Rerunning the Course of Evolutionary Computation

Shawn Saliyev, Nicholas Plucker, and Nicholas Fretiag McPhee
Division of Science and Mathematics
University of Minnesota Morris
Morris, Minnesota, USA 56267
{saliy002, pluck011, mcphee}@morris.umn.edu

Abstract

Evolutionary Computation is an approach to artificial intelligence inspired by biological evolution. In biological evolution, the organisms that are better suited for the environment have a higher chance of surviving and thus making more children. In evolutionary computation, instead of living organisms, we have computer programs, called individuals. These computer programs are implemented with language known as Push, which is a stack-based language designed for evolutionary computation. This is part of a system known as PushGP, which utilizes the Push language to evolve programs. Individuals are tested against the problem and are given an error value. A larger error would imply that the current program performed poorly on the tests. Individuals with lower error values are more likely to be selected as parents for the next generation. The process of evolving populations of individuals over time is called an evolutionary run. In each run, the goal is to evolve a program that solves a specific problem. A run succeeds if an individual is found that can solve the problem, i.e., the individual passes all of the tests and the total error is 0.

The purpose of our research is to identify the impact of random initial populations on the behavior of evolution on different problems. In our research, we created an initial population of 1,000 individuals and used it as the starting point for multiple evolutionary runs on three different problems. We generated a random initial population and used it multiple times on the targeted problem. The data that was gathered from the runs was in the form of evolutionary trees and was uploaded into a graph database for analysis. Preliminary results suggest the initial population can have a substantial impact on some runs, with some initial populations having a substantially higher probability of success than other initial populations. Results also suggest that the trajectory of these evolutionary runs is contingent, i.e., different runs with the same starting point can have very different outcomes.

1 Introduction

Evolutionary Computation is an approach to artificial intelligence inspired by biological evolution. In biological evolution, the organisms that are better suited for the environment have a higher chance of surviving and thus making more children. In evolutionary computation, instead of living organisms, we have computer programs, called individuals. Individuals are then tested against a problem and given an error value. A larger error value would imply that the individual performed poorly on the tests. Individuals with a lower error values are more likely to be selected as parents for the next generation. The process of evolving populations of individuals over time is called an evolutionary run. In each run, the goal is to evolve a program that solves a specific problem. A run succeeds if an individual is found that can solve the problem, i.e., the individual passes all of the tests and the total error is 0.

Every evolutionary run has an initial population which is a first generation of randomly generated programs. The goal of our research was to identify the impact of random initial population on the behaviour of evolutionary runs. This paper presents our methods which we used for identifying the impact and results of our research.

The paper proceeds as following: Section 2 introduces Genetic Programming (GP) and various tools/applications used for our research, Section 3 discusses our methods used for our experiment, Section 4 presents our results and findings from our experiment and Section 5 will conclude this paper.

2 Background

In this section we will briefly cover several concepts central to our research. These include the idea of genetic programming and evolving programs, how individuals (programs) are represented and manipulated in the evolutionary process, and how we select individuals to be parents during evolution.

2.1 Genetic Programming

Genetic Programming (GP) is the type of Evolutionary Computation (EC) that was used in this research. The general goal of EC systems is to stochastically transform solutions into hopefully better versions through generations. This process is inspired by natural evolution and it tries to find the solution by using random mutation, crossover, fitness functions and selection mechanisms.

In GP, the solutions being evolved are computer programs, typically represented in some form designed for evolutionary manipulation. In what are arguably "simpler" systems like Genetic Algorithms (GAs), where individuals are represented as fixed-length bit strings, genetic operations like mutation are reasonably straightforward: One can flip one or a handful of bits, usually without worry that the resulting bit string will in some way be "illegal" or fail to interpretable as a solution. With computer programs, however, small changes often lead to serious problems, especially in typically programming languages designed for human programmers. Removal or substitution of a single punctuation mark in most traditional programming languages will often completely break the system. This makes them poorly suited for evolutionary purposes.

Thus GP systems typically use special representations or languages designed specifically for evolution. These representations allow for mutation and recombination of programs with little or no risk of such changes leading to a program that can't be run. Much GP work has focused on a parse tree representation, where mutation and recombination involve replacing and swapping parts of those trees. Other systems have used circuit-like representations, grammars to generate programs, or register-based Assembly-like languages. In our work we used the Push language, designed specifically for evolutionary computation work; this is described in the next section.

2.2 Push

The Push language was introduced Spector and Robinson in 2002 [1] and has been extended and developed in numerous ways since then.[1] Push is a stack-based language that uses separate stacks for each data type. Push programs can have nested loops and complex control structures like programs written in other languages. To aid in evolution, Push programs can be represented as linear sequences of genes called Plush Genomes. Each gene represents either an instruction (like integer-add or string-length) or some constant (like 7 or "Hello"). *Any* such linear sequence of genes can then be translated into a working Push program, which makes this representation very robust to evolutionary change. The simple syntax and structure of Plush genomes and Push programs make it easier to apply mutation and recombination operators that are used for generating and manipulating programs.

Besides having separate stacks for each data type, there is also an execute stack that controls the execution of a Push program. The execute stack initially contains the Push program to be executed. If the top element in the execute stack is a constant, like an integer value or a string, the system pops that value from the execute stack and pushes it onto the appropriately typed stack. Thus if the top value on the execute stack was the integer 7, for example, then that 7 would be popped from the execute stack and pushed onto the integer stack [2].

[1] See http://pushlanguage.org for additional information and resources.

If the top element on the execute stack is an instruction like integer-add, the system then checks to see if the necessary arguments are available on the appropriate stacks; if they are the instruction is performed, but if they are not, then the instruction is simply popped off the execute stack and ignored. As an example, the string-length instruction takes a single string from the string stack and pushes its length onto the integer stack. If, however, the string stack is empty, then this instruction would be skipped. Similarly, the integer-add instruction acts on the integer stack. If the integer stack has at least 2 integers then it will pop these two elements and push their sum back on the integer stack. However if the integer stack has less than 2 elements, then the integer-add instruction will be ignored and there will be no change to the integer stack.

Here is a simple example of how such a program is executed:

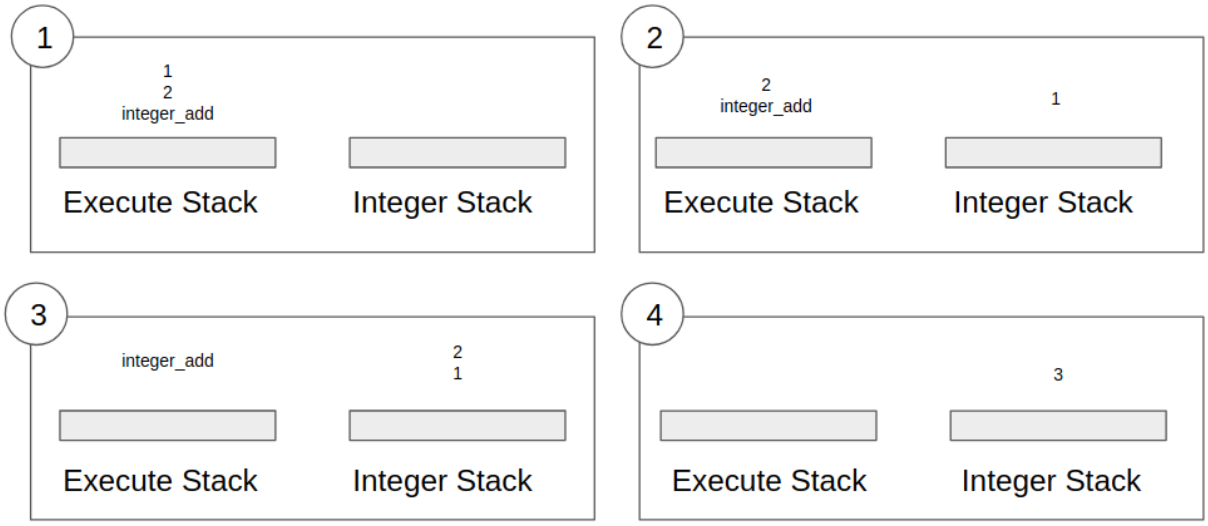

Figure 1: Stack execution

Initially the execute stack contains two integer constants (1 and 2) and one instruction (integer-add). The system will pop the top element from the execute stack and will push it into integer stack as shown in picture 2 of Figure 1. The same process will happen to the next top element of the execute stack (picture 3). After the two integer constants have been moved, the only element left on the execute stack is the integer-add instruction (picture 3). This instruction will simply look at the integer stack, and if there are two integers it will add them (picture 4).

The ability to simply ignore instructions that don't have appropriate arguments is a key part of Push's evolutionary flexibility. These instructions don't fail in some catastrophic way, or attempt to "fix" the problem by the use of some repair mechanism, but are instead simply ignored during execution. They remain in the genome, however, which means that through mutation and

recombination they could later be evaluated in circumstances where they do in fact perform their action.

2.3 Errors, test cases, and Lexicase selection

A key step in any evolutionary computation system is the selection of "parent" programs that are modified or combined to create "child" programs in the next generation.

In most EC systems there are one or more test cases that the evolved solution is tested against. In our runs, for example, we use software synthesis benchmark problems that have hundreds of test cases, each of which generate an error value.

There are a numerous EC selection mechanisms, many of which aggregate these individual test cases into something like a total error. This often favors "mediocre generalist" programs over "specialist" ones that are very good at parts of the problem, but quite bad at others. If there are 10 test cases, for example, a program that is perfect on 9 of the cases (error 0) and has an error of 100 on the tenth will have a total error of 100. A program that has an error of 5 on all 10 test cases will have a total error of 50, and is much more likely to be selected in most systems using aggregate error. Successful evolutionary runs often benefit from being able to build on or combine information from specialists, which requires that such specialists can be selected, even if they're quite weak on some of the test cases.

The selection mechanism we that is used for finding suitable "parent" programs is called **lexicase selection** [3]. This selection mechanism does not aggregate the error outputs from each test case into one single value. Instead, *each* time a selection is performed, it randomly orders the test cases and selects programs that have a best performance on the first test case. Then the algorithm will remove the first test case, and repeat the same process with the next case. This will continue until there is only one individual left or all the test cases are processed.

Here is the pseudocode for lexicase selection:

1. Initialization:
    a. Set entire population of a current generation to be the **list of candidates**.
    b. Set all test cases to be the **list of cases** in a random order.
2. Loop
    a. Based on the first test case from the **list of cases** select all candidates that have best performance compared to any other program in the **list of candidates**.
    b. If the **list of candidates** contains only one individual, then return it.
    c. If the **list of cases** contains only one test case, then randomly choose an individual from **list of candidates** and return it.
    d. Otherwise remove the first test case from the **list of cases** and repeat the loop

2.4 Genetic Operators

Mutation is a type of genetic operator that only involves using one "parent" program. When mutation is happening, the system randomly chooses parts of the program and alters them thereby creating a new mutated program. Another type of genetic operator is called the crossover. During the crossover randomly selected parts from two "parent" programs are used for creating a new program for the new generation.

One could mutate a Plush genome, for example, replacing a gene with a new, randomly generated gene, e.g., replacing the string constant "Hello" with the instruction integer-multiply. Similarly one could remove a few randomly selected genes, or insert a few randomly generated genes at randomly chosen locations.

2.5 PushGP and Clojush

PushGP is a rather generic term for a whole family of evolutionary systems built around the Push programming language. There are PushGP systems written in a variety of programming languages, including PyshGP in Python[2] and propel in Clojure.[3] The work presented here used the Clojush "reference" implementation,[4] a PushGP system in Clojure programming language, utilizing Clojure's facilities for multi-core concurrency. It is really important that Clojure is suited for parallel execution, since typically hundreds of programs need to be tested at the same time.

Clojush also has support for a wide variety of types and instructions, numerous genetic operators, and selection mechanisms such as lexicase selection.

2.6 Individuals

All individuals or programs are generated by using Clojush implementation. Before we could start our evolutionary runs, we needed to create the initial population. Programs in the population are randomly generated and have a really small chance of solving the targeted problem. Individuals will also have all needed constants for the test problem. For example if the problem involves counting vowels in the given string, then individuals will have the vowels as constants.

[2] http://erp12.github.io/pyshgp/
[3] https://github.com/thelmuth/propel
[4] https://github.com/lspector/Clojush

After the initial population is created, all individuals will be evaluated on each of the test cases. Then the selection algorithm will choose a set of parents from the initial generation for creating the new generation.

3 Methods

We generated the initial random population, and did twenty independent runs each with the same target problem. For our target problem we used the Syllables problem from Helmuth and Spector [4]. For this problem the input is a string that might contain letters, spaces and digits. The goal is to count the occurrences of vowels (defined to be 'a','e','i','o','u',or 'y') in the string and print out the number as X in the "The number of syllables is x". When individuals are being generated for the initial population, they all receive constants that are required for the target problem. In our case these constants were vowels, the string "The number of syllables is" and string "aeiouy"[5].

There were 200 test cases for this problem, and each individual is tested on each of these test cases. If program returns the same value as expected by an particular test case it means that error value is 0. Otherwise the error value depends on how different the program's output is from the expected value. The total error is simply the sum of the error values from each test case. If an individual perfectly solves all 200 cases then all the error values are zeroes, so the total error is also 0.

There are some parameters that need to be set before starting an evolutionary run: population size, maximum number of generations, selection operator and genetic operator.

The maximum number of generations for each run was decided to be 300. If run reaches the generation 300 and no individual that can solve the target problem is found, then the run stops. These runs are considered as fail runs. The population size is 1000 individuals per generation. For the selection operator "Lexicase selection" algorithm was used. A new operator called UMAD [5] was used as genetic operator. UMAD operator only involves mutation of the an individual, that is why each individual only has one ancestor.

4 Results

Three of the twenty runs succeeded, where the other 17 failed to find a solution in the given amount of time. The Figure 2 shows the change of the total error over 300 generations for 6 out of 20 evolutionary runs that we did. Three of these runs have successfully found the solution for the target problem. Other three were randomly selected out of 17 failed runs.

[5] https://github.com/lspector/Clojush/blob/master/src/clojush/problems/software/syllables.clj

Since we used UMAD as the genetic operator, we were able to get total errors for all ancestors of the winner individual. For failed runs, since there were no winner, we selected the individual with the smallest total error in the last generation.

We have noticed that all runs have similar total error over the majority of 300 generations. However each run still has its own unique behaviour. An example for that can be run 5 (blue line in Figure 2). The total error for this run sometimes becomes very high, but this individual was still selected by the Lexicase selection. Another run which is represented by the red line has a weird behaviour at the end, that happened due to the fact that there were multiple individuals with the same total error.

Here are the final forms of programs that were able to solve the targeted problem:
Run 2:

```
("The number of syllables is " print_string \e \y \u exec_s (\o in1 \i) (in1 \a s
tring_occurrencesofchar string_occurrencesofchar) (exec_dup (in1 integer_add string_occurr
encesofchar integer_add)) print_integer)
```

Run 6:

```
(string_empty integer_empty \y "The number of syllables is " integer_empty \e int
eger_empty \a \o \i \u print_string exec_dup (exec_do*while (integer_add in1
string_occurrencesofchar)) integer_add print_integer)
```

Run 3:

```
(\o boolean_empty char_iswhitespace \e exec_empty char_stackdepth exec_empty \u
boolean_stackdepth boolean_flush "The number of syllables is " \i \o char_dup_items \o char_dup \a
print_string boolean_empty char_isdigit \i \y \o in1 char_dup exec_rot (exec_dup (char_dup \i
boolean_stackdepth \o char_yankdup char_dup char_stackdepth boolean_yank \y exec_dup
(exec_do*count (integer_mult \a string_replacefirstchar)) \e \a string_frominteger boolean_flush
char_rot string_concat)) string_reverse (exec_yankdup string_nth) string_occurrencesofchar
print_integer)
```

Solutions from run 6 and 2 are way simpler than the solution from run 3. Even with the same initial population, the forms of the final solutions differ. That shows how trajectories of runs are contingent.

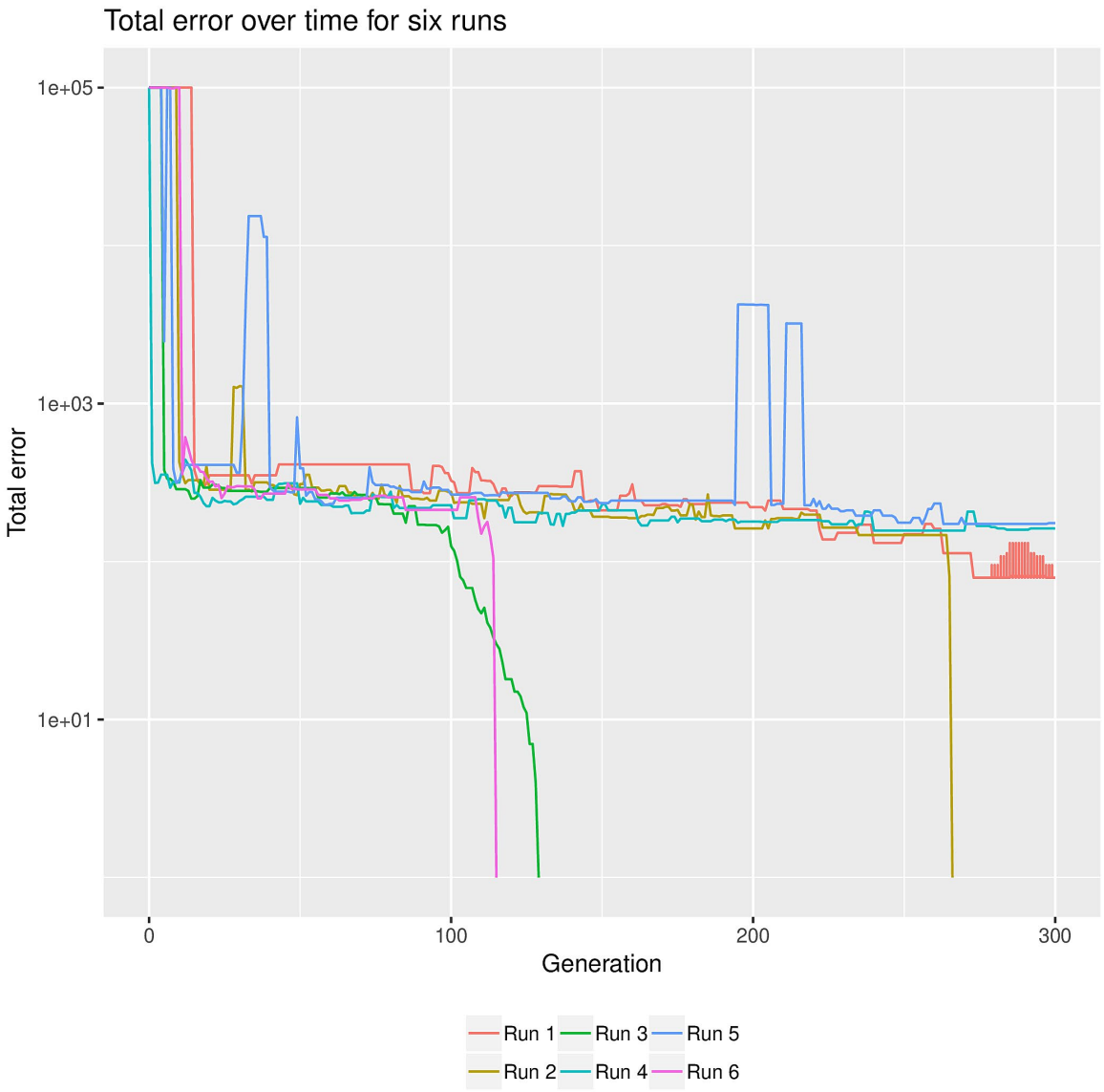

Figure 2: Plot of total error change over time for six runs.

5 Conclusion

Results of all runs showed that even with the same initial population the chances of finding the solution are still random. Only 15% of runs successfully found the solution and each had a

different initial ancestor. However these are results that are based on our evolutionary run parameters. There are other selective mechanisms and genetic operators that can also be used. Different set of parameters might provide results that suggest that there is an impact of initial population on the behaviour of the run.

References

[1] Lee Spector and Alan Robinson. 2002. Genetic programming and autoconstructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3.1, 7-40

[2] Lee Spector and Nicholas Freitag McPhee. 2018. Expressive genetic programming: concepts and applications. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '18), Hernan Aguirre (Ed.). ACM, New York, NY, USA, 977-997.

[3] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2016. The Impact of Hyperselection on Lexicase Selection. In Proceedings of the Genetic and Evolutionary Computation Conference 2016 (GECCO '16), Tobias Friedrich (Ed.). ACM, New York, NY, USA, 717-724.

[4] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15), Sara Silva (Ed.). ACM, New York, NY, USA, 1039-1046

[5] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program synthesis using uniform mutation by addition and deletion. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18), Hernan Aguirre (Ed.). ACM, New York, NY, USA, 1127-1134.