Problem 3. Genetic Programming

Victor Montiel Argaiz

August 15, 2012

Contents

1 Introduction **2**

2 Description of the GP **2**
- 2.1 Encoding . . . 2
- 2.2 Evaluation . . . 3
- 2.3 Initialization . . . 4
- 2.4 Mutation . . . 4
- 2.5 Recombination . . . 5
- 2.6 Selection . . . 5
- 2.7 Termination Condition . . . 6
- 2.8 Bloat effect . . . 6

3 Code Structure and Implementation **6**
- 3.1 Evolving Objects Library . . . 7
- 3.2 Additional Development . . . 8
- 3.3 Code Structure . . . 8

4 Results **10**
- 4.1 Varying size of population . . . 10
- 4.2 Mutation and Recombination Probabilities . . . 11
  - 4.2.1 Mutation Probabilities . . . 11
  - 4.2.2 Recombination Probabilities . . . 14
- 4.3 Selection . . . 14
- 4.4 Training set . . . 18
- 4.5 Tree Depth Evolution . . . 20
- 4.6 Summary of solution with simplified symbol set . . . 22
- 4.7 Solution with a complete symbol set . . . 22

5 Summary and Improvements **23**

6 Appendix **23**

A Program Compilation **23**

B Program Execution **24**

C Acronyms **24**

1 Introduction

The aim of this problem is to present *Genetic Programming (GP)*, one of the newest members within the *Evolutionary Computing (EC)* family. Although the underlying idea of *GP* is still the same, using evolutionary-like techniques to solve a wide range of problems, the are two main differences with respect to the other techniques previously studied. The first distinctive feature is the representation of the chromosomes. While in the previous specializations of *EC*, such as *Genetic Algorithm (GA)*, or *Evolutionary Strategy (ES)* we use linear-structures (array) of a given data type (strings, bits, real numbers) to represent the individuals, in *GP* trees are used instead. The other important distinction is that the previous techniques are mainly used for optimization problems, where we have to **search for parameters values** which minimizes the objective function, while in *GP* we use **models** which achieve a maximum fit, that is, *GP* techniques could be considered as *machine learning* methods instead of a simple optimization solver.

In order to get acquainted with the *GP* methods, the exercise tackles the problem of finding a closed formula for the solution of the quadratic equation in formula (1):

$$ax^2 + bx + c = 0 \tag{1}$$

Given the coefficients a, b, c defining the equation, the algorithm must derive an expression $f(a, b, c)$ which reproduces the well-known formula (2). To derive this expression, the algorithm uses a *training-set*, containing several examples of coefficient sets representing each an equation. The solution $f(a, b, c)$, a program specifying the operations to apply on the coefficients to calculate the unknown of the equation, is built so that when applied to the quadratic equation in formula (1), the result of the left-hand polynomial expression is zero.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

2 Description of the GP

As with other paradigms within the EC family, one of the first steps when developing the algorithm is to clearly state the encoding as well as the transformation operators which will apply on each individual.

2.1 Encoding

In *GP* the encoding for the genotype is a tree, that is, a connected graph in which there are no cycles. The tree nodes belong to one of the two sets, the *function set*, for the internal nodes, which encode the allowed operators and the *terminal set*, which are the leaves of the tree, that is, nodes not containing subtrees under them, and which encode, in this particular problem, the constant and coefficients. In the general case, we might not know the operators and constants to include in the two sets, which might do the search process harder, since we have to include more symbols that we may require. However, in well-known problems such as the quadratic equation, the set of both symbols and operators are perfectly known. Intuitively, the larger the *function set* and *terminal set*, the harder the search process for the *GP* algorithm since the search space grows exponentially for each new symbol. Thus, from the very beginning of the implementation of our algorithm, it is worth to spend some time to carefully choose a symbol set as reduced as possible, since this will limit the possible *s-expression* the algorithm has to explore and so, will speed-up the convergence to the final solution.

In our implementation, the *function set* is made up of binary operators such as $+$, $-$, $\times$, $\div$, and the unary operator $\sqrt{}$, the square root. Observe that the $\div$, $\sqrt{}$ might pose some problems in the tree evaluation, since both operators are not defined in cases when the denominator is zero for the $\div$ operator or the

expression under the radical square-root operator $\sqrt{}$ is negative. There are many work-arounds for these ill-defined expressions. In this problem we have opted to use the simplest, and also the most commonly used, solution. For the $\div$ operator, we have circumvented the problem of zero-division using the 'à la Koza' modified $\div$ operator (see [3]) where, when the denominator is zero, the result of the operator is always equal to 1. For the $\sqrt{}$ operator we have employed complex-numbers through the entire evaluation of the tree, so that the program can correctly deal with this cases. Observe also that, by introducing complex-arithmetic in the tree evaluation, we also add the capability of finding the solution for second order equations with imaginary solutions. Other more intelligent treatments are possible, for example, returning a poor fitness for the individual, or even not allowing at all in initialization, mutation and crossover operators the generation of subtrees that potentially could lead to these situations. However, this last method is by far much more complex to implement and for the sake of simplicity, we have opted for the above solution.

On the other hand, the *terminal set* is made up of the coefficients of the quadratic equation and, eventually, constants such as integer numbers necessary to build the final solution. Observe however that these numerical constants $2, 4, \ldots$ might not be necessary at all for this particular problem, since we can reproduce them anyway with arithmetic expression such as $\frac{b+b}{b} = 2$. In order to solve the problem step by step, and following the advice at the beginning of this section of keeping the symbol set as small as possible, we have used two different *sets*, one *simplified symbol set*, which paves the way for the final implementation of the algorithm, including the minimum set of constant to reach the solution, and one extended, with the constants we would include in a naive implementation. Observe that from the definition of the quadratic equation in formula (1), we can derive a simpler expression, without loss of generality, assuming $a = 1$. In this case we can get rid of the a symbol in the terminal set. Also, defining $d = \frac{b}{2}$, we can simplify the solution in equation (2) to the solution in equation (3), where we have assumed $a = 1$. With this simplified set of symbols, getting rid of a and 2, we have reduced considerably the search space. Finally, we have equally implemented the algorithm using the natural *extended symbol set*, including $a, b, c, 2$ as symbols in the *terminal set*. The difference in speed and results performance is notorious, always in favor of the simplified set, which restates the importance of spending some time to carefully choose the minimum set of symbols needed to solve a problem whenever this is possible from a theoretical point of view. A diligent and correct simplification of the problem at the early stages can save up time and complexity, as the quotation says, 'Artificial Intelligence is no match for natural stupidity', so, when initially tackling with a new problem, it is worth to spend some time studying and simplifying the problem instead of using the naive brute force approach from the very beginning.

$$x = -d \pm \sqrt{d^2 - c} \tag{3}$$

After a first implementation with the above operators, we have realized that the algorithm always produced the same result, $\sqrt{d^2 - c} - d$ (for the simpler symbol set), instead of generating the two possible solutions. Indeed, this is so because with the *function set* defined, we cannot generate the other solution $-\sqrt{d^2 - c} - d$ unless we include either the unary $-$ operator or the zero constant. Either with this new operator or the zero constant we would be now enabled to build the second solution as $-\sqrt{d^2 - c} - d$ or $0 - \sqrt{d^2 - c} - d$. For our implementation, we have included the unary $-$ instead of the zero constant, since we have thought that for this particular problem the addition of zero does not add diversity to the exploration, since most of the operators are either invariant $(0 + a, a - 0)$ or null $0 \cdot a$, $\frac{0}{d}$ when applied to a zero argument. The addition of the unary $-$ operator have increased the complexity of the *s-expressions*, thus, increasing the computation time needed for convergence and reducing the SR score maintaining the rest of parameters equal, but it has enabled to successfully find the two valid solutions.

2.2 Evaluation

With the encoding above, each individual in the population is a *symbolic-expression* including coefficients, constants and operators. When substituting the coefficients a, b, c by real numbers and evaluating the tree, we are calculating the unknown of the equation. In case the individual represents the right solution,

the tree evaluation will be zero, since this is, by definition, the solution of the quadratic equation. Typically, as in other learning problems, we have a *training-set* containing different instances of the problem that are passed to each individual, generating for each instance a given candidate solution.

Thus, the fitness evaluation function for this problem consists in the evaluation of each individual against every triplet a, b, c from the *training set*. The partial results of each evaluation are aggregated in the final fitness function through the *Root Mean Square (RMS)* metric. Consider $fitness_i$ as the fitness for the ith individual, $tree_i(a_j, b_j, c_j)$ as the evaluation of the tree for the triplet j in the training set, and $f(x; a, b, c) = ax^2 + bx + c$ as the evaluation of the second order polynomial representing the equation. The fitness function is thus defined in equation (4), being M the number of instance problems in the training-set:

$$fitness_i = \sum_{j=0}^{M} (f(tree_i(a_j, b_j, c_j); a_j, b_j, c_j) - 0)^2 \tag{4}$$

The closer $fitness_i$ to zero the closer to the solution the individual is. Notice that the individual representing the right solution will evaluate 0 in all instances from the *training set*. However, the fact that an individual evaluates 0 in all problem instances from the *training set* is not enough to be considered as the right solution, since it could be that the *training set* does not contains enough cases of second order equation to generalize the solution. Think for example that, in the case of a *training set* containing only *degenerated* quadratic equations, that is, $ax^2 + c = 0$, many individual may evaluate all problem instances to zero while not being the correct solution thus failing to provide the results for the generalized second order equation. For this reason the *training set* must be diverse enough to contain different types of equations.Obviously, the greater the *training set* used for the function evaluation, the longer the evolutionary process will take to reach the solution, so we must find a balance between size and speed to reach the optimal results.

Other evaluation functions are possible for this problem. Since the quadratic equation problem has a well-known close solution, we might use the known solution to evaluate each individual approximation against it. This way, we could define the evaluation function as the *RMS* of the difference between the known solution x defined in equation (2) and the approximate solution $\hat{x} = tree_i(a, b, c)$. We have discarded this evaluation function from the very beginning from two main reasons. The first one, because in the general case, we might not know the solution to the problem we are studying, and more important, there might not even be a close-form solution for it. The second reason is that, since in the general case we obtain two solutions for each instance problem, we should define the fitness with respect to of one of the two solutions, usually the closest, creating this way a possible ambiguity in the evaluation process.

2.3 Initialization

Initialization of individuals is performed by extracting repeatedly nodes from both the *function* and *terminal set*. The procedure draws randomly a node from the set of symbols. Depending on the arity of this symbol, the procedure is called recursively on each of the branches of the node. When a *0-arity* symbol is drawn (*terminal set*), the recursive call is finished. On the other hand, the procedure must keep track of the current depth of the tree. When the initialization method is called, we must provide the maximum depth of the tree that we want to create, once the initialization procedure achieves this maximum depth, the next symbol to be drawn must be restricted to the *terminal set*, to ensure we properly finish the tree with *0-arity* leaves.

2.4 Mutation

In *GP*, the mutator operators are different from other techniques of evolutionary computing given that the genotype representation is no longer a linear data structure such as an array, but a tree-like structure. The mutator operators performs modifications in the individual either by changing a single node in the

tree or by modifying a sub-tree under a randomly given node.

We have experimented with five different mutator operators in this problem. The first one is *Branch Mutation*, which, given a randomly chosen node from the tree, it replaces the sub-tree under it by a randomly created subtree. The creation of the new subtree is similar to the one performed in the initialization of each individual. *Point Mutation* consists in replacing a randomly chosen node, by another node of the same arity. *Expansion Mutation* is a particular case of the *Branch Mutation*, and replace a terminal node by a randomly created subtree, increasing thus the depth of the tree. *Collapse Subtree Mutation* has the opposite effect, and, given a random node, replaces a subtree with a randomly chosen terminal node. Finally, *Hoist Mutation* replaces an individual by one of its subtrees, given a randomly chosen node in the initial tree.

Observe that when applying mutator operators which potentially can increase the maximum depth of the tree, we must ensure that the individual does not grow beyond a given maximum depth, to avoid the *bloat* effect. On the other hand, in order to experiment with several mutation methods and to achieve a maximum diversity in the population, we have made a mutator operator which combines the five abovementioned methods. Each individual method from the five above is assigned a probability and when the global mutator operator is called, it calls a method out of the five according to its probability assignment.

2.5 Recombination

Recombination or Crossover operators in *GP* consists mainly of the *Subtree Crossover*, which takes two random nodes, one from each parents, and swap the subtrees under them, creating two new individual with the genetic material interchanged. The operator is widely explained in [1] and [3].

2.6 Selection

The selection process in *GP*, is used, as in the other techniques in the evolutionary computing family, twice through each generation of the algorithm. The first time it is used to choose, amongst the individual of the population, the parents which will mate to create new individuals. Once the off-springs have been created, we have to trim the population so that we keep the number of individuals constant at each generation. The selection operator are basically the same as in other techniques. We can thus apply the same concepts of roulette-wheel and tournament selection, (μ, λ) and $(\mu + \lambda)$ schema, *fitness-proportional* and *ranking selection*.

A usual selection scheme for *GP*, *Steady-State Genetic Programming (SSGP)*, has also been considered. The standard methods traditionally use two populations at the selection stage, one containing the parents and another containing the off-springs, that will be merged at a later stage. The *SSGP* method uses the same population for parents and off-springs, that is, it creates the next generation in the same parent population, when the mating process is completed, the new offspring takes places of the parent population, being able to be eligible for mating within the same generation. In this process of generating the new population each generation continues until no parents remain in the new generation. This technique is well suited for *GP* since the fact of avoiding creating two different populations, one for parents and one for off-springs, halves the memory consumption. This can be important in *GP* where population sizes can be considerably big and individual representation is usually more costly than linear structures.

For this problem we have used *tournament-selection* as the selection algorithm, and we have evaluated several method of survival selection, such as (μ, λ), $(\mu + \lambda)$, *survive and die* and *SSGP* schema based on ranking to increase the selection pressure. The choice of the ranking method instead of fitness proportional has been done in order to compensate for the destructive effects of the mutation and crossover operators. Effectively, contrary to others techniques such as *GA* or *ES* where the application of the mutation operator creates a new individual usually very similar to the initial one, in *GP* the application of these operators,

even if just one node is changed, can lead to individuals with very different fitness function values. So, an elitism approach has been necessary to avoid this destructive effects and retaining only a few good mutations.

2.7 Termination Condition

Two termination conditions have been implemented in this problem set. Firstly, the usual termination condition of *maximum number of generation* achieved was developed. Under this termination condition, the algorithm stops evaluating new generations once the maximum number of generations, passed as a parameter to the program, has been achieved.

At the early stages of the design of the algorithm, we have stated that the convergence is best and faster when the population is big, usually leading to the solution in a few generations. Since managing large populations takes considerably more time, and at the same time the solution is typically achieved in a few generations, we have implemented a new and complementary termination condition which stops the algorithm once we detect we have converged to the solution of the problem. This condition is what we have called *convergence condition*. In the quadratic equation problem, this condition is easily checked, since, providing we use a diverse enough *training-set*, we know that we have reached the solution when the resulting fitness function evaluates to zero. This termination condition has considerably reduced the execution time of the algorithm, specially when running batch executions of up to 30 repetitions, avoiding useless new generations once the solution has been accomplish.

2.8 Bloat effect

The bloat effect issue has been tackled in our implementation by limiting the depth of the tree at every stage where the individuals can potentially grow. The fist stage where this limit must be imposed is at initialization of the individual. As we discussed previously, the initialization process is done by growing recursively the tree until we reach either a *terminal node* or the maximum depth. The other stages where we must apply this procedure is on mutation and recombination operators. In some mutator operators, we grow a subtree starting from a given node of the tree. In this process we must ascertain that the current depth is less than the maximum permitted depth, or, in case of being equal, finish the tree with a node from the *terminal set*. Finally, at recombination, after the sub-trees have been swapped, we will check that no node is at a depth greater than the maximal, finishing the three with a terminal node in case of being at the limit point.

Another powerful idea that has not been considered on this problem because of its difficult implementation, but which eventually might speed-up the algorithm and at the same time limit the *bloat effect* is the idea of *optimization* or *simplification*. Random trees that represent individuals are usually very redundant, having expressions such as $(b-b)+(c-c)$ that can easily be simplified to 0. An optimization procedure at a later stage of the algorithm, and previous to the evaluation, would lead to a speed-up in the evaluation of each individual, and, at the same time, could limit considerably the size of the trees, getting rid of useless sub-branches which evaluate to a more simpler expressions. This tree optimization or simplification technique has been widely studied, specially for compilers and computer algebra systems.

Nevertheless, in this particular problem and implementation, as we will see in results section, the bloat effect does not appear to be an issue, since the size of the individuals does not usually overpass the limit size we define for each simulation.

3 Code Structure and Implementation

For the purpose of this project we have decided to use an external open source framework for EC. In previous projects we wrote our own code to solve the problems, which gave us some insight about the design and development of GA and ES. The main advantage on writing the own libraries is that one grasps

better the ideas, and the behavior of the algorithm if he goes through all the stages of the development, since designing, debugging and testing of the code provide many times with a unique view allowing for a superior understanding.

However, once one gets the previous experience, it might be also interesting to employ existing frameworks and libraries to solve problems. The strong points for this arguments, are, mainly:

- The use of other people libraries may provide you with new ideas of design and development for EC frameworks.
- Many times good libraries and frameworks become a de-facto solution, being used by many teams around the world, which makes the code more robust, and generic for a wide variety of problems, enabling people to focus on the underlying problem and not in the particular implementation, since the latter is usually well-known by experts in the field.
- Ultimately, large projects are usually developed by teams, and one learns to better programming by using other people code.

Observe however that, many times, when writing professional high-level or cutting-edge applications, we might be forced again to write our own framework, since existing frameworks do not satisfy our needs, or the performance we might require is not achieved with general purpose libraries. For example, we might be interested in developing a particular application to be solved by an specific Graphic Processing Unit (GPU) platform. In these cases, previous experience with well-established libraries might provide with a unique insight for a better design of our own library.

3.1 Evolving Objects Library

After exploring several ready-made C++ solutions, the external library which have been used to solve this problem is *Evolving Objects (EO)* [2]. EO is a template-based, ANSI-C++ evolutionary computation library which helps you to write your own stochastic optimization algorithms. It uses an object-oriented approach combined with template definitions for the classes, which makes the code generic, and fast at the same time by avoiding the use of polymorphism.

The library is based on a component-based framework, and the design of a new application with EO consists in choosing the components you need for your specific needs. For many classical problems, with usual encodings, mutation and recombination operators, components are already available, which limits the development of the code to implement the specific fitness function. For other more exotics challenges, you might probably have to write your own specializations of your individuals, and additionally mutation and recombination operators. However, the other components such as selection and replacement operators are generic enough to be used on any problem.

The framework provides classes for the main algorithm paradigms within the EC world, such as GA, ES, GP. It contains ready-made mutation and recombination operators for the main encodings. Selection and replacement operators are designed in a generic way, being able to chose between a wide variety of methods such as *rank-based*, *fitness-proportional*, *deterministic and stochastic tournaments*, *roulette*.

Finally, it provides the *CheckPoint* component, which are objects that are called every generation to perform some computations apart from the fitness evaluation, and which helps guiding the algorithm to the solution. For example, termination conditions are implemented as *CheckPoint*, being able to use generic conditions such as maximum number of generation achieved, minimum tolerance for a solution achieved, maximum number of generation with no improvement in the fitness evaluation and so on. Also, monitoring of the algorithm is made by using *CheckPoints*. At each generation, we are able to print out, log, write to a file, or create a graph, of any information we may consider relevant, such as best, average

and worst fitness values, best individual and computation time spent.

The library provides many other functionalities which have not been used in the problem, such as command-line parsing, and save and restore of the simulations, just to name a few.

3.2 Additional Development

Although the EO provides many components that make the problem of writing a new GP strategy from scratch very easy, we have also developed and extended some of the functionality that we have considered important for the solution of this project. In particular, we have written new *Checkpoints* and *Monitors* to print out relevant information about the population.

We have considered important the tracking of the average, minimum and maximum depth of the individuals at each generation, which have been implementing by deriving new classes from the `eoStat` base class. Consumed User-CPU is another information that we have decided to keep track of, which is important when deciding if a particular new functionality is worth to use according to the time consumed-improvement in performance balance. Also, a simple generation and execution number monitors have been developed, to save all results in a text file for latter processing.

To study the evolution of the population, we have also developed a monitor to measure the diversity of the population, by measuring, at each generation, the number of individuals that are unique.

Finally, other monitors such as a measure of the standard deviation of the fitness evaluation, and a string representation of the best individual have been included in the code.

3.3 Code Structure

Using the EO library, new development has been limited to the implementation of the symbol set adapted for this problem, as well as the fitness function evaluation. Additionally, as we have mentioned in previous section, new *monitors* and *checkpoints* have been created, aimed at obtaining results to better understand the behavior of the algorithm with different configurations. The code created is all contained under the folder `src`, and it is linked with EO static library to generate the binary file. The code files are:

- `CSVTableParser.hpp` defines and implements a Comma Separated Value (CSV) reader to read files containing training sets.
- `types.hpp` defines templatized types to be used to create the EO objects. It defines as well an enumeration type representing the selection method used.
- `utils.hpp` defines useful methods to be used through the code. Specifically, it defines a method to print the arithmetic expression for the best individual within a population.
- `TreeNodeQuadraticEq.hpp` defines the class representing each node of the tree of our GP. Each node will have an **arity** function that will return the arity of the node, and the functor (**operator()**) method, which recursively evaluate the node.
- `QuadraticEqEvaluator.hpp` defines the method that calculate the fitness value for an individual. It saves internally the training set, and it defines the functor method which return the fitness value for an individual. To calculate the fitness value, it evaluates the individual against the instances of equations in the training set, returning the RMS value of all evaluations.
- `CustomStats.hpp` defines *monitors* and *checkpoints* used to keep track of the evolution of the execution. Specifically, it defines the monitors: `AvgDepthStat, MaxDepthStat, MinDepthStat`, to display the average, maximum and minimum depth of the tree representing the individual within the population, `CPUTimeStat` which returns the user-CPU consumed time, `GenerationNumber,`

ExecutionNumber, which display the execution and generation number on each experiment, FitnessStdDev, which returns the standard deviation of the fitness values of the population, PrintBestSolution, PrintBestSolutionDepth which print out the best individual for a given generation as well as its depth, and finally DifferentFitness, which implements a measure of diversity within the population.

- ResultsAccumulator.hpp implements a class which keeps track of the partial results of each execution and print out a summary for all executions, calculating SR and MBF metrics, as well as the statistic of the number of generation needed to achieve a solution and the consumed User-CPU time.
- OptionParser.hpp defines the class which parses either the command-line arguments or the configuration file to specify the parameters of the experiment. It uses Boost program option library.
- main.cpp This file contains the main method, which, after reading the command line parameter and loading the training set, creates the necessary object and call the evolutionary engine to get the results.

The execution flow for our solution is similar to those we wrote for previous problem. main method first reads parameters either from command line or from a configuration file, using the OptionParser. Once we have the configuration for the experiment, it prints on the screen the parameter of the execution, and start creating the object necessary to run the simulation. Firstly, it creates the symbols, both from terminal and function set, which will be used by the initialization and transformation operators. As we have discussed, we have tested two symbol sets, one *simplified*, with the minimum set of node that we would need in theory to get the solution, and the *extended* one, with the symbols we naturally choose to solve the quadratic equation problem. After defining the symbol set we create a eoGpDepthInitializer object, which is used to initialize the population with a maximum depth.

Then, we continue with the transformation operators, creating objects such as eoSubtreeXOver for recombination, and eoBranch, eoHoist, eoCollapseSubtree, eoExpansion and finally eoPointMutation for mutation, that will be used to create the global mutation operator object, eoPropCombinedMonOp, according to the probabilities defined in the configuration file. Finally, both the crossover and mutation operators are combined in eoSGATransform object, with their respective probabilities.

The training set is read using the CSVTableParser class, and the object created will be passed to the QuadraticEqEvaluator object, which will calculate the fitness of the individuals. For parent selection, we will use eoDetTournamentSelect and eoSelectMany classes. Finally, for survival selection, once off-springs have been created, we will use one of the following objects, eoCommaReplacement, eoPlusReplacement, eoDeterministicSaDReplacement and eoSSGAWorseReplacement classes, according to the selection method we have defined in the configuration.

Afterwards, we define the stopping points and monitors of the execution. For stopping criteria, we have used eoGenContinue and eoFitContinue classes, which respectively, will stop the execution until the maximum number of generations have been reached or a solution (fitness evaluation is null) has been found. All check points and monitors objects previously described are also created in this part of the code.

Once we have created the object we need to run a simulation, we create a ResultsAccumulator object, which will save the partial results of each execution and will generate statistics. Now, the following step is to create the code that will run the number of executions defined in the configuration file. To do that, for each execution, we create a new population, reset some of the monitors that need to be reset on each execution, initialize the individuals of the population and create the eoEasyEA object passing as argument the transformation operator, stopping points, monitors and checkpoints as well as selection method. This object will run the GP strategy. Once the execution of one experiment is finished, the

partial results stored in the monitors are updated using the `ResultsAccumulator` object, and then we loop until we complete all the executions.

Finally, once all executions have been run, we write results to a file.

4 Results

Following the explanation of the implementation of the problem, we have to study the behavior of the algorithm. To fine-tune the parameters, and operators, we have run simulations with the simplified symbol set, comparing the results of each experiment over 30 executions of the program. Once identified the best configuration for the problem, we try to solve it by including the whole set of symbols, what we have called the *extended set*.

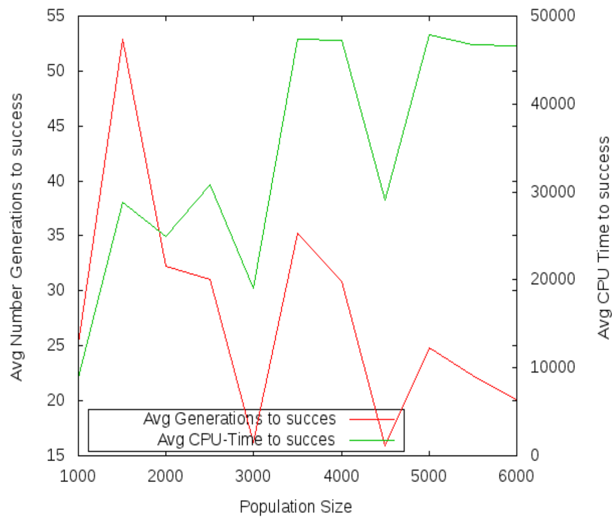
4.1 Varying size of population

As we verified in previous activities, the size of the population is an important parameter for a EC strategy, and the optimal configuration, in terms of CPU resources consumed and results achieved, might vary from problem to problem. Some algorithms perform better on small populations over a great number of generations. Others favors big populations evolving over a small number of generations. In order to check for this dependency on the population size, we have set several experiments with population sizes from 500 to 6000 in steps of 500 individuals. Experiments for this results can be found in configuration files: `simplifiedpopSize500.cfg`,..., `simplifiedpopSize6000.cfg`

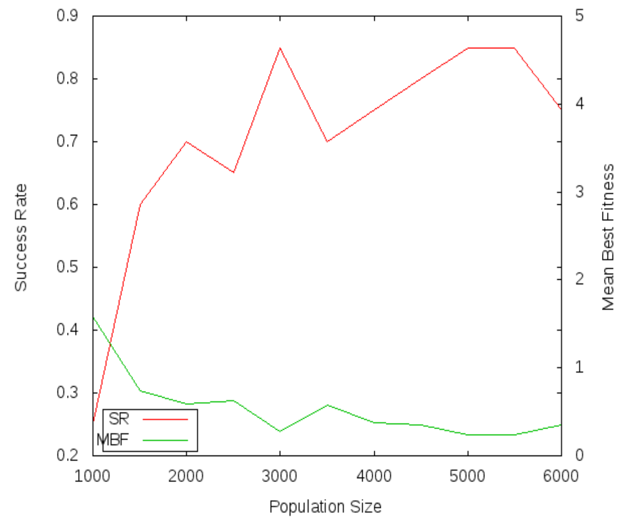
One of our first steps to calibrate the configuration of the experiments is thus to find the impact that the population size has in the final results as well as in the performance. We will study how the population size affects to the *Success Rate (SR)* as well as in the *Mean Best Fitness (MBF)* metrics. Observe that, since this is a problem in which a known solution exists, SR metric is indeed more significant since it states the number of times the strategy found the right solution. MBF metric is more meaningful for problems where no known solution exists, and thus, there is nothing optimal to compare against. However, in this problem, we can use anyway the MBF metrics as a way to check how close to the optimal solution we are in case we do not converge to it. In figure (1), we can see the dependence between the number of individuals in the population and the CPU resources consumed to achieve a solution, in terms of number of generations and user CPU-Time. It is noteworthy to say that for this graph we have only considered executions which finished successfully, skipping those experiments which did not converge to the solution. On the right-side of the figure we have plotted the *SR* and *MBF* metrics.

As we can see on the figure, filtering out the peaks of the graph (increasing the number of executions per experiments would have reduced this noise), the algorithm has the expected behavior, the greater the population size the better the results. On the other hand, we can appreciate that the average number of generations that the algorithm need to achieve a solution decreases as the population size increases. However, increasing considerably the population size might lead to an important increase in the consumed CPU-Time, since, even if we need fewer generations to achieve the solution, the calculation needed for each generation is greater.

To fine tune our program we should thus observe the resulting SR against the consumed CPU-Time. Figure (2) shows these results and, as we can notice, and disregarding the noise (running the simulations with a smaller step would cast better results), populations between 3000 and 4000 individuals seems to be favored in terms of resources consumed for a given SR



(a) Average number of generations and CPU consumed to achieve solution



(b) SR and MBF statistics

Figure 1: Performance and results with respect to the population size

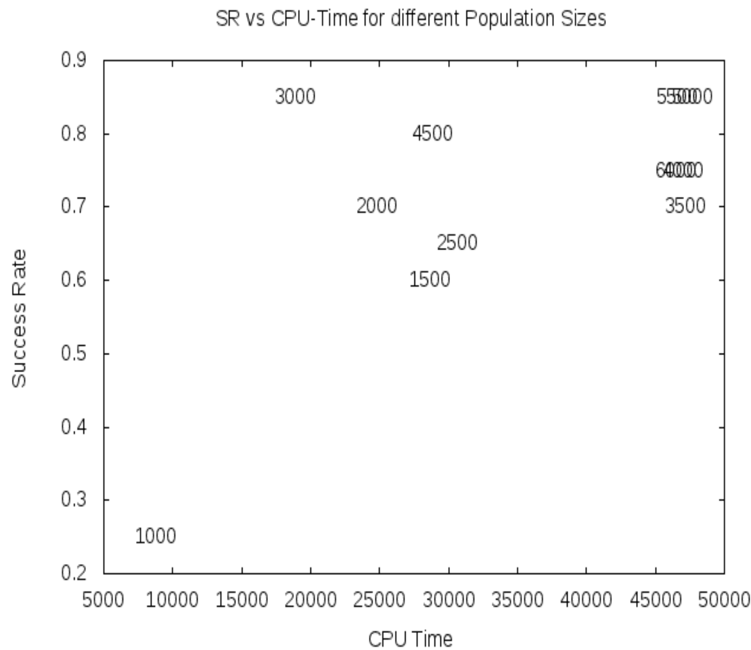


Figure 2: Success Rate vs Consumed CPU time for different population sizes

4.2 Mutation and Recombination Probabilities

4.2.1 Mutation Probabilities

In order to see the effects of the different mutation operators, we have run simulations varying the probability of each of the five types of mutation. Each simulation still contains a mixture of the five operators, but the one that we have to test is assigned a probability of 0.9, and the others 0.025. For the sake of clarity, the 0.9 probability is not the real probability of having a mutation using that operator, but the probability that, given that the current individual is affected by the mutation operator, the applied mutation operator is the one whose effects we want to study.

The files executing for these experiments are those with the prefix `simplified-mutation-*.cfg`, where `*` can be substituted by each mutation name `branch`, `collapse`, `expansion`, `hoist` and `point`. Table (1) shows the results for different probabilities assigned to each method. For each experiment, we obtain the Success Rate (SR) and Mean Best Fitness (MBF), as well as the performance metrics Average Number of Generations (ANG) and Standard Deviation of Number of Generations (SNG). As we can see, *branch* and *expansion* mutation are the ones that seems to perform better in this kind of problems. This was expected since, as we explained in previous section, both methods are similar, being the only difference that one replace the tree under the randomly selected node, and the other one replace the node it-self along with the tree under it. After this tie in mutation operator, *point* mutation has been the second best operator. Finally, *hoist* and *collapse* operators are the worst performers, both of them working in a similar way, pruning or reducing the tree representing a solution under a randomly selected node. That is, this problem benefits from big trees rather than smaller ones. As expected, it is also important to remark that the best operators requires also fewer generations to achieve the solution, except for the *point* operator, which, although is less efficient than *branch* and *expansion*, for those cases when it succeed, the generations needed to achieve the solution are fewer. Using these results we have used for the purpose of resolve our problem, a mutation operator which combines *branch*, *expansion* and *point* operator with probabilities of 0.425, 0.425 and 0.15.

Branch	Hoist	Collapse	Expansion	Point	SR	MBF	ANG	SNG
0.90	0.025	0.025	0.025	0.025	0.833	0.301	21.8	23.503
0.025	0.90	0.025	0.025	0.025	0.233	1.615	39.857	41.670
0.025	0.025	0.90	0.025	0.025	0.333	1.307	48	49.63
0.025	0.025	0.025	0.90	0.025	0.866	0.221	21.077	36.557
0.025	0.025	0.025	0.025	0.90	0.466	1.048	15.357	18.5

Table 1: Performance for each mutation operator. Each line summarizes the experiment of testing each mutation method. Probability of each mutation operator are on the left-side of the table. Showed results are metrics such as SR, MBF as well as performance metrics such as ANG and SNG until convergence, only for those executions which returned the right solution.

On the other hand, we have tested the behavior of the algorithm varying the global mutation probability itself, and not the relative probability of choosing one particular mutation method. Table (2) shows the results for these experiments. Configuration files to reproduce this experiments are saved with names `simplified-mutation-005.cfg`, `simplified-mutation-025.cfg`, `simplified-mutation-045.cfg`, `simplified-mutation-065.cfg`, `simplified-mutation-085.cfg` and `simplified-mutation-095.cfg`. All experiments have been done with 0.425 of mutation probability for *branch* mutation, 0.425 for *expansion* method and 0.15 for the *point* mutation operator.

Results are self-explanatory. The higher the mutation probability, the better the results, achieving a SR of 1.0 when using mutation probabilities near 1.0. Although in text books such as [1] and [3] it is mentioned that *mutation probability* should be kept a low number, given the destructive effect of mutation and crossover operator, in our experiment we have found quite the opposite effect, being required high probabilities of mutation to achieve better results. One of the causes of this effect might be the diversity within the population. We have tracked the diversity of each generation and we have observed that after a few generation the *diversity ratio*, the number of unique solutions within the population, gets really low values, being the population dominated with just a few well-fitted individual and discarding, after each mutation, everyone else because of most of the individuals are not as fitted as the existing parents. The introduction of a high probability of mutation would thus help to create diversity at each generation given that the initial selection method tested is quite elitist. Other configurations with different selection methods will be also tested later on.

Mutation Prob	SR	MBF	ANG	SNG
0.05	0.23	1.66	19.28	22.23
0.25	0.56	0.83	42.17	41.75
0.45	0.66	0.66	53.3	53.07
0.65	0.66	0.55	26.35	31.69
0.85	0.76	0.38	24.47	24.67
0.95	1.00	0.00	35.43	35.62

Table 2: Impact of the mutation probability on the evolutionary process. Metrics such as SR and MBF, along with performance metrics to measure the average number of generations to achieve mutation are displayed for each of the mutation probabilities.

4.2.2 Recombination Probabilities

We have ran similar experiments to the ones executed in the above section to test the impact of recombination probability on the evolutionary process. Results, showed in table (3) here are not quite clear. Intuitively it seems that the greater the recombination probability the best the SR and MBF metrics. However the improvement on these metrics is not so neat as for the mutation probability. On the other hand the dispersion on the average number of generations to convergence are also erratic, and do not follow a clear trend. Results on this section have been obtained after executing scenario files: `simplified-recombination-*.cfg`, where `*` is a three digit number with the probability of recombination which has been tested.

Recombination Prob	SR	MBF	ANG	SNG
0.05	0.633	0.667	17.842	31.042
0.25	0.7	0.571	24.333	36.915
0.45	0.667	0.636	40	50.21
0.65	0.833	0.321	25.08	30.279
0.85	0.8	0.365	24.916	25.749
0.95	0.8	0.341	31.042	32.002

Table 3: Impact of the recombination probability on the evolutionary process.

The conclusion after this study of mutation and recombination probability is that, contrary to the standard literature on GP, mutation operator seems to be more useful on this problem, or at least, on this configuration. Thinking of it, recombination is the process by which we create new individuals using current existing genetic information from other creatures within the population, while mutation is the ingredient that adds really new genetic material to the population. What these results are telling us is that the strategy is evolving, mainly, thanks to this random addition of new genetic material, more than the mixture of the current individual features. Observe however that this effect might not be viewed as a weird result, but it rather depends on the kind of problem we are dealing with. The evaluation of arithmetic expressions as a tree might be the root of this behavior. A small change in a branch of the tree might lead to a completely different individual, which is very distant from the previous one in terms of fitness value, so, recombinations and mutations can have a very destructive way of doing evolution, even if they act locally at a certain node. In other problems such as GA, we can design adapted cross-over and mutation operators whose application create individuals that are relatively close to the original way. These operators are usually specific for the representation of the individual we have used to solve the problem (binary string, floats, permutations...), as we saw on *Problem 1*. Even if we can see, as in table (2), that some mutation operators work better for this problem, this may be not because of they create individuals close to the originals ones, but because the kind of new features it introduces is better suited for a given problem.

On the other hand, and given that the erratic behavior showed in this experiment, we should further investigate this effects within a greater number of executions for each experiment to try to filter out the effects of randomness.

4.3 Selection

We have also experienced with several selection methods, such as (μ, λ), $(\mu + \lambda)$, *Survive and Die* and *Steady State*. The first two methods are the ones used in ES, and were discussed in previous problem set. *Survive and Die* selection method selects the best individuals within a population, and discard the worst, according to some predefined proportion. This process is applied both to parent and offspring

populations. Then, the best individuals from parents and off-springs are included in the final population, along with the individual that were not discarded, up to the number of individuals to complete the right population size. Finally, *Steady State* methods are typical within the GP paradigm. Instead of using a generational replacement, using two populations, parents and off-springs as in the previous schema, *Steady State* selection method uses the same population for parents and off-springs, writing the newly created individuals to the same parent population were individuals are selected for crossover. The advantages of using this method is that is more memory-efficient, since we have to maintain only one population at a time, and this is an important issue when dealing with very large population of million of individuals, typical in some GP problems. Additionally, it is thought to increase the diversity within the population given the immediate availability of eventually superior individuals, opposite to the generational approach. These *Steady State* advantages emerge only above a given population size, and for small number of individuals results do not outperform the previous methods.

Results for this experiment are shown in table (4), and they have been producing by executing the scenarios described in configuration files: `simplified-selection-*.cfg`, where `*` is the selection method used, `comma`, `plus`, `survivedie`, `steady-state`.

Selection method	SR	MBF	ANG	SNG
(μ, λ)	0.3	1.592	66.555	56.68
$(\mu + \lambda)$	0.733	0.468	38.4545	34.55
Survive and Die	0.8	0.286	31.167	28.567
Steady-State	0.767	0.882	59.73	51.636

Table 4: Impact of the selection method.

For the first method, (μ, λ), we have drawn pretty poor results. SR and MBF metrics are bad, and the number of generations needed, on average, to achieve convergence, for the cases we actually achieve it, are large. One important issue to highlight is that there might be chances that *involution* or *backward evolution*, appears, having generations that have worst fitness values for the best individuals than previous ones. This result is indeed quite intuitive since we do not save the parents from one generation to another, but the whole generation is replaced by the new best off-springs, even if these are of a poorer quality than parents. On the other hand, and since we do not save the best individuals from generation to generation, when we use high probabilities of recombination and mutation the population diversity metrics tend to be quite high, being most of the time over 50%. Given the poor convergence results of this method, which stems from the lack of elitism on the selection, and the destructive work of the mutation and crossover operators, we have also run experiments with this selection method with a lower probability of the transformation operators. In this case we have obtained better convergence results, but still not good enough to outperform the next methods.

$(\mu + \lambda)$ introduces some elitism to the selection process, retaining the best individuals from parent population that are able to pass through several generations if their fitness values are good enough. At the same time, this elitism reduces significantly the diversity of the population, which is counterproductive for the selection process, since it limits exploration. However, overall, results are by far much better than for the previous strategy, achieving the right result more than half of the times. Number of generations needed on average to converge to the right solution are also smaller. Finally, although with this method we can have the *involution* or *backward evolution* effect previously discussed, the occurrence of this issue is by far less frequent than in the (μ, λ) schema. Studying more in detail the execution log of this selection method, we quickly understand the well-known property of *punctuated equilibrium*. This property states that most of the time, evolution is kept frozen for many generations, where individuals hardly evolve and improve, then all of a sudden, a new gen appears within the population which introduces a better adaptation of the individuals, and the population increase substantially its diversity as this gen

is spread to all individuals, at the same time that the population achieves better fitness values. After this momentaneous increase in diversity, once the gen has been spread to most of the individuals population stabilize, decreasing diversity and getting stuck in a fitness level until a new innovative gen comes in.

Survive and Die selection method improves the $(\mu + \lambda)$ results. By introducing a stronger elitism, since we guarantee that best individuals from parents and off-springs are passed through generations, we achieve improvements in the SR and MBF metrics, as well as a reduction in the average number of generations needed for convergence. The involution effect is thus removed by this elitist method. One counterintuitive effect is that population diversity is, on average, bigger than in the $(\mu + \lambda)$ schema, even if the method is more elitist. Population diversity hardly falls below 10%, while in the $(\mu + \lambda)$ diversity ratio of 1% are common after a few generations with no big improvements in the individuals.

Finally, *Steady-State* method cast also results as good as $(\mu + \lambda)$. However, as we can see if study more in detail the results of the executions, the evolutionary process seems to be rather erratic. Diversity of the population is always kept high, but the best individual of each generation has not a decreasing monotonic fitness, as we would expect from an algorithm which heads to the optimal value. Instead, many cases of evolution and involution are interleaved during the evolutionary process, until, all of a sudden, a solution is found in one of the generations. Thus, at least for the configuration we have tested, even if the method has a decent SR statistic, we have discarded this selection method as it does not correctly apply selection pressure to the individuals to achieve a uniform and monotonous convergent evolution. As we discussed previously, the population size may be too small for this method to properly shows its virtues.

Table (5) shows basic statistics on the *diversity coefficient*, accumulated on the 30 executions run for this experiment. Recall, as explained previously, that *diversity coefficient* is measured as the ratio of the number of individuals with unique fitness evaluation over the number of individuals within the population. Even if the first and second moment of the *diversity coefficient* are rough approximations to describe the behavior of the diversity of the population for each selection method, results are representative enough to arise some basic conclusions. As we discussed in the previous paragraph, (μ, λ) maintain a good diversity within the population, although the lack of selection pressure or *elitism* prevents the algorithm to evolve in a timely manner to a good solution. Diversity for the $(\mu + \lambda)$ selection method is quite low, which can make the process to get stuck on local minima, delaying the evolutionary process, since we will need more generations to come up with a solution. Observe that the *standard deviation* of the *diversity coefficient* is of the same magnitude as the *average*. In practice, the behavior of the *diversity coefficient* in a $(\mu + \lambda)$ simulation is as follows, fist it starts with high diversity, typically around 30% but, as a good solutions appear, the population quickly reproduce its genes, and the *diversity coefficient* falls below a threshold of 10% that prevents the algorithm to explore a solution on a broader front, being many times reduced to diversities of 0.01%. For this reason, even if we have obtained good results with $(\mu + \lambda)$, as we have showed on table (4), they can be improved just by increasing the diversity. *Survive and Die* method solves this problem, as we can see, the *average* diversity for this method is higher than for $(\mu + \lambda)$, but lower than for (μ, λ), which in practice means that it explores the solutions on a wider front than $(\mu + \lambda)$ and at the same time retains good solutions, focusing the evolution on the right path, oppositely to (μ, λ). Finally, for *Steady-State* the diversity of the population is always kept high, but instead, the population does not head uniformly towards convergence of the optimal solution, but keep exploring individuals until all of a sudden finds, by a random transformation, the solution.

Figure (3) plots these results. The picture shows the evolution of the *diversity coefficient* through a number of generations for the four selection method for a given random execution. The figure intuitively resumes the results explained on the paragraph above.

After the above results, we thus recommend the use of the $(\mu + \lambda)$ or *Survive and Die* strategies, since they have been proved to have a decent enough SR statistic, and the behavior of the algorithm seems to

Selection method	Average Diversity	Std. Diversity
(μ, λ)	0.506	0.119
$(\mu + \lambda)$	0.122	0.151
Survive and Die	0.321	0.120
Steady-State	0.522	0.039

Table 5: Diversity within the population over the 30 executions.

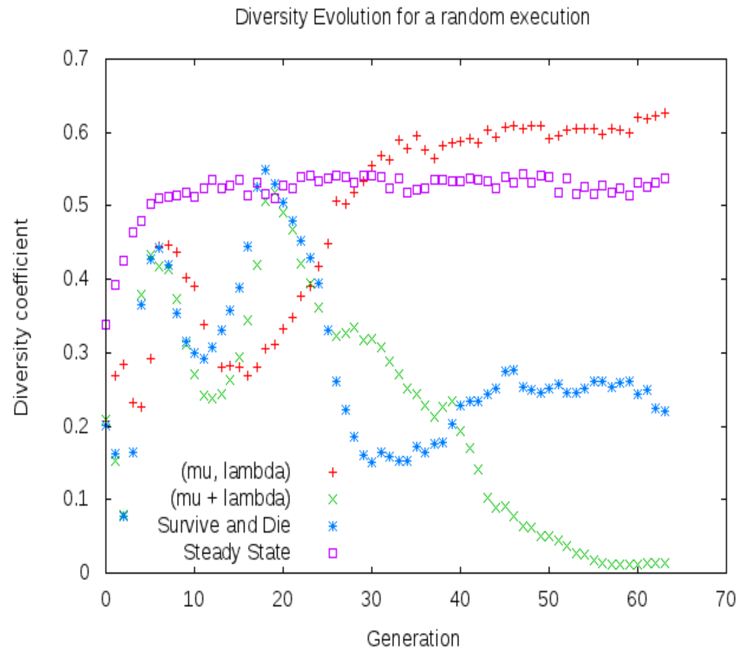


Figure 3: Diversity evolution for different selection method

be better fitted, being more or less elitist, but always advancing monotonously to the optimal solution.

4.4 Training set

In this section we will study the impact of the size of the training set in our GP strategy. As we explained in the implementation section, each individual (tree) is valued against the set of coefficient within the training set. Then, the fitness value is calculated as the root mean square of each evaluation. The size of the training set will thus have impact on two aspects, learning process and performance.

The training set should be diverse enough so that it contains examples of different kind of equations, with a double real solution, imaginary solutions and real solutions. Also, it should contain examples of degenerated equations, i.e, with some coefficients being null. Although in other learning problems big training sets are needed in order to infer a general rule or knowledge, this is not the case for the quadratic equation problem. In other kind of learning problems, the *over-learning* issue might arise when the training set is big, and the results (rules, knowledge inferred) are *over-fitted* to those examples and fail to work in the test set. In our strategy this is not the case since indeed, given that what we are looking for is a general formula that gives the exact results on every instance of the problem, what we precisely need is an *over-fitting*, that is, that our algorithm perfectly matches the right results for all instances of the training set. Hence, if we choose a diverse enough training set which includes cases from all kind of second order equations, the *over-fitting* issue will not be a problem, but a desired result.

On the other hand, the size of the training set impacts directly on the performance of the algorithm, since each individual must be evaluated against every instance of this set, and thus, the larger the set, the longer it takes the evaluation of the population. Accordingly, we should study the dependency between the size of the training set and the performance and SR metrics to see which size better adapts this problem. Observe that, to measure performance in this case, we have to compare not only the number of generations needed on average to achieve the result, but the consumed user-cpu time on average for each execution, since the longer the training set, the longer it will take in absolute time. Table (6) shows results. As we can see, with a well diversified training set we can achieve better results, in terms of SR, as well as less User-CPU time consumed, achieving convergence in fewer generations with small training sets. Figure (4) plots the same results

Training Set Size	SR	ANG	Mean CPU-Time
50	0.967	16.621	10904.8
75	0.967	14.551	10064.1
100	0.866	21.270	17780.8
120	0.766	23.869	21757.8
150	0.766	22.434	22597.8
200	0.700	21.857	27847.6

Table 6: Results for training sets of different sizes

Figure 4: Impact of the training set size on SR and User-CPU time

4.5 Tree Depth Evolution

As we have discussed on the *bloat effect* section, in our current implementation trees do not seem to grow unlimitedly, and, even if we put means to limit the size of the solution candidates, these limit size are not usually reached.Through all the experiments we have ran, we have been used a limit depth of 20 levels, which is, by far, much more than we might need to achieve the correct solution, specially for the simplified function set. Figure (5) shows, for each execution of the `simplified-selection-survive-die.cfg` experiment, the minimum, average, maximum depth of the individuals within the population snapshot at the last generation, as well as the depth of the best individual. As we can observe, only 8 out of 30 executions, or about 26% of the times, the maximum depth is achieved, being the average depth of the population always below 12 levels.

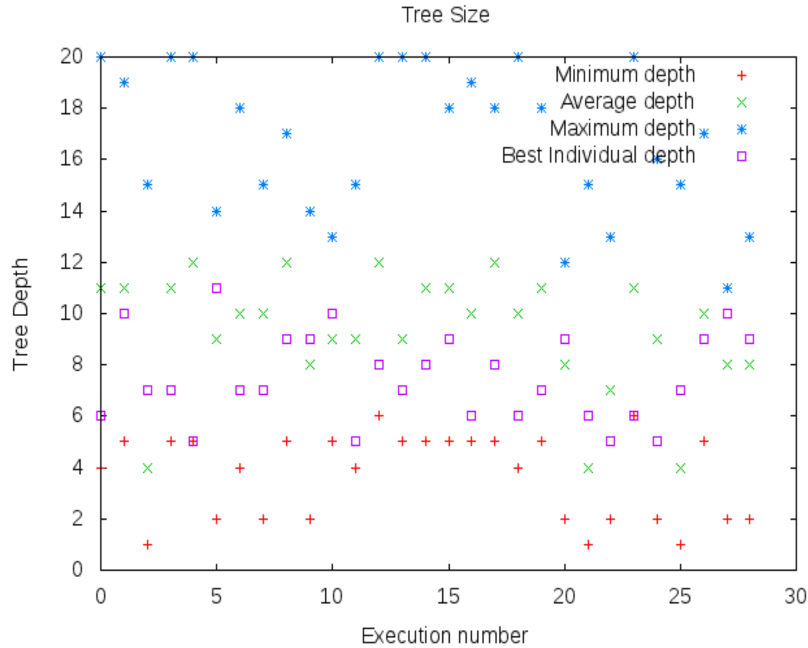


Figure 5: Statistics for size of the individuals at the last generation of each execution

The ‘quality’ of the final solution can be somehow measured in terms of the depth of the individual. For the simplified case, in which we look for the solution in equation (3), represented as a tree on figure (6), we only need a tree with five or six levels. Table (7) shows the statistics for the distribution of the depth of the best individual. As we can see, we reach in some of the execution the most simplified expression for the solution, although on average, solutions have depth of 7.46, which although is not the best of the cases, is a solution equally valid and quite small. Should have we used an optimization tree technique as previously mentioned, our solution would have been reached always with the minimum depth, but probably the effort is not worth given the quality of the solution achieved with the naive implementation.

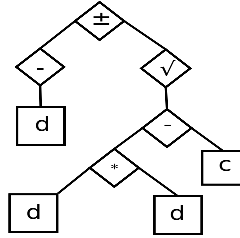


Figure 6: Solution for the simplified symbol set in a tree representation

Statistic	Value
minimum	5
0.25 percentile	6
0.5 percentile	7
Mean	7.46
0.75 percentile	9
maximum	11
std	1.717

Table 7: Distribution of depth on the best individuals of each execution

4.6 Summary of solution with simplified symbol set

After all experiments executed in order to test the behavior of the algorithm under different configuration, we have come up with a solution which could be consider as final for the purpose of this problem set.

The solution can be run using the configuration file `simplified-solution.cfg`, and it uses a training set of just 50 equations. The experiment is run 30 times to test the robustness of the results. The *symbol set* used is the simplified version $\{d, c, +, -, \cdot, \surd\}$, assuming the canonical representation of the quadratic equation with $a = 1$. Depth has been limited to 20 levels, knowing, as we saw, that the solution can be achieved with a tree of just 5 levels of depth. The population size is $\mu = 3000$, and the maximum number of generations to achieve a solution is limited to 150. Selection method used is *Survive and Die*, and the number of off-springs created at each generation is $\lambda = 2\mu$. Mutation rate is $p_{mutation} = 0.8$, and recombination rate $p_{recombination} = 0.85$. *Branch*, *expansion* and *point* mutation have been used as mutator operators.

The SR achieved is 0.967, that is, we have found the right solution $-d \pm \sqrt{d^2 - c} = -\frac{b}{2} \pm \frac{1}{2}\sqrt{b^2 - 4c}$ in all cases except for one out of thirty. The MBF is 0.048, and the only case where the solution was not found after 150 generation, the RMS fitness was 1.4689. Should we have let the algorithm run for more than 150 generations, we would have likely achieved a solution. But with such a good SR we think it is not worth. On average, the algorithm found the solution after 16.62 generations, with a standard deviation of 17.32, and consuming 10.904 seconds of CPU, although this number is unimportant since it varies largely across machines.

4.7 Solution with a complete symbol set

After tuning our GP with the best parameters for the reduced symbol set $d, c, +, -, \cdot, \surd$, we have used this configuration in order to try to solve the quadratic equation problem with the extended symbol set $a, b, c, +, -, \cdot, \div, \surd$. Unfortunately, with this extended symbol set, we have not been able to find the solution in any of the executions. We have tried to tune the experiments with different population sizes, different number of off-springs and several selection method, but in any of the configuration we have reached a single right solution. One example of configuration file to solve the problem with the extended symbol set is `complete-test-c.cfg`.

However, it is noteworthy to mention that even if we have not reached the solution, i.e. $SR = 0.0$, the MBF metrics have been always very close to zero. For example, using the above-mentioned configuration file, we have achieved a $MBF = 0.074$ valuing the candidate solution against a set of 50 sample quadratic equation. Unsurprisingly, even if we have not found the exact solution, we have found a very accurate approximation, up to $10e^{-2}$ precision.

Studying more in detail the solution suggested by the GP we realize that it is always the same, and it is showed in equation (5). Given the good approximation, and the suspicious resulting solution, we have investigated and surprisingly the solution is similar to the *continued fraction expansion* of the quadratic equation solution. Continued fraction representation are algebraic expression obtained through an iterative process by which we can represent a number of another more complex algebraic expression, just by means of sum and subtraction of fractions (possibly with infinite terms).

$$x = \frac{-c}{b - \frac{c}{b - \frac{c}{b - \frac{\sqrt{2}c}{b}}}} \tag{5}$$

To check this hypothesis, we can easily derive a continued fraction expression for the canonical quadratic equation $x^2 + bx + c = 0$. We can group terms and express the equation as $x(x + b) = -c$,

hence $x = \frac{-c}{b+x}$. Using the idea of continued fraction iteration, we can thus expand this expression to infinite, resulting in equation (6):

$$x = \frac{-c}{b+x} = \frac{-c}{b+\frac{-c}{b+x}} = \frac{-c}{b+\frac{-c}{b+\frac{-c}{b+\dots}}} \tag{6}$$

We therefore conclude that equation (5) given as the solution of our GP strategy is just the truncated series of equation (6), which is the continued fraction expression for the solution of the quadratic equation. Even if we have not been able to find the exact expression, we have came up with a good aproximation which have a solid and logic justification.

5 Summary and Improvements

Following all previous experiments, we have finally obtained the results for the quadratic equation problem. On our first attempt, with a simplified symbol set, we have found the exact solution, while, when attacking the problem with the extended symbol set, we have came up with the continued fraction approximation of the solution, instead of finding the well-known close-formula. One important conclusion to derive from this is that, whichever problem we might face with Artificial Intelligence techniques, a good approach is first to think about the problem, about possible approximate solutions as well as simplifications, which might enormously ease the process of finding a solution. Many times, spending some time exploring simpler alternatives to find the solution might be worth of it.

Several mutation method have been tested, of which just two were really useful, mainly *branch* and *expansion* mutation, and another third, *point* mutation, which added some value as a complementary method. The results have been proved to be highly sensitive to the mutation probability, and, contrary to the common GP configuration, specially those cited in [3] and [1], the higher the mutation probability the better the overall results.

Four selection methods have been implemented, and the results favored those elitist methods which at the same time maintained a minimum diversity within the population. *Survive and die* method has been chosen as the preferred method, followed by a $(\mu + \lambda)$ schema. Other typical methods such as *Steady State*, widely used in GP, has been discarded because, even if we achieved good SR, it was worse performer than the previous one. As we discussed, SSGP is very sensitive to the size of the population, and we have only tested this method with populations up to 5000 individuals. It might be the case that with considerably bigger populations, being in GP usual populations of hundred of thousands, the goodness of SSGP method start to emerge.

Being GP more a *machine learning* method than an optimization tool, we have to use a training set with examples on which the inferred knowledge is based. This particular problem, with a closed-form solution, have not required a big training set, obtaining very good results with 30 or 50 instance problems. Moreover, the issue of *over-fitting* or *over-learning* does not seem to apply for this application, since the aim of the problem is exactly that, finding an expression that exactly gives the right solution for all instances within the training set.

6 Appendix

A Program Compilation

Program building process has be done using the standard make gnu tool. To compile and link the program just invoke the `make` command on the root directory. The binaries will be found either under the `bin/debug` or `bin/release` directories, depending on the `dbg` flag specified at `config.mk` file. To

link the program and generate the binary, we need the Boost and EODev Libraries properly installed in our system. The `libraries.mk` file contains the list of libraries that we need for the linking process.

B Program Execution

The program `GP.bin` is invoked from the command line, passing all required parameters needed to the execution of the algorithm. For a comprehensive list of parameters, we can invoke:

```
bin/release/GP.bin --help
Allowed options:
  --conf_file arg                       Path to the configuration file. If
                                        omitted, arguments are read from
                                        command line
  --popSize arg (=2500)                 Population size
  --selection arg (=plus)               Selection Method: comma, plus,
                                        survive_die,steady_state_worst
  --lambdaRate arg (=2)                 Lambda rate for (mu + lambda)
  --maxGen arg (=500)                   Maximum number of generations
  --tolerance arg (=0.01)               Tolerance for termination condition
  --maxDepth arg (=20)                  Maximum Tree Depth
  --maxDepthInit arg (=10)              Maximum Tree Depth at initialization
  --tournamentSize arg (=2)             Tournament size
  --mutationRate arg (=0.75)            Mutation Rate
  --branchMutationRate arg (=0.65)      Branch Mutation Rate
  --hoistMutation arg (=0.14)           Hoist Mutation Rate
  --collapseMutation arg (=0.01)        Collapse Mutation Rate
  --expansionMutation arg (=0.15)       Expansion Mutation Rate
  --pointMutation arg (=0.15)           Point Mutation Rate
  --xOverRate arg (=0.85)               Recombination Rate
  --executions arg (=3)                 Number of independent executions
  --functionSet arg (=simplified)       Terminal and function set to be used
                                        [extended|simplified]
  --output arg (=results)               File where results are written to
  --trainingSet arg (=Ec_2do_Grado.txt) File with equation coefficients for the
                                        training set
  --help                                print out help message
```

All experiments run to create this report have been saved in a configuration file with suffix `.cfg`, saved under the `results` folder. Refer these files for further examples.

C Acronyms

ANG Average Number of Generations 12

CSV Comma Separated Value 8

EC Evolutionary Computing 2

EO Evolving Objects 7

ES Evolutionary Strategy 2

GA Genetic Algorithm 2

GP Genetic Programming 2

GPU Graphic Processing Unit 7

MBF Mean Best Fitness 10

RMS Root Mean Square 4

SNG Standard Deviation of Number of Generations 12

SR Success Rate 10

SSGP Steady-State Genetic Programming 5

References

[1] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computation.* Natural computing series. Springer-Verlag, 2003.

[2] Maarten Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. *Artificial Evolution*, 2310:829–888, 2002.

[3] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.