there is an improvement in the objective function, the algorithm returns to the particle swarm method, meaning that some other region is more likely of having a global minimum. If there is no improvement of the objective function, this can indicate that this region already contains the global value expected and the algorithm automatically switches to the BFGS method in order to find its location more precisely.

3. Extending genetic programming

The most important part of the GP-AS system is the Trainer, which is used for generating new Solvers based on some examples. Genetic programming [27,28] is used as underlying mechanism for the Trainer. The GP technique is described in this section. We have enriched the GP individuals with the ability to output multiple values.

3.1. Standard GP representation

Standard GP chromosomes are represented as trees. The maximal number of levels of a tree (the depth of a tree) is restricted to a given value. Each internal node of a tree contains a function and each leaf contains a terminal (or the argument(s) of a function).

Example. A chromosome using the function set $F = \{+, -, *\}$ and the terminal set $T = \{a, b, c, d, e\}$ as depicted in Fig. 1.

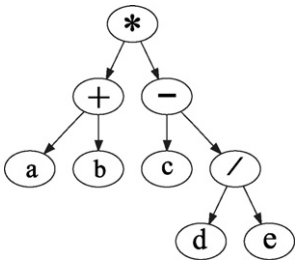

Fig. 1. A GP chromosome which encodes the mathematical expression: $(a + b)*(c - d/e)$. The output of this program is given by its root.

Please cite this article in press as: M. Oltean, L. Dioşan, An autonomous GP-based system for regression and classification problems, Appl. Soft Comput. J. (2008), doi:10.1016/j.asoc.2008.03.008

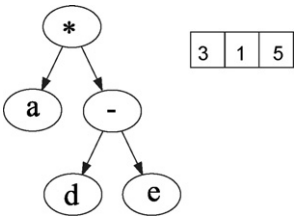

Fig. 2. GP with multiple outputs: the tree and the array of indexes providing the output. The first output is provided by the 3rd node ($d - e$), the second output by the root node ($a \times (d - e)$) and the last output by the 5th node (e).

3.2. GP with multiple outputs

Because we want to obtain a system which is able to solve problems with any number of outputs we will describe the way in which genetic programming may be efficiently adapted for dealing with problems having multiple outputs (such as designing digital circuits [31,34]).

In this paper the number of inputs is denoted by NI and the number of outputs is denoted by NO.

In standard GP [27], the root in a chromosome is chosen to provide the output of the program. Thus, when a single value is expected as output, we simply choose the root of the chromosome. When the problem has multiple outputs, we have to choose NO nodes which will provide the desired outputs. It is obvious that the genes must be distinct unless the outputs are redundant. A similar situation is encountered in the case of Cartesian genetic programming [31], where the genes providing the output of the program are evolved in the same way as all the other genes.

Each GP chromosome will also store an array of NO integer values which represent the index of the nodes providing the outputs. If the problem has only one output it will always be provided by the root of the tree. No other array of outputs will be stored in this case. The nodes are indexed starting with value 1 (the root node). The order for indexing is given by the breadth-first parsing of the tree.

Example. Let us consider a problem with three outputs and a GP chromosome with five nodes. The chromosome and the associated list of output nodes are depicted in Fig. 2.

Note that the list of nodes may require corrections after the application of genetic operators so that all indices point to the valid range of values.

This representation has the advantage that it can deal with both cases when the outputs are interconnected and cases when the outputs are truly independent one of the others. In the first case, the outputs may share the same sub-tree. In the second case, another sub-tree may provide each output (with no common parts).

3.3. Fitness assignment process

The quality of a GP individual is usually computed by using a set of fitness cases [27,28]. For instance, we can consider a problem with NI inputs: $x_1, x_2, \ldots x_{\mathrm{NI}}$ and NO outputs $f_1, f_2, \ldots f_{\mathrm{NO}}$. Each fitness case is given as a one-dimensional array of (NI + NO) values:

$$v_1^k, v_2^k, \ldots, v_{\mathrm{NI}}^k, \quad f_1^k, f_2^k, \ldots, f_{\mathrm{NO}}^k$$

where v_j^k is the value of the jth attribute, x_j, with $j = 1$, NI, in the k th fitness case and f_i^k is the ith output (or the target) with $i = 1$, NO for the k th fitness case ($k = 1, m$, where m represents the number of examples used by the GP for learning).

3.3.1. Fitness assignment for regression problems

For the regression problems, the quality of a chromosome is given by the mean absolute error (MAE) computed using the formula:

$$\mathrm{MAE} = \frac{1}{m} \times \sum_{k=1}^{m} \sum_{i=1}^{NO} |o_i^k - f_i^k|$$

where f_i^k is the expected ith output value (the value that must be predicted), o_i^k is the obtained ith output value for the kth fitness case, and m is the number of fitness cases.

When the outputs are different a multi-objective approach can be applied.

3.3.2. Fitness assignment for classification problems

Each class has associated a numerical value. This was either given as input by the user or it was obtained by preprocessing of input data (see Section 4.1).

In the case of problems with multiple outputs, each of them can have assigned different number of classes. For instance, the first output can belong to five classes; the second output can belong to three classes and so on.

The value o_i^k of each output i (in a GP chromosome), for each example k in the training set is computed. Then, each example in the training set will be classified to the nearest class (the class c_i for which the difference $o_i^k - c_i$ is minimal) for each i, $1 \leq i \leq \mathrm{NO}$.

The quality of an output is equal to the number of incorrectly classified examples in the training set over the number of examples (m). The fitness of a chromosome will be the sum of the quality for all outputs i, $1 \leq i \leq \mathrm{NO}$.

3.4. Genetic operators

Search operators mainly used within the GP algorithm are crossover and mutation. These operators preserve the chromosome structure. All the tree offspring are syntactically correct expressions. The arrays encoding the indexes for output might require some corrections. This is because the number

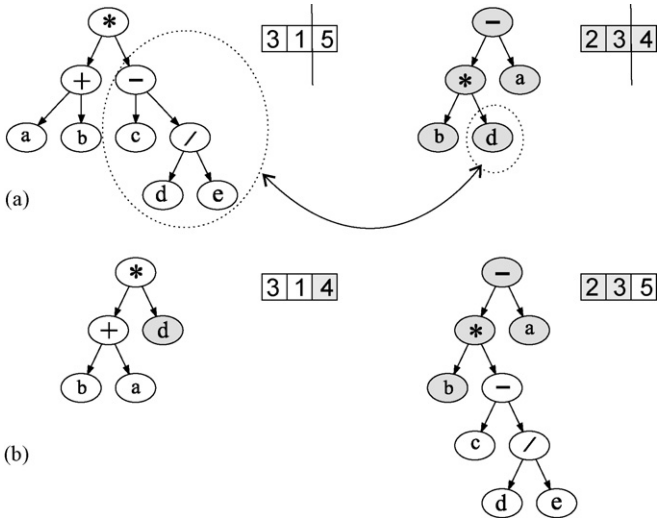

Fig. 3. GP one-cutting point crossover. Two sub-trees are exchanged between parents.

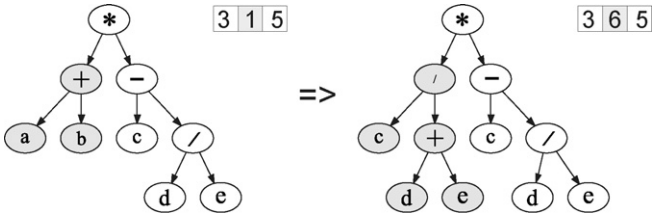

Fig. 4. GP mutation.

of nodes in a tree can decrease and some indexes may become invalid. In order to solve this problem one has to check all the positions in the array of outputs and new random values must be generated for indexes greater than the number of nodes in the associated tree.

The genetic operators are not modified during the search process. Only one type of mutation and one type of crossover are applied for all the problems.

3.4.1. Crossover

By crossover two parents are selected and recombined. For instance, within the cutting-point recombination two sub-trees (one from each parent) are exchanged.

The exchange of information is also performed for the arrays encoding the indexes of the outputs. Because these arrays have a fixed-length (equal to the number of outputs—NO), we could apply the genetic operators from binary encoding.

After crossover the array of outputs may require corrections.

Example. Let us consider the two parents C_1 and C_2 given in Fig. 3 a. The two offspring O_1 and O_2 are obtained by one cutting-point recombination as given in Fig. 3 b.

When after the crossover the depth of one or of both offspring exceeds the maximal height allowed for a GP tree, this (these) offspring is (are) not kept. Instead, the parents are stored.

3.4.2. Mutation

In order to perform a mutation operation a sub-tree of the chromosome is deleted and a new sub-tree is grown by using the same procedure like that one used during the initialization stage. The maximal depth of a tree limits the growth process.

A correction procedure (similar to that for crossover) is applied to mutation.

Example. Consider the chromosome C given in Fig. 4. For a point mutation, the filled sub-tree from the parent is deleted and a new sub-tree is growth there.

4. Detailed description of GP-autonomous solver

Our purpose is to build an autonomous system which does not rely on the human interference when solving problems. For achieving this we have to provide to the system as much information as we can. Section 4.1 describes what information the system needs and which is the correct format for sending it.

The system is organized as follows:

(i) a Decision Maker (described in Section 4.2),
(ii) a Trainer (described in Section 4.3),
(iii) a Solver Repository (described in Section 4.4),
(iv) an Idle-time Manager (described in Section 4.6) and
(v) a Failure Manager (described in Section 4.7)

When GP-Autonomous Solver (GP-AS) is run for the first time, it starts with an empty Repository of Solvers. In other words, the system does not know how to solve any problem.

When a problem is presented to the system, the Decision Maker decides which Solver will try to solve that problem. It does that by sending a request to the Repository Manager which in turn will query its database for a Solver meeting the requested criteria. If no suitable Solver is found, the Decision Maker activates the Trainer which will try to build a Solver for the problem. This new Solver is added to the Repository of existing Solvers. The training process of new Solvers is performed by using the examples (also called fitness cases or training data) requested from the user.

In order to improve the performance of the system, we could perform multiple runs to evolve a Solver and we could take the best of them. When no request is sent to the system it can try to improve the existing ones by calling the Idle-time Manager. Failure cases are identified by a Failure Manager.

The components of the GP-AS system are minutely described in the following sections of the paper.

4.1. The structure of the input

Each input of a problem is described by five pieces of information. All of them are required when a new Solver is trained and usually only four of them are needed when a Solver is called to solve a problem (the output is not required in this case). The system will be able to extract the type of each variable from the input.

(i) The number of variables (features or inputs) of the information to be analyzed.
(ii) The number of elements in the output vector. Some problems might have only one output but there are a lot of problems which have multiple outputs (see for instance the problem of designing digital circuits [31,34] or the *Building* problem [41]).
(iii) In the case that there are multiple problems which have the same input we have to store some supplementary information in them. For instance both even-3-parity and odd-3-parity problems [27] have the same input (i.e. binary strings of length 3), but their output is quite different. This is why we need more information for distinguishing them. For a human being the identifiers "odd-parity" and "even-parity" are enough. This is why we use a similar mechanism for our system. Problems having the same input will be discriminated by an identifier (a string of chars).
(iv) An array of values representing the inputs.
(v) The array of values representing the output. This information is required only when a new Solver is trained.

Each variable/output can have one of the following types: *Boolean* (given as 0 or 1), *real* (with dot separating integer from fractional part), *integer* and *string* (delimited by "").

The system automatically converts all inputs/outputs to the same type by using the following rules:

(i) Strings are automatically converted to integers by performing a preprocessing step: the set of all strings (corresponding to a feature) is extracted and each string is replaced by its index in that set. If we have only two strings they are converted to Boolean values.

Table 1
The pairs of types which are allowed to appear in our system

Input	Output	Applications
Boolean	Boolean	Boolean function finding
Boolean	Integer	Regression and classification
Boolean	Real	Regression
Real	Boolean	Classification
Real	Integer	Classification or regression with integer values only
Real	Real	Symbolic regression
Integer	Boolean	Classification
Integer	Integer	Classification or regression with integer values only
Integer	Real	Regression

The first column represents the type of the input. The second column represents the type of the output. The third column contains a problem where the corresponding type of input/output could appear. Strings were omitted from this table because they are converted to integers and interpreted in this way.

(ii) If one input/output is *real* all other inputs/outputs are converted to real values.
(iii) If one input/output is *integer* and none is *real* all other inputs/outputs are converted to integer values.

The possible applications for each pair (input, output) are shown in Table 1.

Some examples of the input presented to the system are given below:

Example 5. An example of input for the even-3-parity problem (used for training a new Solver) is given below:

```
3                       // the number of inputs
1                       // the number of outputs
even-parity             // problem identifier
0, 0, 0, 1              // input and output arrays
0, 0, 1, 0
0, 1, 0, 0
0, 1, 1, 1
1, 0, 0, 0
1, 0, 1, 1
1, 1, 0, 1
1, 1, 1, 0
```

Example 6. When we want to find the solution for this problem (assuming that the corresponding Solver already exists) we do not have to provide the output anymore. Such an example is given below (only for one fitness case we are interested in finding a solution):

```
3
1
even-parity
0, 1, 0
```

Example 7. If the user is not very sure whether there is a Solver for the problem or he does not know the correct identifier he will also have to provide some examples. This means that some fitness cases have the output specified (and they are used for locating the good Solver), and the other cases have no output specified (for these cases a solution is expected if the good Solver is found). In this case the corresponding problem identifier may be skipped. An example is given below:

Table 2
The function sets for different data types

Data type	Function set
Binary	F_{Binary} contains all the operators accepting two inputs and providing one output
Integer	$F_{\text{Int}} = \{+, -, \times, \%, /\}$, % provides the modulo of the result and / performs the integer division. These two operators are protected against division by zero by returning the value 1 in that cases
Real	$F_{\text{Real}} = \{+, -, \times, /\}$, / is protected against division by zero

```
3
1
0, 0, 0          // the input for which the user expects an output
1, 0, 0, 0       // some examples
1, 0, 1, 1
1, 1, 0, 1
1, 1, 1, 0
```

4.2. The Decision Maker

The Decision Maker (DM) is the component that decides whether the system knows how to solve a particular problem. There are two ways in which the decision may be taken:

- by looking at the structure of the input. For instance, if the input consists of three binary digits, then the DM should call the corresponding Solver which knows how to handle problems with three binary inputs. If the input is an array of real values, then the DM should call a Solver, which is able to solve problems whose input is of that type.
- by having some training examples. The following steps are applied:
  - [
  - bullet] [S.] The DM finds all the available Solvers that match the data type, the size of the input and the size of the output.
  - [
  - bullet] [S.] Each of these Solvers is applied in order to solve each example provided by the user.
  - [
  - bullet] [S.] If the worst error generated by applying a Solver to each example is lower than or equal to the error stored internally by that Solver (see Section 4.4), then the system decides that the Solver is good enough for solving that problem. If multiple Solvers are equally good for a particular problem, the decision on which to use is randomly taken.

The first method is preferred, but the second one is applied in cases when there are not enough data for the first one.

Discipulus [9] software from AimLearning[1] has a similar feature: it decides the type of the problem: regression or classification according to the input data.

4.3. The Trainer—Constructing Problem Solvers

When the Decision Maker is not able to find a suitable Solver for a particular problem, it calls the Trainer. This component will try to build a Solver for the problem that is being solved. Since the trainer

[1] http://www.aimlearning.com.

Please cite this article in press as: M. Oltean, L. Dioşan, An autonomous GP-based system for regression and classification problems, Appl. Soft Comput. J. (2008), doi:10.1016/j.asoc.2008.03.008

is GP-based, it needs some training examples. The user usually supplies these examples.

There are two basic requirements for standard GP. As standard GP does, the Trainer should also compute the size of the Solvers (the number of nodes from a GP chromosome). In addition, the Trainer must find a solution within a reasonable amount of time. A big population and a small number of training data or a small population and a large real-world [41] training set are two opposite examples of inefficient behavior.

In order to meet these two criteria we have chosen to implement an adaptive algorithm for the Trainer. This algorithm will try to find a good population size and a good chromosome depth for the problem that is being solved.

4.3.1. Function set

The function set used for evolving Solvers depends on the type of data involved (inputs and outputs) in the problem (see Table 2):

- F_{Binary} as it was suggested in [40].
- The integer function set F_{Int} is used for integer inputs.
- The real function set F_{Real} is used for real values of the inputs.

The decision on which function set to use is usually made based on the inputs of the problem. In the current stage of the system, the function set is kept fixed during the search process. This is somehow inefficient because a large function set (such that are used for Boolean problems) means a large search space. In the near future we plan to use an adaptive strategy for searching the best decision. In this case, we will encode (into each GP tree) a set of functions, which are allowed to be used by that chromosome.

4.3.2. Terminal set

Problem inputs are used as terminals for generating new solvers. It is possible to add constants if necessarily [27].

4.3.3. The algorithm

The modified GP algorithm will start with a small population size and a small value for the maximal chromosome depth. These values will be increased as the search process advances. Note that this is not a true adaptation (as in [19,25,48]) because the population size never decreases. We have chosen not to reduce the population size because Genetic Programming techniques usually require large populations [27] for solving problems. However, we do plan to investigate in the near future the effect of true adaptation in our system.[2]

From experimental trials, we have deduced some good values for these parameters (the number of the individuals added to the current population and the number of genes added to the current chromosome). However, the parameter settings are really problem dependent and a general strategy cannot be devised.

Another problem is that different strategies for changing the population size and the number of genes in a chromosome will affect the number of generations required to find a good solution to the problem that is being solved.

After many trials we have devised the following strategy:

- The algorithm starts with a population of 1 GP individual which has one node (the tree depth is 1). The choice of these parameters (the initial population size and the initial chromosome length) is problem independent.
- The size of the population should doubled and the chromosome maximal depth should increase with 1 if no improvements of the best individual occur for $A = 10$ generations. The newly inserted individuals are randomly generated. The size of the trees may also increase due to the effect of genetic operators (see Section 3.4).
- The algorithm will stop when there have been no improvements of the best individual in the population for $B = 30$ consecutive generations.

The detailed pseudo-code is given in Algorithm 1.

Algorithm 1. The Trainer's algorithm

```
PopSize = 1;
MaxDepth = 1;
Chromosome = CreateChromosome(MaxDepth);
Add(Chromosome, Population)
NoImprovementsCount = 0;
previousBest = Chromosome;
while (NoImprovementsCount < B) do
  for i = 1 to PopSize do
    Parent1 = Selection(Population);
    Parent2 = Selection(Population);
    Offspring = Crossover(Parent1, Parent2);
    Offspring* = Mutate(Offspring);
    if (Offspring* is better than Worst(Population)) then
      Replace(Offspring*, Worst(Population));
    end if
  end for
  if (Best(Population) == previousBest) then
    NoImprovementsCount++;
  else
    previousBest = Best(Population);
    NoImprovementsCount = 0;
  end if
  if (NoImprovementsCount % A == 0) then
    MaxDepth++;
    for i = PopSize + 1 to 2 × PopSize do
      Chromosome = CreateChromosome(MaxDepth);
      Add(Chromosome, Population)
    end for
    PopSize = PopSize × 2;
  end if
end while
```

The values A and B should be carefully chosen because the behavior of the algorithm depends on them largely. An inap-

Table 3
Several examples of inappropriate behavior of the algorithm if the parameters are not correctly set

Case	Possible result
A is too small	The population size and the tree height will increase too much
A is too big	The population might loose its diversity before a new increase takes place
B is too small	The algorithm might stop too soon
B is too big	The population size and the tree height might increase too much

A is the number of consecutive generations (with no improvements of the best individual) before we increase of the population size and of the maximal height of the tree. B is the number of consecutive generations (with no improvements of the best individual) after which the algorithm is stopped.

[2] Note that the size of the GP chromosomes can increase or decrease due to the effect of genetic operators.

propriate choice of these values can lead to very poor results (see Table 3 for a possible list of unwanted behaviors).

4.3.4. Reasons for search stagnation

There are several reasons for the algorithm not is able to improve the best solution for several generations:

- It has already found the best solution possible for the problem.
- Not enough generations have been run.
- The population size and/or the maximal tree size are too small.

The last case means that we have to increase either the population size or the maximal tree depth. We will increase the size of both, since we do not know which should be increased.

4.3.5. Steady-state implications

We have chosen to double the population size mainly because we are using a steady-state GP algorithm (see Section 3). This algorithm will rapidly eliminate the worst individuals. When we randomly generate an individual to add it to the population, it is very likely that the quality of this individual is very poor. Because of that, it has big chances to be eliminated in the next few generations. Thus, we have to increase the population with a consistent number of individuals (preferably comparable to the current value for the population size). This will increase the chances for some individuals to be kept in the new generation. Some of them will actively participate in the search process.

4.4. The Solver Repository

The created solutions for solving various problems are kept in a special place called Solver Repository (SR) permanently stored on HDD. The SR is populated by Problem Solvers. SR has a manager which efficiently manages the solvers.

4.4.1. The structure of Problem Solvers

The Problem Solvers are some simple GP individuals (computer programs) (see Section 3.1) evolved by Trainer. In the current version of the GP-AS system, each Solver (GP individual) has a fixed structure and is able to solve only one problem. The function set used by each Solver depends on the type of input for the problem being solved. The number of variables is equal to the number of problem inputs.

Internally, a Solver also stores the following information (which is used by the Decision Maker (see Section 4.2)):

- Data type—integer.
- Number of inputs (variables)—integer.
- Number of outputs—integer.
- Problem identifier—string.
- The worst error for a fitness case from a particular run—double.
- The random seed used for last run. This is useful when the Trainer should be run multiple times (see Section 4.6)—integer.
- Training data – optional – only when the training error is too high.

4.4.2. Repository Manager

Since GP-AS is run on a computer with limited resources we have to take care of this aspect too. For this purpose a special component called Repository Manager (RM) has been designed. This manager will perform the following tasks:

- Add new Solvers as soon as the Trainer has created them.
- Keeps the most frequently utilized solvers in RAM memory for fast access.
- Query the Repository for solvers meeting some criteria as requested by the Decision Maker.

4.5. The structure of the output

When a problem has been solved (either by creating a new Solver or reusing an existing one) the result will be displayed in the same format as the input (see Section 4.1).

4.6. The Idle-time Manager

In some cases the solution obtained by the first run of the Trainer is of very poor quality. This can happen usually when the fitness landscape is very rugged. Since we use evolutionary algorithms for generating solvers is likely to obtain different solutions in different runs. However, since we want to obtain (very fast) a solution to a problem we will not perform multiple runs when the problem is first presented to the system. Instead, a special component called Idle-time Manager (IM) will be activated when there is no other request from user. IM will randomly pick a problem and it will run the Trainer with a different seed. Note that this is possible only if the training data have been stored along in Repository. If the new solution is better than the existing one it will be added to Repository and the old one will be removed.

IM will stop its activity when the user sends a new request to the system.

4.7. The Failure Manager

An autonomous system should be capable to run for an extended period of time without failures.

There are several cases when the GP-AS can fail. These cases along with the actions taken by the Failure Manager (FM) are described in Table 4.

5. Numerical experiments

We perform some numerical experiments for creating solvers. The tested problems are:

- Boolean function finding,
- symbolic regression,
- classification.

Regression and classification problems actually refer to several real-world problems taken from PROBEN1 [41](which have been adapted from UCI Machine Learning Repository [54]). Linear GP has been previously used [10] in order to solve problems from this set. For all the problems, we deal with one output only. Therefore, the root of the GP tree provides the potential solution for a problem.

Since it uses a deterministic algorithm, the Decision Maker proves to be able to take the expected decision in all the cases. We shall no longer focus on this aspect. The difficult task is performed

Table 4
Cases when failure manager prevents the failure of the system

Case	Prevention from FM
Power failure	The Solvers are stored on HDD. Before running the Trainer all input data are stored on a tmp file which can be later recovered
Arithmetic exceptions	The Trainer is endowed with an exception handling mechanism
Not enough memory for storing the Solvers	Delete old, unused Solvers from the Solver Repository

Table 5
Parameter settings for all experiments performed with GP-AS

Parameter	Value
Population size at the beginning	1 GP individual
Maximal tree height at the beginning	1 level (a node)
Number of generations without improvements before the population size is doubled and the maximal tree height is increased by 1 (A)	10
Number of generations without improvements before the training algorithm is stopped (B)	30
Mutation	point mutation
Crossover	cutting point
Selection	Binary tournament.

by the Trainer, which has to evolve good Solvers for the considered problems. In what follows, we shall focus on the results obtained by the Trainer.

In order to compute some performance metrics we perform more independent runs (when training new Solvers). However, in the standard variant of the GP-AS system, only one run is performed and the obtained Solver is added to the set of existing Solvers. We could extend our program to the performance of multiple trails and the choice of the best, but we are not interested in this aspect at the current stage of the project. If the user is not satisfied with a particular Solver, it can ask for multiple runs of the Trainer.

Some general parameters for Trainer are given in Table 5.

Several statistics were computed for all these experiments:

- Success rate—the proportion of the successful runs in the total number of runs.

$$\text{SR} = \frac{\text{of successful runs}}{\text{of all runs}} \quad (1)$$

- Average evaluations—we counted how many times we have called the fitness function in each run and than we averaged these values over all the runs.
- Average generations—the mean of the number of generations before the Trainer has stopped in each run.
- Average tree depth—we average the depth of all the chromosomes from the GP population in each generation and in each run.

$$\text{ATD} = \frac{\sum_{k=1}^{\text{NoRuns}} \left(\sum_{j=1}^{\text{NoGener}} \left(\sum_{i=1}^{\text{Pop\_Size}} \text{DepthChromo}_i / \text{Pop\_Size} \right) / \text{NoGener} \right)}{\text{NoRuns}} \quad (2)$$

- Average population size—the mean of the population size (the population size from the last generation) in each run.
- Average fitness of the best chromosome—the sum of the best chromosome fitness from the last generation in each runs over the number of runs.

In order to determine whether the differences between the GP-AS and standard GP are statistically significant, we use a *t*-test with a 0.05 level of significance. Before applying the *t*-test, an *F*-test is used for determining whether the compared data have the same variance.

As a comparison, we perform some experiments (for all the problems) by using a classical GP algorithm [27]. Note that this comparison is not fair because in our GP-AS system the population size and the chromosome depth are adapted during the search process. These comparison drawbacks can be reduced by performing the same number of fitness function evaluations.

5.1. Boolean function finding

Several numerical experiments are performed for the well-known even-*n*-parity problems. We have a string of *n* binary values. If an even number of positions are 1, then the output should be 1; otherwise, the output should be 0 [27]:

- The even-3-parity problem has three Boolean inputs, one Boolean output and $2^3 = 8$ training examples.
- The even-4-parity problem has four Boolean inputs, one Boolean output and $2^4 = 16$ training examples.
- The even-5-parity problem has five Boolean inputs, one Boolean output and $2^5 = 32$ training examples.

All those data are used for training. For these problem we use the all 16 Boolean function set.

For these three problems the Trainer of GP-AS system starts with a population composed by an individual whose initial height was 1. The limits used for increasing the population size and the chromosome depth are: $A = 10$ and $B = 30$.

The values from Table 6 indicates the performance of our Trainer. We can observe that for the even-3-parity problem we obtain 89 successful runs (out of 100). This means that in 89 runs (out of 100) the Trainer is able to evolve a perfect Solver for this problem.

As a comparison, we performs some experiments with a standard GP [27] algorithm containing 100 individuals. The initial chromosome depth is set to 3–5 for even-3-parity, even-4-parity and, respectively, even-5-parity problem. The success rates obtained with this GP algorithm are presented in Table 6 also.

Note that the Trainer has performed more functions evaluations than the standard GP.

5.2. Symbolic regression problems

13 test regression problems are chosen for these experiments, each problem having a single output and one or more inputs:

T_1 *Quadratic polynomial.* Find a function that best satisfies a set of fitness cases generated by the quartic polynomial [27] function:

$$f(x) = x^4 + x^3 + x^2 + x.$$

T_2 *Bank.* Predict the fraction of bank customers who leave the bank because of full queues [16].

Table 6
Results (averaged over 1000 runs) obtained for even-parity problems

	GP-AS			GP		
	Success rate (%)	Average evaluations	Average chromosome depth	Success rate (%)	Average evaluations	Average chromosome depth
Even-3	89	22580.60	4.57	100	2066.14	1.97
Even-4	50	33487.82	4.89	73	15913.81	2.84
Even-5	3	33682.34	3.29	27	25362.30	3.27

We give the average number of fitness evaluations because standard GP performs less fitness evaluations. This is because standard GP starts with larger trees then the GP-AS

Please cite this article in press as: M. Oltean, L. Dioşan, An autonomous GP-based system for regression and classification problems, Appl. Soft Comput. J. (2008), doi:10.1016/j.asoc.2008.03.008

Table 7
The characteristics of the test problems

Problem		# Instances	# Attributes	Type of attributes
T_1	Quartic	100	1	All Real
T_2	Bank	8192	8	All Real
T_3	Puma	8192	8	All Real
T_4	Kin	4096	8	All Real
T_5	Add	8192	10	All Real
T_6	CpuSmall	4096	12	All Real
T_7	Boston	506	13	All Real
T_8	Building-electric	2104	14	All Real
T_9	Cpu	4096	21	All Real
T_{10}	Flare	533	24	All Real
T_{11}	Heartac	460	35	All Real
T_{12}	Ticdata	5000	85	All Real

T_3 *Puma.* This is a family of datasets synthetically generated from a realistic simulation of the dynamics of a Unimation Puma 560 robot arm [16]. The task is to predict the angular acceleration of one of the robot arm's links. The inputs include angular positions, velocities and torques of the robot arm.

T_4 *Kin.* This is a family of datasets synthetically generated from a realistic simulation of the forward kinematics of an eight link all-revolute robot arm [16]. The task is to predict the distance of the end-effector from a target. The inputs are things like joint positions, twist angles, etc.

T_5 *Add.* A synthetic function suggested by Jerome Friedman in his "Multivariate Adaptive Regression Splines" paper [22]. The function is:

$$f(x_1, \ldots, x_{10}) = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + n,$$

where n is a random noise (a normally distributed one-dimensional random number with mean 0 and standard deviation 1). The inputs $x_1, \ldots x_{10}$ are sampled independently from a $[0, 1]$ uniform distribution.

T_6 *CpuSmall.* The purpose of this problem is to predict a computer system activity from system performance measures by using a restricted number of attributes (excluding the paging information) [16].

T_7 *Boston.* Prediction of the housing values [54].

T_8 *Building-electric.* Predict the electric power consumption in a building [41].

T_9 *Cpu.* The purpose of this problem is to predict a computer system activity from system performance measures by using all attributes [16]. This is a generalization of problem T_7.

Table 9
The characteristics of the test problems

Problem		# Instances	# Attributes	Type of attributes
T_{13}	Diabets	400	8	All Real
T_{14}	Cancer1	350	9	All Real
T_{15}	Cancer2	683	9	All Real
T_{16}	Glass	214	9	All Real
T_{17}	Twonorm	5000	20	All Real
T_{18}	Thyroid	4000	21	All Real
T_{19}	Hearta	460	35	All Real
T_{20}	Horse	364	58	All Real
T_{21}	Mushroom	1000	125	All Binary

T_{10} *Flare.* Guess the number of solar areas (of small, medium, and large size) that will happen during the next 24-h period in a fixed active region of the sun surface [41].

T_{11} *Heartac.* The purpose of this problem is to predict heart disease [41].

T_{12} *Ticdata.* This data set used in the CoIL 2000 Challenge contains information on customers of an insurance company [54].

The main characteristics of the test datasets used in the experiments are presented in Table 7.

The set of function symbols used in regression problems case is F_{Real}. The terminal set consists of the problem inputs (see Table 7).

Standard GP population size is manually set to 100 individuals. Maximal tree depth for the initial population is set to 4. The number of generations is not pre-specified. Instead, we perform the same number of function evaluations as the GP-AS. For achieving this goal, we run GP-AS first based on that we have already computed: the number of generations for the standard GP. The same strategy applies for classification problems (see next section).

In Table 8 we present the results obtained by the Trainer and by the classical GP algorithm for all the symbolic regression problems. We can observe that both algorithms find (in average) good solutions.

In all cases (but one) standard GP performs better than GP-AS. Note again that the parameters of standard GP are manually set, while the GP-AS discovers by itself some good values for the population size and for the tree height.

The *P*-values of a two-tailed *t*-test with nine degrees of freedom are given in Table 8. These values show that the differences between the results obtained by GP-AS and standard GP are statistically significant in 50% of cases.

Table 8
Results (average over 10 runs) obtained for symbolic regression problems

	GP-AS		Standard GP		*F*-test	*t*-Test	Average evaluations
	AvgBest	S.D.	AvgBest	S.D.			
T_1	0.68206	3.89791	0.00001	0.00010	0.00000	0.08325	26848.83
T_2	0.03524	0.00848	0.02685	0.00190	0.00000	0.01818	25183.86
T_3	3.30141	0.10965	3.06285	0.11444	0.57529	0.00632	53904.20
T_4	0.37759	0.14602	0.28223	0.00958	0.00000	0.20955	34509.60
T_5	2.28124	0.09754	2.03422	0.03112	0.16652	0.03040	153393.67
T_6	46.65821	1.11648	37.05010	9.63054	0.17274	0.17978	95535.33
T_7	5.75115	1.43830	4.01619	0.55457	0.05997	0.07244	211651.60
T_8	0.07404	0.00732	0.09992	0.01404	0.22119	0.24970	140769.8
T_9	30.29716	6.47158	23.64011	1.11441	0.00403	0.11073	87997.80
T_{10}	0.03803	0.00005	0.03673	0.00052	0.06784	0.50000	20708.40
T_{11}	0.31100	0.28174	0.22605	0.00734	0.00000	0.53020	20024.00
T_{12}	11.69360	13.83271	0.05805	0.00012	0.00001	0.44502	20004.50

AvgBest means that we have averaged the best solutions in each run (over all runs).

Please cite this article in press as: M. Oltean, L. Dioşan, An autonomous GP-based system for regression and classification problems, Appl. Soft Comput. J. (2008), doi:10.1016/j.asoc.2008.03.008

Table 10
Results (average over 10 runs) obtained for classification problems

	GP-AS		Standard GP		*F*-test	*t*-Test	Average evaluations
	AvgBest	S.D.	AvgBest	S.D.			
T_{13}	129.4000	0.5477	89.8000	1.3038	0.1215	0.0000	23,859
T_{14}	26.6000	5.3198	19.4000	8.6487	0.3693	0.0844	100,422
T_{15}	58.4000	18.6360	61.0000	12.6886	0.4748	0.8544	93,685
T_{16}	69.0000	0.0000	48.2000	5.1672	0.0000	0.0008	21,908
T_{17}	2991.0000	1122.5061	2470.4000	10.2859	0.0000	0.3599	30150
T_{18}	101.0000	0.0000	75.2000	10.7331	0.0000	0.0058	20,993
T_{19}	107.7647	7.5955	104.5882	6.0628	0.9976	0.4410	85,241
T_{20}	97.0000	4.3970	107.2000	7.5542	0.3718	0.1041	158,842
T_{21}	73.0000	37.2022	21.6000	18.3981	0.1494	0.0430	86,344

AvgBest means that we have averaged the best solutions in each run (over all runs).

5.3. Classification problems

Nine classification problems are used in these experiments:

- T_{13} *Diabets*. Diagnose diabetes of Pima Indians [41].
- T_{14} *Cancer1*. Classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination [41].
- T_{15} *Cancer2*. Breast cancer classification into benign and malignant classes [54].
- T_{16} *Glass*. The aim of this problem is to classify glass types [41].
- T_{17} *Twonorm*. Leo Breiman's two normal example [16]. Classify a case as coming from one of two normal distributions. Leo Breiman's twonorm example [11] is a 20-dimensional, 2 class classification example. Each class is drawn from a multivariate normal distribution with unit variance.
- T_{18} *Thyrod*. Diagnose thyroid hyper- or hypofunction based on patient query data and patient examination data [41].
- T_{19} *Hearta*. Predict heart disease [41]. The purpose is to decide whether at least one of four major vessels is reduced in diameter by more than 50%.
- T_{20} *Horse*. Predict the fate of a horse that has a colic [41] based on the results of a veterinary examination.
- T_{21} *Mushroom*. The aim of this problem is to discriminate edible from poisonous mushrooms [41]. The decision is made based on a description of the mushroom's shape, color, odor, and habitat.

The main characteristics of the test datasets used in the experiments are presented in Table 9.

The set of function symbols used in these experiments is F_{Real}. The terminal set consists of the problem inputs (see Table 9).

Again we compare the solutions found by Trainer with those obtained by the standard GP algorithm having a population of 100 individuals (see Table 10). In general, the standard GP algorithm performs better than the Trainer, but only for two problems (T_{15} and T_{19}) the Trainer was able to evolve a better solution than the GP algorithm.

The *P*-values of a two-tailed *t*-test with nine degrees of freedom are given in Table 10. These values show that the differences between the results obtained by GP-AS and standard GP are statistically significant in three (out of nine) cases.

Table 11
Results (average over five runs) obtained for building problem

	AvgBest	S.D.	Average evaluations
All	0.113	0.048	105901.6
1st out	0.074	0.007	140769.8
2nd out	0.046	0.005	118444.4
3rd out	0.084	0.032	57763.8

All is the result when all outputs are taken into account. The other results are obtained when each output is considered separately and the other outputs are ignored.

5.4. Symbolic regression with multiple outputs

GP-AS is able to solve problems with multiple outputs. In this section we focus our attention on real-world problem with multiple outputs. The involved problem is the *Building* problem where the one has to predict the energy consumption in a building (electric power, hot water, cold water) [41].

A simple variant (with a single output) of it was solved in Section 5.2. Here (see Table 11) we show the results for the case where the complete problem (with all three original outputs) was taken into account.

6. Conclusions and further work

A system called GP-AS has been investigated in this paper. GP-AS has six main components: a Decision Maker, a Trainer, a Repository of Problem Solvers, a Repository Manager, an Idle-Time Manager and a Failure Manager. The decision on whether the system knows how to solve a problem or not is taken by the Decision Maker. The Trainer – which is the most important part of the system – is based on an adaptive variant of Genetic Programming. Each problem has its own Solver (a GP program) which is clearly defined by several parameters.

At its current stage, the GP-AS system is able to handle any kind of regression and classification problem. If it knows how to solve it, the system calls a special Solver for that problem. If it does not, the Trainer will be able to generate a new Solver for that problem. The Trainer algorithm is problem independent, being able to construct solutions based on training sets of variable size. No other human interference is required in this part of the system. When no requests are sent to the system it is able to improve its existing solutions.

Numerical experiments have shown that GP-AS was able to handle a wide range of problems yielding good results.

Further work directions will be focused on:

- Extending the system so that it could be able to deal with problems from other interesting classes.
- Replacing Solvers by heuristics. This will help the system to solve more problems with a single heuristic. For instance current Solvers could be replaced by evolutionary algorithms. Several methods for generating evolutionary algorithms have been proposed in [35,36].

Uncited references

Q1

[32,33,38,39,51].

Acknowledgments

The authors thank to anonymous reviewers for their useful suggestions. This research was supported by grant IDEI-543 from CNCSIS.

References

[1] P.J. Angeline, J.B. Pollack, The evolutionary induction of subroutines, in: Proceedings of the 14th Annual Conference of the Cognitive Science Society, Lawrence Erlbaum Associates, Hillsdale, NJ, (1992), pp. 236–241.
[2] P.J. Angeline, J.B. Pollack, Coevolving high-level representations, in: C.G. Langton (Ed.), Artificial Life III, Addison-Wesley, Reading, MA, 1994, pp. 55–71.
[3] P.J. Angeline, Adaptive and self-adaptive evolutionary computations, in: M. Palaniswami, Y. Attikiouzel (Eds.), Computational Intelligence: A Dynamic Systems Perspective, IEEE Press, 1995, pp. 152–163.
[4] P.J. Angeline, Two self-adaptive crossover operators for genetic programming, in: Advances in Genetic Programming 2, MIT Press, 1996, pp. 89-110.
[5] P.J. Antsaklis, K.M. Passino, S.J. Wang, Towards intelligent autonomous control systems: architecture and fundamental issues, J. Intell. Robot. Syst. 1 (1989) 315–342.
[6] T. Back, Self-adaptation in genetic algorithms, in: F.J. Varela, P. Bourgine (Eds.), Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, MIT Press, (1992), pp. 263–271.
[7] T. Back, The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm, in: R. Manner, B. Manderick (Eds.), Parallel Problem Solving from Nature, vol. 2, North Holland, Amsterdam, 1992, pp. 85–94.
[8] J.D. Bagley, The behavior of adaptive systems which employ genetic and correlation algorithms, PhD Thesis, University of Michigan, 1967.
[9] W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, Genetic Programming—An Introduction; On the Automatic Evolution of Computer Programs and its Applications, 3rd edition, Morgan Kaufmann, Verlag, 2001.
[10] M. Brameier, W. [illegible]f, A comparison of linear genetic programming and neural networks in medical data mining, IEEE Trans. Evol. Comput. 5 (1) (2001) 17–26.
[11] L. Breiman, Bias, variance, and arcing classifiers, Technical Report 460, Statistics Department, Berkeley, 1996.
[12] M.J. Colaco, G.S. Dulikravich, T.J. Martin, Control of unsteady solidification via optimized magnetic fields, Mater. Manuf. Process. 20 (3) (2005) 435–458.
[13] M.J. Colaco, G.S. Dulikravich, Solidification of double-diffusive flows using thermo magneto-hydrodynamics and optimization, Mater. Manuf. Process.,in press.
[14] F. Corno, G. Squillero, Exploiting auto-adaptive micro gp for highly effective test programs generation, in: The 5th International Conference on Evolvable Systems: From Biology to Hardware, 2003, 262–273.
[15] L. Davis, Adapting probabilities in genetic algorithms, in: J.J. Grefenstette (Ed.), Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum, Hillsdale, NJ, 1989, pp. 61–69.
[16] C.E. Rasmussen, R.M. Neal, G. Hinton, D. van Camp, M. Revow, Z. Ghahramani, K. Kustra, R. Tibshirani, Data for evaluating learning in valid experiments, 1996.
[17] D. Edelman, A comparative study of neural network, genetic programming, and support-vector machine methods in forecasting financial time series, in: International Conference on Statistics, Combinatorics and Related Areas, 2001.
[18] J. Eggermont, J.I. van Hemert, Adaptive genetic programming applied to new and existing simple regression problems, genetic programming, in: Proceedings of EuroGP'2001, LNCS 2038, 2001, pp. 23–35.
[19] A.E. Eiben, R. Hinterding, Z. Michalewicz, Parameter control in evolutionary algorithms, IEEE Trans. Evol. Comput. 3 (2) (1999) 124–141.
[20] A.E. Eiben, J.I. van Hemert, SAW-ing EAs: adapting the fitness function for solving constrained problems, in: D. Corne, M. Dorigo, F. Glover (Eds.), New Ideas in Optimization, McGraw-Hill, London, 1999 , pp. 389–402 (Chapter 26).
[21] L.J. Fogel, D.B. Fogel, P.J. Angeline, A preliminary investigation on extending evolutionary programming to include self-adaptation on finite state machines, Informatica 18 (1994) 387–398.
[22] J.H. Friedman, Multivariate adaptive regression splines, Ann. Stat. 19 (1) (1991) 1–141.
[23] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA, 1989.
[24] C. Grosan, M. Oltean, Adaptive representation for single objective optimization, Soft Comput. 9 (8) (2005) 594–605.
[25] R. Hinterding, Z. Michalewicz, A.E. Eiben, Adaptation in evolutionary computation: a survey, in: Proceedings of the 4th IEEE International Conference on Evolutionary Computation, 1997, pp. 65–69.
[26] H. Iba, H. de Garis, Extending genetic programming with recombinative guidance, in: P. Angeline, K. Kinnear (Eds.), Advances in Genetic Programming, vol. 2, 1996.
[27] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, 1992.
[28] J.R. Koza, Genetic Programming II: Automatic Discovery of Reusable Subprograms, MIT Press, Cambridge, MA, 1994.
[29] J.R. Koza, D. Andre, Evolution of both the architecture and the sequence of work-performing steps of a computer program using genetic programming with architecture-altering operations, in: P. Angeline, K. Kinnear (Eds.), Advances in Genetic Programming, vol. 2, 1996.
[30] N. Krasnogor, S. Gustafson, Towards truly "memetic" memetic algorithms, in: D. Corne, G. Fogel, W. Hart, J. Knowles, N. Krasnogor, R. Roy, J.E. Smith, A. Tiwari (Eds.), The Third Workshop on Memetic Algorithms, The 7th Parallel Problem Solving from Nature Conference, Granada, Spain, (2002), pp. [illegible]
[31] J.F. Miller, D. Job, V.K. Vassilev, Principles in the Evolutionary Design of Digital Circuits. Part I. Genetic Programming and Evolvable Machines, vol. 1(1), Kluwer Academic Publishers, 2000, pp. 7–35.
[32] M. Oltean, C. Groşan, Evolving evolutionary algorithms by using multi expression programming, in: European Conference on Artificial Life, LNAI 2801, Springer-Verlag, (2003), pp. 651–658.
[33] M. Oltean, C. Groşan, A comparison of several linear genetic programming tech[illegible]s, Complex-Systems 14(4) (2003) 282–311.
[34] M. [illegible] C. Groşan, Evolving digital circuits using multi expression programming, in: R. Zebulum (Ed.), NASA/DoD Conference on Evolvable Hardware, June 24–25, Seatle, IEEE Press, NJ, (2004), pp. 87–90.
[35] M. Oltean, Evolving evolutionary algorithms using linear genetic programming, in: Evolutionary Computation, vol. 13(3), MIT Press, MA, USA, 2005, pp. 387–410.
[36] M. Oltean, Evolving evolutionary algorithms with patterns, Soft Comput. 11 (6) (2007) 503–518.
[37] M. Oltean, A-brain: a general system for solving data analysis problems, J. Exp. Theor. Artif. Intell. 19 (4) (2007) 333–353.
[38] M. O'Neill, C. Ryan, IEEE Trans. Evol. Comput. 5 (4) (2001) 349–358.
[39] F. Pett[illegible], N. Chakraborti, H. Saxen, A genetic algorithms based multi-objective [illegible]al net applied to noisy blast furnace data, Appl. Soft Comput. 7 (1) (2007) 3[illegible]7.
[40] R. Poli, J. Page, Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine-code GP and demes, in: Genetic Programming and Evolvable Machines, vol. 1, 2000, 37–56.
[41] L. Prechelt, PROBEN1—a set of neural network problems and benchmarking rules, Technical Report 21, University of Karlsruhe, 1994.
[42] I. Rechenberg, Evolutions strategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution, Frommann-Holzboog Verlag, Stuttgart, 1973.
[43] J. Reed, R. Toombs, N.A. Barricelli, Simulation of biological evolution and machine learning I. selection of self-reproducing numeric patterns by data processing machines, effects of hereditary control, mutation type and crossing, J. Theor. Biol. 17 (1967) 319–342.
[44] J.P. Rosca, D.H. Ballard, Genetic programming with adaptive representations, Technical Report 489, University of Rochester, Computer Science Department, 1994.
[45] R.S. Rosenberg, Simulation of genetic populations with biochemical properties, Ph.D. Dissertation, University of Michigan, Ann Arbor, MI, 1967.
[46] C.G. Shaefer, The ARGOT system: adaptive representation genetic optimizing technique, in: J.J. Grefenstette (Ed.), Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum, Hillsdale, NJ, 1987.
[47] J.D. Schaffer, A. Morishima, An adaptive crossover distribution mechanism for genetic algorithms, in: J.J. Grefenstette (Ed.), Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms, 1987, pp. 36–40.
[48] R.E. Smith, Adaptively resizing populations: an algorithm and analysis, in: Proceedings of The Fifth International Conference on Genetic Algorithms, Morgan Kaufman, (1993), p. 653.
[49] W. Spears, Adapting crossover in a genetic algorithm, in: Proceedings of the Fourth Annual Conference on Evolutionary Programming, 1995, pp. 367–384.
[50] L. Spector, A. Robinson, Genetic Programming, autoconstructive evolution with the push programming language, in: Genetic Programming and Evolvable Machines, vol. 1, Kluwer, 2002, pp. 7–40.
[51] G. Syswerda, Uniform crossover in genetic algorithms, in: J.D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms, MKP, CA, 1989, pp. 2–9.
[52] E. Teller, Evolving programmers: the co-evolution of intelligent recombination operators, in: P. Angeline, K. Kinnear (Eds.), Advances in Genetic Programming, vol. 2, 1996.
[53] D.H. Wolpert, W.G. McReady, No free lunch theorems for optimisation, IEEE Trans. Evol. Comput. 1 (1997) 67–82.
[54] UCI machine learning repository, Available from http://www.ics.uci.edu/~mlearn/MLRepository.html