Journal of Experimental & Theoretical Artificial Intelligence
Vol. 00, No. 00, Month-Month 200x, 1–18

A-Brain: a general system for solving data analysis problems

Mihai Oltean
Department of Computer Science
Faculty of Mathematics and Computer Science
Babeş-Bolyai University, Kogălniceanu 1
Cluj-Napoca, 3400, Romania.
moltean@cs.ubbcluj.ro
(*Received 00 Month 200x; In final form 00 Month 200x*)

An intelligent system should be able to solve a wide range of problems from different domains. In this paper we propose a complex and adaptive system capable of solving various data analysis problems without needing human help for parameter settings. The presented system, called A-Brain, consists of several inter-connected components (a Decision Maker, a Trainer and several Problem Solvers) which provide a base for building complex problem solvers. The parameters of the Trainer's algorithm are problem independent. This fact is a requirement for the intelligent systems that cannot count on the human intervention while operating. The A-Brain system is applied for solving some well-known problems in the field of symbolic regression and classification. Numerical experiments show that A-Brain system is able to perform very well on the considered test problems.

1 Introduction

Intelligence is strongly related to the ability of solving multiple and different problems. General problems solvers such as Artificial Neural Networks, Evolutionary Algorithms, Particle Swarm etc, have traditionally been tested against one problem at one time.

The purpose of this research is to build a system capable of efficiently solving symbolic regression and classification problems. These problems are of great interest because they arise in many real-world applications. The choice for solving these problems was mainly motivated by the fact that their input and output has a well-defined structure which is easy to handle: in most of the cases both the input and the output are arrays of numbers.

For solving these problems we have chosen to apply a variant of Genetic Programming (GP) (11) because of its efficiency in dealing with this kind of situations.

When building the system, our first concerns were related to the human independence and the speed of solving problems.

A system is human-independent if it requires a little or at all intervention from the humans when a new (unseen) situation arises in the system. Our system should also obey to this rule because it has to be capable of solving a wide range of classification problems (having different complexities, different number of training data, different number of variables, etc) without any human interference. To achive this our system uses an adaptive mechanism for discovering the optimal solution.

We can achieve a high speed by storing the already computed solutions. Thus, if we have already discover (by using a GP technique) the solution for a regression or a classification problem it would be a good idea to store that solution instead of rediscovering it each time we need to solve that problem. Moreover, by using evolutionary techniques there is no guarantee that we obtain the same, best, solution each time we run the algorithm (this usually happens because Evolutionary Algorithms (8) use a lot of pseudo-random numbers). This is another reason for which storing the already obtained solutions is a very good idea.

The proposed system (called A-Brain) consists of 3 main parts: a Decision Maker, a Trainer and a set of Problem Solvers. When a problem is presented to the system the Decision Maker will decide which Solver will try to solve that problem. 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 set of already

existing Solvers. The training of new Solvers is performed by using examples which are requested from the user.

The A-Brain system has been used for solving several problems in the field of symbolic regression. In this paper we present the results for several interesting and difficult problems: even-3 and 4-parity, even and odd 11-parity, even-12-parity, sum of 7 numbers and a real-world problem taken from PROBEN1 (25). Results show that A-Brain system can perform very well on the considered test problems.

It is difficult to compare the A-Brain system with other problem solvers because the experimental conditions are different. We still have performed a raw comparison for the numerical experiments required to evolve Solvers. Results obtained by the A-Brain are worse than those obtained by the systems that uses a fixed population size and a fixed chromosome length (15; 19). However, the comparison is not fair because experimental conditions are different. The A-Brain system uses an adaptive mechanism for population size and chromosome length which means that it does not know which are the optimal values of these parameters for a given problem. This is different from other systems (15; 19) where several experimental trials have been performed in order to find this information.

A related approach is the CAM-Brain Machine (CBM) (4; 5), a FPGA-based hardware used for evolving neural circuits. The author's vision was to build a brain consisting of 75,000,000 neurons grouped in 64,000 modules. Each module would have a special function encoding a complex behavior which will be designed by a human engineer. However, the project was never finalized due financial problems.

The paper is organized as follows: Multi Expression Programming, used for training Solvers, is briefly described in section 2. The way in which multiple outputs can be easily handled by MEP is deeply discussed in section 2.3. Fitness computing in the case of regression (classification) problems is described in section 2.2 (2.4). The structure of the input required by the A-Brain system is deeply discussed in section 3. The proposed A-Brain system is presented in section 4. The Trainer and its underlying algorithm are presented in section 4.2. Several numerical experiments for solving problems are performed and their results are discussed in section 5. Future research directions are outlined in section 7. Finally, section 8 concludes.

2 Multi Expression Programming

The most important part of the A-Brain system is its Trainer which is used for generating new Solvers based on some training examples. *Multi Expression Programming* (MEP) (15; 16) [1] is used as underlying mechanism for the Trainer. MEP technique is briefly described in this section.

2.1 *Individual representation*

MEP genes are represented by substrings of a variable length. The number of genes per chromosome is constant and it defines the length of the chromosome. Each gene encodes a terminal or a function symbol. A gene encoding a function includes references 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 the 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

We employ a representation where the numbers on the left positions stand for gene labels (or memory addresses). Labels do not belong to the chromosome, they are provided here only for explanation purposes.

[1] Multi Expression Programming source code is available at www.mep.cs.ubbcluj.ro

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

2.2 *Fitness assignment process*

In this section we 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.

There is neither practical nor theoretical evidence that one of these expressions is better than the others. Moreover Wolpert and McReady (29) proved that the search algorithm's behavior so far (for a particular test function) cannot be used to predict its future behavior on that function. Thus one cannot choose a particular expression (let's say expression E_7) to store the output of the chromosome. Even if this expression proves to be useful for the first 10 generations we cannot guarantee that it will be the best option for all generations.

This is why 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. This type of encoding is very important because we can get more solutions within the same running time as in the case of one solution/chromosome. More than that, most GP techniques encode multiple solutions in a

chromosome, but they simply ignore them. For instance, when the quality of a GP tree is computed, the value of all its subtrees is also computed and this basically means multiple-solutions within a chromosome. However, GP ignores the values provided by subtrees and takes into account only the solution provided by the largest tree.

The value of these MEP expressions is computed by reading the chromosome top down. Partial results are computed by Dynamic Programming (3) and are stored in a conventional manner (in an array).

As MEP chromosome encodes more than one problem solution, it is interesting to see how the fitness is assigned. 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:

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

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:

$$f(C) = \min_i f(E_i). \tag{2}$$

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.

2.3 *MEP 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 Multi Expression Programming may be efficiently used for handling problems with multiple outputs (such as designing digital circuits (16)).

Suppose that each problem has one or more inputs (their number is denoted by NI) and one or more outputs (their number is denoted NO).

When the problem has multiple outputs, we have to choose NO genes which will provide the desired output (it is obvious that the genes must be distinct unless the outputs are redundant).

In Cartesian Genetic Programming (22; 23), the genes providing the program's output are evolved just like all other genes. In MEP, the best genes in a chromosome are chosen to provide the program's outputs. When a single value is expected for output we simply choose the best gene (see section 2.2).

When multiple genes are required as outputs we have to select those genes which minimize the difference between the obtained result and the expected output.

We have to compute first the quality of a gene (sub-expression) for a given output (how good is that gene for providing the result for a given output):

$$f(E_i, q) = \sum_{k=1}^{n} |o_{k,i} - w_{k,q}|, \tag{3}$$

where $o_{k,i}$ is the obtained result by the expression (gene) E_i for the fitness case k and $w_{k,q}$ is the targeted result for the fitness case k and for the output q. The values $f(E_i, q)$ are stored in a matrix (by using dynamic programming (3) for latter use (see formula (4)).

Table 1. An example on how to compute genes which will provides outputs. The problem has 3 outputs and the MEP chromosome has 5 genes. The value within cell (i, j) is the fitness provided by gene i for output j as computed by equation 3. The mathematical expressions that have generated these values are not important at this moment of explanation (this is why we have ommited them). Genes providing the outputs have been marked with *

Gene#	$Output_1$	$Output_2$	$Output_3$
1	5	3	9
2	7	6	7
3	1 *	0	2
4	4	1 *	5
5	2	3	4 *

Since the fitness needs to be minimized, the quality of a MEP chromosome may be computed by using the formula:

$$f(C) = \min_{i_1,i_2,\ldots,i_{NO}} \sum_{q=1}^{NO} f(E_{i_q}, q). \quad (4)$$

In equation (4) we have to choose numbers i_1, i_2, ..., i_{NO} in such way to minimize the program's output. For this we shall use a simple heuristic which does not increase the complexity of the MEP decoding process: for each output q ($1 \leq q \leq NO$) we choose the gene i that minimize the quantity $f(E_i, q)$. Thus, to an output is assigned the best gene (which has not been assigned before to another output). The selected gene will provide the value of the q^{th} output.

Example

Let's consider a problem with 3 outputs and a MEP chromosome with 5 genes. We compute the fitness of each gene for each output as described by equation 3 and we store the results in a matrix. An example of matrix filled using equation 3 is given in Table 1.

The way in which genes providing the outputs have been selected is describes in what follows: first we need to compute the gene which will provide the result for the first output. We check the 2^{nd} column in Table 1 and we see that the minimal value is 1 which correspond to gene #3. Thus the result for the first output of the problem will be provided by gene #3. For the second output we see that gene #3 is generating the lowest fitness. We cannot choose this gene again because it already provides the result for the first output. Thus we choose gene #4 for providing the result for the second output. According to the same algorithm (described above) the result for the third output will be provided by gene #5.

Remark

Formulas (3) and (4) are the generalization of formulas (1) and (2) for the case of multiple outputs of a MEP chromosome.

The complexity of the heuristic used for assigning outputs to genes is:

$O(NG \cdot NO)$,

where NG is the number of genes and NO is the number of outputs.

We may use another procedure for selecting the genes that will provide the problem's outputs. This procedure selects, at each step, the minimal value in the matrix $f(E_i, q)$ and assign the corresponding gene i to its paired output q. Again, the genes already used will be excluded from the search. This procedure will be repeated until all outputs have been assigned to a gene. However, we did not used this procedure because it has a higher complexity – $O(NO \cdot log_2(NO) \cdot NG)$ - than the previously described procedure which has the complexity $O(NO \cdot NG)$.

2.4 *Fitness assignment for classification problems*

The purpose of the A-Brain system is to solve both regression and classification problems. In the previous sections we have shown how to solve symbolic regression problems with MEP. In this section we briefly overview the modifications that has to be made to MEP in order to solve classifications problems.

Each class has associated a numerical label: the first class has the value 0, the second class has the value 1 and the m^{th} class has associated the numerical value $m-1$. Any other system of numbers may be used.

The value $o_{k,i}$ of each expression E_k (in an MEP) chromosome for each example i in the training set is computed. Then, each example in the training set will be classified to the nearest class (the class c for which the difference $o_{k,i} - c$ is minimal).

The fitness of a (sub)expression E_k is equal to the number of incorrectly classified examples in the training set.

The fitness of a MEP chromosome will be equal to the fitness of the best expression encoded in that chromosome (which is the expression that has generated the minimal number of incorrectly classified examples).

Remarks. Since the set of numbers associated with the problem classes was arbitrarily chosen it is expected that different systems of number to generate different solutions.

2.5 *Genetic operators*

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

2.5.1 ***Crossover.*** By crossover two parents are selected and 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 2. The two offspring O_1 and O_2 are obtained by uniform recombination as given in Table 2.

Table 2. 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

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

Example

Consider the chromosome C given in Table 3. If the boldfaced symbols are selected for mutation an offspring O is obtained as given in Table 3.

Table 3. 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

2.6 *MEP algorithm*

Standard MEP algorithm uses steady state (28) as its underlying mechanism. MEP algorithm starts by creating a random population of individuals. The following steps are repeated until a given number of generations [1] 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 an individual selected using a negative pressure (two random individuals are chosen and the worst of them is the result of selection).

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

3 The structure of the input

Problems are presented one by one to the A-Brain system. Each input of a problem in described by the followings 7 pieces of information. All of them are required when a new Solver is trained and usually only 6 of them (output is not required in this case) are needed when a Solver is called for solving a problem.

(i) The data type of the input that we want to analyze. The type could be: *binary*, *real* or *integer*. Currently all elements must have the same type, but this is not a hard limitation of the system because all types can be converted, for instance, to the *real* type (**double** data type in C language. The type must be specified because the operators involved in Solvers depends on this information.
(ii) The number of variables (features) of the information to be analyzed.
(iii) An array of values representing the output. For instance if we want to solve "the sum of 7 values" problem, our input will consist of 7 real values. The number of values in the inputs was specified above.
(iv) The data type of the output. The type could be: *binary*, *real* or *integer*.
(v) 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 (16; 23) or the *Building* problem (25)).
(vi) The array of values representing the output. This information is required only when a new Solver is trained. The output must have the same data type as the input (binary, integer or real).
(vii) In the case that there are multiple problems which have the same input we have to store some supplementary information. For instance, both even-3-parity and odd-3-parity problems (11)have the same

[1] In a steady-state algorithm, a generation is considered when the number of newly created individuals is equal to the population size.

Table 4. The pairs of types which are allowed to appear in our system. 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.

Input	Output	Applications
binary	binary	Boolean function finding
real	integer	classification
real	real	symbolic regression
integer	integer	classification

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 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).

Remark. Input and output cannot have any types. Only the pairs of types given in Table 4 are permitted. Some examples of input presented to the system are given below:

Example 1

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

binary, 3, 0, 0, 0, binary, 1, 1, even-parity.
binary, 3, 0, 0, 1, binary, 1, 0, even-parity.
binary, 3, 0, 1, 0, binary, 1, 0, even-parity.
binary, 3, 0, 1, 1, binary, 1, 1, even-parity.
binary, 3, 1, 0, 0, binary, 1, 0, even-parity.
binary, 3, 1, 0, 1, binary, 1, 1, even-parity.
binary, 3, 1, 1, 0, binary, 1, 1, even-parity.
binary, 3, 1, 1, 1, binary, 1, 0, even-parity.

Example 2

When we want to find the solution for this problem (assuming that the corresponding Solver already exists) we don't have to provide the output anymore. Below is such an example (only for one fitness case we are interested in finding a solution):

binary, 3, 0, 1, 0, binary, 1, , even-parity.

Example 3

In the cases that the user is not very sure if there is a Solver for the problem or it does not know the correct identifier he will also has to provide some example. 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. Below is an example:

binary, 3, 0, 0, 0, binary, 1, , .
binary, 3, 1, 0, 0, binary, 1, 0, .
binary, 3, 1, 0, 1, binary, 1, 1, .
binary, 3, 1, 1, 0, binary, 1, 1, .
binary, 3, 1, 1, 1, binary, 1, 0, .

4 A-Brain system

Our purpose is to build a system able to solve a wide range of regression and classification problems. The system consists of 3 main parts:

- a Decision Maker,
- a Trainer and
- a set of Problems Solvers.

When a problem is presented to the system the Decision Maker will decide which Solver will try to solve that problem. If no suitable Solver is found, the Decision Maker will activate the Trainer which will try to train a Solver for the problem. This new Solver will be added to the set of already existing Solvers. The training process of new Solvers is performed by using examples (also called fitness cases or training data) requested from the user.

Figure 1 presents a schematic view of the A-Brain system.

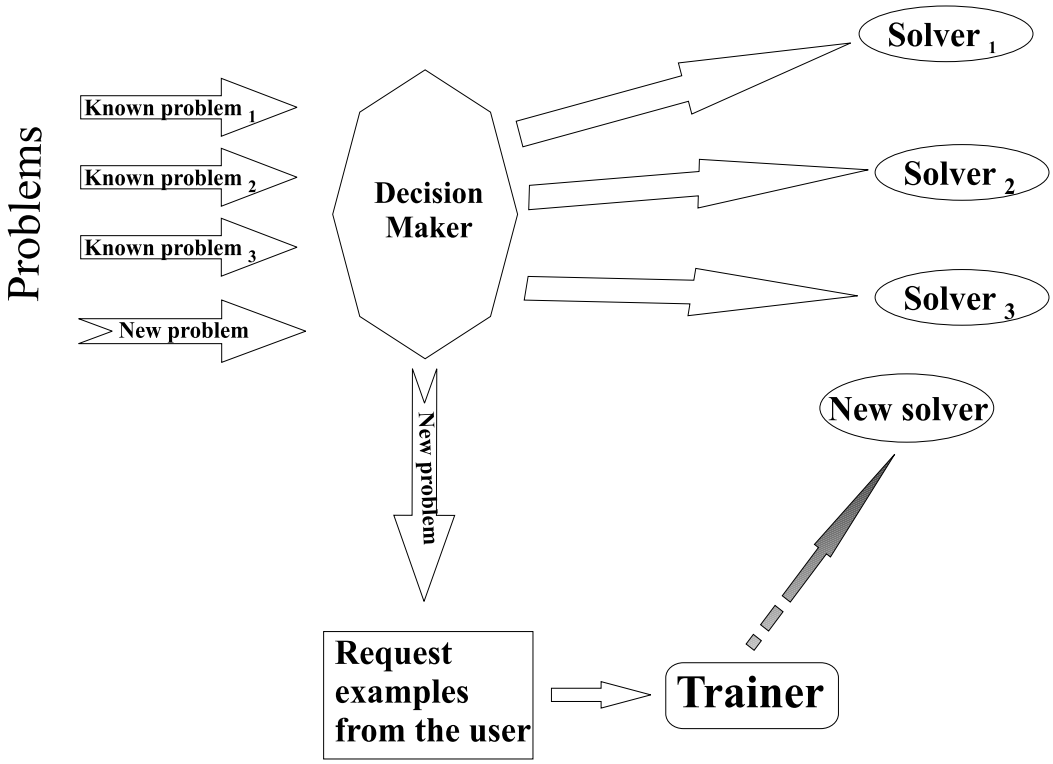

Figure 1. A-Brain system. Problems are presented to the Decision Maker. There are two kind of problems: known problems (the system knows how to solve them) and new problems (the system does not know how to solve them). In this second case the A-Brain system will request some examples. A new Solver will be evolved based on the training data.

The components of the A-Brain system are deeply described in the next sections of the paper.

4.1 *The Decision Maker*

The Decision Maker (DM) is a complex component of the A-Brain system. DM should decide whether the system is capable of solving a problem or not. The DM will call the corresponding Solver in the case that the system knows how to solve the problem. If the system is not able to solve the problem (because there is no Solver available for it) the DM should call the Trainer in order to generate a Solver for that type of problem.

The Decision Maker has a difficult task: based on the input data (the problem) it should decide the type of the problem. Generally speaking, if the input consists of 3 binary digits the DM should call the corresponding Solver which knows how to handle problems with 3 binary inputs. If the input is an array of real values the DM should call an Solver which is able to solve problems whose input is of that kind. An incorrect assignment of the problem to the Solvers will lead to an incorrect solution for that problem. This is why the DM should be the most accurate component of the system.

The aim of the DM is to make distinction between problems not to solve them. This is why the DM is very aware of the input's structure. Currently main part of DM is based on a simple algorithm that checks the data type, the size of the input and output data and the problem identifier. If this matches an existing type of problem the corresponding Solver will be called. The problem identifier is not enough because there may be multiple problems which have the same identifier. For instance many problems can be labeled with "even-parity" identifier, but they can have 1,2, or more inputs.

There are cases when the user is unable to provide the correct identifier for the problem. For instance:

- the provided problem identifier is wrong,
- the user does not remember it,
- the current user does not know it because it was created by another user.

In these cases the system offers another, more flexible way, to identify the problem and the suitable Solver for it: by using some training examples (provided by the user). The DM will check all the available Solvers which match the data type and the size of the input and output. Each of these Solvers will be applied for solving each example provided by the user. If the worst error generated by applying a Solver to each example is lower or equal to the error stored internally by that Solver (see section 4.3), 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 taken randomly.

These simple methods works for the problems that we have tested, but better algorithms (such as those used for text categorization (2; 26)) are required for more complex inputs.

Discipulus (14) software from AimLearning [1] has a similar feature: based on the input data decide the type of the problem: regression or classification.

4.2 *The Trainer*

When the Decision Maker is not able to find a suitable Solver for a particular problem, it will call the Trainer. This component will try to build a Solver for the problem being solved. For achieving this, the Trainer needs some examples. Usually these examples are supplied by the user.

The Trainer is basically an evolutionary algorithm which evolves computer programs. We have chosen Multi Expression Programming (15; 16) due to its simple representation and implementation. There is no particular reason for choosing this technique; any other Genetic Programming (15) method can be used instead. We also plan to create a version of our system which uses Neural Networks (9) (see section 7) as underlying mechanism for Trainer.

The Trainer's task is quite difficult because it has to compute the size of the Solvers. Also, 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 training set are two opposite examples of inefficient behavior. This is why we have chosen to implement an adaptive algorithm for the Trainer. This algorithm will try to find an optimal population and an optimal chromosome length for the problem being solved.

This modified MEP algorithm will start with a small population and a small value for the chromosome length. These values will be increased as the search process advances. Note that this is not a true adaptation (6; 10; 27) because the population size never decreases. We have chosen not to decrease the population size because Genetic Programming techniques usually require large populations (11) for solving problems. However, we do plan to investigate the effect of true adaptation in our system.

[1] www.aimlearning.com

From experimental trials we have deduced some good values for these parameters (the number of individuals added to the current population and the number of genes added to the current chromosomes length). However, the parameter settings are really problem dependent and a general strategy cannot be devise. This is due to the No Free Lunch theorems (29; 30) which state that we cannot use the algorithm's behavior so far in order to predict its future behavior. This means that even if the current strategy (parameter settings, genetic operators, etc) has generated good results so far, it cannot guarantee good results in the future (next generations).

Also, 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 being solved.

For achieving a great generalization ability we have used the following strategy (which was obtained after many trials):

- The algorithm starts with a population of 1 MEP individual each having 1 gene. The choice for these parameters (the initial populations size and the initial chromosome length) is problem independent.
- the size of the population and the chromosome length should doubled if no improvements of the best individual occur for $A = 10$ generations. The newly inserted individuals and genes are randomly generated.
- The algorithm will stop when there have been no improvements of the best individual in the population for $B = 50$ consecutive generations.

The values (A and B) should be chosen carefully because the algorithm's behavior greatly depends on them. An inappropriate choice of these values can lead to very poor results. For instance:

- If A is very small, the population size and chromosome length will increase too much.
- If A is too big the population might looses its diversity before a new increase will take place.
- The algorithm will stop very soon if the value for B is too small.
- The population size and chromosome length might increase too much if B is too big.

There are several reasons on why the algorithm is not 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 performed,
- The population size or the chromosome length are too small.

The last case means that we have to increase the value for either the population size or the chromosome length. We don't know which one should be increased, so we will increase the size of both of them.

We have chosen to double the population size mainly because we are using a steady-state MEP algorithm (see section 2.6). This algorithm will rapidly eliminate the worst individuals. When we generate a random 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 chances that not all new individuals will be eliminated. Some of them will actively participate to the search process.

Doubling the population size and the chromosome length is just one of the ways to adapt these parameters. For instance, one can triple these values instead of just doubling them. Or, one can increase them by only one tenth of their current values. Any strategy which increases the values for the population size and chromosome length in a direct proportional manner with their current values should work very well. Of course the adaptation strategy should be coupled with good values for A and B parameters. For instance, if the number of individuals which are added to the population is small, the value for A should be also small. This will ensure that the population size will still increase fast enough. However, we cannot devise any general strategy for this aspect. As we said before these parameters are problem dependent and no general suggestion will work for all problems (29; 30).

4.2.1 ***Function set.*** The function set used for evolving Solvers depends on the type of the involved data (inputs and outputs).

- Usually all 16 Boolean operators with 2 inputs an 1 output are used as function set (F_{Binary}) for binary inputs as described in (24). However, in other cases, a reduced set ($F'_{Binary} = \{AND, OR, NAND, NOR\}$) has been used in order to non-trivially solve smaller instance of the even-parity problem (11).
- The set $F_{Int} = \{+, -, *, \%, /\}$ is used for integer input. % will provide the modulo of the result and / will perform the integer division. These 2 operators are protected against division by zero by returning value 1 in this case.
- The set $F_{Real} = \{+, -, *, /, sin, exp\}$ is used for real values of the input.

4.3 *The Problem Solvers*

The Problem Solvers are some simple MEP individuals (computer programs) (see section 2.1) evolved by Trainer. In the current version of the A-Brain system each Solver (MEP 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 of the problem being solved. The number of variables is equal to the number of the problem's inputs.

Internally, a Solver also store the following information (which is used by the Decision Maker (see section 4.1)):

- Data type,
- Number of inputs (variables),
- Number of outputs,
- Problem identifier,
- The worst error for a fitness case from the particular.

4.4 *The A-Brain algorithm*

At the first run A-Brain starts with an empty set of Solvers. In other words the system does not know to solve any problem. When a problem is presented to the system it will decide if it knows how to solve it or not. If yes, the system will call the corresponding Solver. Otherwise it will ask for some training examples from the user. These examples will be used by the Trainer for evolving a new Solver. The Trainer is run once and the best obtained solution is saved as the new Solver. The Solver obtained after training state is added to the set of existing Solvers. For improving the performance of the system we could perform multiple runs for evolving a Solver and we could take the best of them.

The order in which the problems are presented to the system is not important and it will not affect its performance.

4.5 *System management*

The user can interfere with the system any time. The user actually can:

- Remove Solvers,
- Add Solvers by hand,
- Edit the existing Solvers,
- Ask for other statistics from the Trainer: success rate, computational effort (11),
- Repeat the training process for a particular Solver.

Parameter	Value
Population size at the beginning	1 MEP individual
Chromosome length at the beginning	1 gene
Number of generations without improvements before the training algorithm is stopped	50
Number of generations without improvements before the population size and chromosome length are doubled	10
Mutations	2 / chromosome.
Crossover probability	1
Selection	Binary tournament.

Table 5. Parameter settings for all experiments performed with A-Brain.

5 Numerical experiments

We have applied our system to a wide range of problems. We will show the results for several problems: even-3 and even-4 parity, even and odd 11-parity (11), even-12-parity, sum of 7 numbers and several real-world problem taken from PROBEN1 (25) (which have been adapted from UCI Machine Learning Repository (31)).

Since it uses a deterministic algorithm, the Decision Maker was able in all cases to take the expected decision. We will not focus anymore on this aspect.

The difficult task was performed by the Trainer which has had to evolve good Solvers for the considered problems. In what follows we will focus on the results obtained by the Trainer.

For computing some performance metrics we have performed 100 independent runs (when training new Solvers). However, in the standard variant of the A-Brain system only one run is performed and the obtained Solver is added to the set of existing Solvers. We could extend our program to perform multiple trials and to choose the best one, but we are not interested in this aspect at the current stage of the project. If the user is not satisfied by a particular Solver it can ask for multiple runs of the Trainer.

Some general parameters for Trainer are given in Table 5.

5.1 *Even-3-parity*

We have a string of 3 binary values and our output should be 1 if an even number of positions are 1 and 0 otherwise (11; 19). This problem has 3 Boolean inputs, one Boolean output and $2^3 = 8$ training examples. All those data have been used for training. For this problem we have used the reduced function set $F'_{Binary} = \{AND, OR, NAND, NOR\}$, because the set containing all 16 Boolean functions can lead too fast to a solution.

For the even-3-parity problem we have obtained 41 successful runs (out of 100). This means that in 41 runs (out of 100) the Trainer was able to evolve a perfect Solver for this problem. The smallest value (obtained in a successful run) for the population size and chromosome length was 16. The largest population contained 256 individuals. The average population size was 78.4 with a standard deviation of 14.9.

As a comparison, standard MEP (19) obtained 40 successful runs using a population of 100 individuals with 30 genes evolved for 100 generations. Note that this comparison is not fair because in our A-Brain system the population size and the chromosome length are adapted during the search process.

5.2 *Even-4-parity*

We have a string of 4 binary values and our output should be 1 if an even number of positions are 1 and 0 otherwise (11; 19). This problem has 4 Boolean inputs, one Boolean output and $2^4 = 8$ training examples.

All those data have been used for training. For this problem we have used again the reduced function set $F'_{Binary} = \{AND, OR, NAND, NOR\}$, because the set containing all 16 Boolean functions can lead too fast to a solution.

For the even-4-parity problem we have obtained 12 successful runs (out of 100). The lowest value (obtained in a successful run) for the population size and chromosome length was 128. The largest population contained 2048 MEP chromosomes with 2048 genes each. The average population size at the end of the successful runs was 480.9, standard deviation 34.6.

As a comparison, standard MEP (19) obtained 13 successful runs using a population of 400 individuals with 50 genes evolved for 100 generations. Note again that this comparison is not fair because in our A-Brain system the population size and the chromosome length are adapted during the search process.

5.3 *Even and odd 11-parity*

We have a string of 11 binary digits and our output should be 1 if an even/odd number of positions are 1 and 0 otherwise (11; 19). This problem has 11 Boolean inputs, one Boolean output and $2^{11} = 2048$ training examples. All those data have been used for training. The extended set of all 16 Boolean functions have been used in this case (see section 4.2.1).

For the even-11-parity problem we have obtained 6 successful runs (out of 100). This means that in 6 runs (out of 100) the Trainer was able to evolve a perfect Solver for this problem. The smallest population (able to obtain a solution) contained to 256 individuals and the shortest chromosome length was increased to 256 genes.

As a comparison, standard MEP (19) obtained 25 successful runs using a population of 10 individuals with 300 genes evolved for 100 generations.

We have tested our system against the odd-11-parity problem only because it has the same type of input as in the case even-11-parity. The Decision Maker was able to recognize the odd-parity problem as a new problem and to train a Solver for it. For the odd-11-parity problem we have obtained 8 successful runs (out of 100). The population has been increased to 160 individuals and the chromosome length has been increased to 160 genes.

5.4 *Even-12-parity*

For training we have used all $2^{12} = 4096$ data. One hundred independent runs have been performed.

For the even-12-parity problem we have obtained 3 successful runs (out of 100). The smallest value for the population size was 1024 individuals and the chromosome length was increased to 1024 genes.

As a comparison, standard MEP (19) obtained 80 successful runs using a population of 25 individuals with 500 genes evolved for 100 generations.

5.5 *Sum of 7 integer numbers*

In this problem we have 7 integer numbers and we want to compute their sum. For training we have used 100 randomly generated data. Each number was uniformly generated over the interval [0..1000]. One hundred runs have been performed.

Our Trainer was able to find 18 perfect solutions (out of 100 runs). The smallest population able to obtain a solution has 128 individuals and the chromosome length was increased to 128 genes.

5.6 *The Building problem*

The purpose of this problem is to predict the electric power consumption in a building. The problem was originally designed for predicting the hourly consumption of hot and cold water and electrical energy based on the date, time of day, outdoor temperature, outdoor air, humidity, solar radiation, and wind speed. In this paper, the original problem was split into three subproblems (predicting the consumption of

electrical power, hot water, and cold water) because not all GP techniques (involved in the comparison) are designed for handling multiple outputs of a problem. A-Brain system was used for training a solver with 3 ouputs. However, only the results for prediction of cold water consumption was compared with the other GP systems.

The Building problem has 14 inputs restricted to the interval $[0, 1]$. The data set contains 4208 examples (25; 31). Whereas in (25) the data set was divided into training, validation, and test sets, we apply our algorithm for the entire set of 4208 data in order to see the efficiency of our algorithm. Test data can be presented any time later to the A-Brain system for further measurements.

In PROBEN1 three different variants of each dataset are given concerning the order of the examples. This increases the confidence that results do not depend on a certain distribution of the data into training, validation, and test sets (25). The original problem is called *Building1* and the other two versions are called *Building2* and *Building3*.

We have performed again 100 independent runs for the *Building1* set. In this case we compute only the training error (15; 25) and the standard deviation for that error.

The average error was 7.58 with a standard deviation of 2.19. The best (minimal) error was 6.41. The population size which was able to obtain the minimal solution contains 1024 individuals and the chromosome has 1024 genes.

As a comparison, standard MEP technique was able to obtain an average error of 3.81 with a population of 100 individuals with 34 genes evolved for 100 generations (see (15) for more results). In that case (15) only 50% of the examples were used for training; the other data was used for validation and testing. Note that the results obtained by A-Brain are worse than those obtained by standard MEP, but are better than those obtained by Gene Expression Programming (GEP) (more information about GEP can be found in (7)) and Grammatical Evolution (GE) (more information about GE can be found in (21)) techniques (15).

6 Discussions

The A-Brain system is endowed from the beginning with a Trainer which knows how to construct Solvers for some particular classes of problems. Also the Trainer knows how to evolve Solvers, for some particular problems, from the beginning of the process. This is why we cannot call our system *intelligent*.

Taking into account the No Free Lunch theorems for Search and Optimization (29; 30) we cannot expect to evolve *the best* Solver for a given problem. No claims on the generalization ability of the Trainer should be made before seeing the experimental results.

At least at this stage there are some important differences between A-Brain and CBM (4): the A-Brain's modules are evolved by another computer program (the Trainer), whereas CBM's modules are designed and evolved by human engineers.

7 Further work

Some work needs to be done for improving the A-Brain system. Short term efforts will be spent in the following directions:

(i) Improving the performance and the speed of the system.
(ii) Using alternate systems for some parts of the A-Brain. For instance, in some cases, the output of a Solver is just a number. In this case we can use Traceless GP (TGP) (17) as a replacer for MEP. TGP is a special GP variant which stores only the output of the current program. The speed of the Solvers will be increased significantly in this case as shown in (17).
(iii) Using Artificial Neural Networks (9) for some parts (Decision Maker and Problem Solvers) of the A-Brain system. A comparison between these two representations will be done as soon as ANN-based A-Brain will be ready and up.
(iv) Using Sub-machine code GP (24) for improving the speed of training solvers in the case of binary inputs.

Long term goals include:

(i) Extending the system so that it could be able to deal with problems from other interesting classes.
(ii) A big problem with the Solvers is that an exponential number of problems will lead to an exponential number of Solvers. An idea for avoiding this problem is to replace Solvers by Evolutionary Algorithms (8) or by some complex strategies. Thus each Solver will be a complete evolutionary algorithm able to solve problems of a particular class. There will be an important benefit if this method is adopted: the increased generalization ability. An evolutionary algorithm is able to solve a class of problems rather than solving particular instances of a problem. Thus, instead of having a Solver for each instance of a particular problem (for instance TSP, QAP (13), Boolean function finding) we will have a single evolutionary algorithm able to evolve solutions for all particular instance of a given problem. A robust system for evolving evolutionary algorithms has been proposed in (20) and it will be used as Trainer in the A-Brain system.
(iii) In some cases the solution encoded into a MEP Solver might be longer than the algorithm which has generated it. In this case it could be better to store the algorithm instead of the solution generated by that algorithm.
(iv) Inserting A-Brain system into a hardware system for testing it in real-world situations. In this way a better parallelization of the system is obtained allowing it to evolve/solve multiple problems in the same time.
(v) Providing a mechanism of interaction between Solvers. In order to improve the generalization ability, the Solvers must be able to exchange information. This interaction is a killer feature of the system; without it there are low hopes for good generalization abilities.
(vi) Natural language interface for the system.

8 Conclusions

A new system called A-Brain has been proposed in this paper. A-Brain has 3 main components: a Decision Maker, a Trainer and many Problem Solvers. The decision on whether the system knows or not how to solve a problem is taken by the Decision Maker. The Trainer - which is the most important part of the system - is based on a linear variant of Genetic Programming namely Multi Expression Programming. Each problem has its own Solver (a MEP program) which is clearly defined by several parameters.

In its current stage, the A-Brain system is able to handle any kind of regression and classification problem. If it does know how to solve it, the system calls a special Solver for that problem. If it does not know how to solve it, 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 are required in this part of the system.

A-Brain system has been used for solving several well-known problems in the field of symbolic regression and classification. Numerical experiments have shown that A-Brain was able to handle a wide range of problems yielding good results.

Acknowledgments

More information about the A-Brain system (including the source code) will be available at www.cs.ubbcluj.ro/∼moltean/a-brain/index.htm

References

[1] Aho, A., Sethi R., Ullman J., Compilers: Principles, Techniques, and Tools, Addison Wesley, 1986.
[2] Apte, C., Damerau, F., Weiss, S.M., Automated learning of decision rules for text categorization, ACM Transactions on Information Systems Vol 12: 3, pp. 233 - 251, 1994.

[3] Bellman, R., Dynamic Programming, Princeton, Princeton University Press, NJ, 1957.
[4] de Garis, H., Korkin, M., The CAM-BRAIN Machine (CBM): An FPGA Based Hardware Tool which Evolves a 1000 Neuron Net Circuit Module in Seconds and Updates a 75 Million Neuron Artificial Brain for Real Time Robot Control, Neurocomputing journal, Elsevier, Vol. 42 (1-4), 2002.
[5] de Garis, H., (et al.), CAM-Brain ATR's Billion Neuron Artificial Brain Project : A Three Year Progress Report, Artificial Life and Robotics Journal, Vol. 2, pp. 56-61, 1998.
[6] Eiben, A.E., Hinterding, R., and Michalewicz, Z., Parameter Control in Evolutionary Algorithms, IEEE Transactions on Evolutionary Computation, Vol.3, Issue 2, pp. 124-141, 1999
[7] Ferreira, C., Gene Expression Programming: a New Adaptive Algorithm for Solving Problems, Complex Systems, Vol. 13, Nr. 1, pp. 87-129, 2001.
[8] Goldberg, D.E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989.
[9] Haykin, S., Neural Networks, a Comprehensive Foundation, second edition, Prentice-Hall, Englewood Cliffs, 1999.
[10] Hinterding, R., Michalewicz, Z., and Eiben, A.E., Adaptation in Evolutionary Computation: A Survey, Proceedings of the 4^{th} IEEE International Conference on Evolutionary Computation, pp.65-69, 1997
[11] Koza, J.R., Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, 1992.
[12] Koza, J.R., Genetic Programming II: Automatic Discovery of Reusable Subprograms, MIT Press, 1994.
[13] Krasnogor, N., Studies on the Theory and Design Space of Memetic Algorithms, PhD Thesis, University of the West of England, Bristol, 2002.
[14] Nordin, P., A Compiling Genetic Programming System that Directly Manipulates the Machine Code, in Kenneth E. (editor), Advances in Genetic Programming, pp. 311-331, MIT Press, 1994.
[15] Oltean, M., Groşan, C., A Comparison of Several Linear Genetic Programming Techniques, Complex-Systems, Vol. 14, Issue 4, pp. 282-311, Champaign, IL, 2003.
[16] Oltean, M., Groşan, C., Evolving Digital Circuits using Multi Expression Programming, NASA/DoD Conference on Evolvable Hardware, 24-25 June, Seatle, Zebulum, R., (et al.) (editors), pp. 87-90, IEEE Press, NJ, 2004.
[17] Oltean, M., Solving Even-Parity Problems using Traceless Genetic Programming, IEEE Congress on Evolutionary Computation, CEC'05, G. Greenwood (editor), pp. 1813-1819, IEEE Press, 2004.
[18] Oltean, M., Dumitrescu, D., Evolving TSP Heuristics using Multi Expression Programming, International Conference on Computational Sciences, ICCS'04, Bubak, M., (et al.) (editors), LNCS 3037, pp. 670-673, Springer-Verlag, Berlin, 2004.
[19] Oltean, M., Improving Multi Expression Programming: an Ascending Trail from Sea-level Even-3-parity Problem to Alpine Even-18-Parity Problem, Evolvable Machines: Theory and Applications, Springer Verlag, Berlin, Nedjah, N., (et al.) (editors), pp. 229-255, 2005.
[20] Oltean, M., Evolving Evolutionary Algorithms using Linear Genetic Programming, Evolutionary Computation, MIT Press, MA, USA, Vol. 13, Issues 3, pp. 387-410, 2005.
[21] O'Neill, M., Ryan, C., Grammatical Evolution, IEEE Transaction on Evolutionary Computation, Vol. 5, Issue 4, pp. 349-358, 2001.
[22] Miller, J.F., Thomson, P., Cartesian Genetic Programming. In Proceedings of the 3^{rd} International Conference on Genetic Programming (EuroGP2000), R. Poli, J.F. Miller, W. Banzhaf, W.B. Langdon, J.F. Miller, P. Nordin, T.C. Fogarty (Editors), LNCS 1802, Springer-Verlag, Berlin, pp. 15-17, 2000.
[23] Miller, J. F., Job, D., Vassilev, V.K., Principles in the Evolutionary Design of Digital Circuits - Part I, Genetic Programming and Evolvable Machines, Vol. 1(1), pp. 7–35, Kluwer Academic Publishers, 2000.
[24] Poli, R., Page, J., Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine-code GP and demes. Genetic programming and evolvable machines, Vol 1, pp. 37-56, 2000.
[25] Prechelt, L., PROBEN1 - A Set of Neural Network Problems and Benchmarking Rules, Technical Report 21, University of Karlsruhe, 1994.
[26] Sebastiani, F., Machine Learning in Automated Text Categorization, ACM Computing Surveys, Vol

34(1), pp. 1-47, 2000

[27] Smith, R. E., Adaptively Resizing Populations: An Algorithm and Analysis, Proceedings of The Fifth International Conference on Genetic Algorithms, Morgan Kaufman, pp. 653, 1993

[28] Syswerda, G., Uniform Crossover in Genetic Algorithms, Schaffer, J.D., (editor), Proceedings of the 3^{rd} International Conference on Genetic Algorithms, pp. 2-9, MKP, CA, 1989.

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

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

[31] UCI Machine Learning Repository, Available from www.ics.uci.edu/∼mlearn/MLRepository.html