Structured Grammatical Evolution Applied to Program Synthesis

by Andrew H. Zhang

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the

Massachusetts Institute of Technology

May 2019

© 2019 Andrew H. Zhang. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: ____
Department of Electrical Engineering and Computer Science
May 20, 2019

Certified by: ____
Una-May O'Reilly
Principal Research Scientist, MIT CSAIL
Thesis Supervisor
May 20, 2019

Certified by: ____
Erik Hemberg
Research Scientist, MIT CSAIL
Thesis Co-Supervisor
May 20, 2019

Structured Grammatical Evolution Applied to Program Synthesis

By Andrew Zhang

Submitted to the Department of Electrical Engineering and Computer Science

May 28, 2018

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Grammatical Evolution (GE) is an evolutionary algorithm that is gaining popularity due to its ability to solve problems where it would be impossible to explore every solution within a realistic time. Structured Grammatical Evolution (SGE) was developed to overcome some of the shortcomings of GE, such as locality issues as well as wrapping around the genotype to complete the phenotype[2]. In this paper, we apply SGE to program synthesis, where the computer must generate code to solve algorithmic problems. SGE was improved upon, because the current definition of SGE[2] does not work. Given that the solution space is very large for possible codes, we aim to improve the efficiency of GE in converging to the correct solution. We present a method in which to remove cycles from a grammar for SGE, to be able to make sure that a genotype matches to a phenotype with reusing parts of the genotype, and analyze results to shed insight on future improvements.

Table Of Contents

List of Figures

1. Introduction

Coders have used code to automate many different industries, but the development of program synthesis has yet to show as impactful applications. In this paper, we refer to the problem of program synthesis as receiving an algorithmic problem in terms of inputs and outputs, and then writing the code to be able to transform those inputs into outputs. Currently, these algorithm problems are of beginner level, usually found in students' first course in computer science. For context, a problem that we will be examining later in the paper is whether the code can print True if two lists are the reverse of each other. The main reason for the lack of success in this field is the sheer difficulty of the problem. The problems prove too complex for modern A.I. to understand and building code is just as if not even more complex. Also, the enormous number of possible combinations of lines of code makes it impossible to optimize search over the solution space in reasonable time. Thus, program synthesis is well suited for using unsupervised genetic methods to solve. Genetic algorithms generate genotypes (lists of integers) that can be translated into phenotypes (lines of code). We can take the best performing genotypes and merge them together randomly to get better solutions in the next generation, similar to evolution. The algorithm does not need to cognitively understand what the problem is, or what the lines of code mean, but knows how well its current solutions perform, and that combining solutions lead to better performance over time. How the genetic algorithm evolves its solution to the correct answer may provide interesting ideas on how to solve program synthesis for the future. At the moment, the benchmark for genetic algorithms solving common introductory algorithms is not very high, and definitely not usable in real world solutions[4]. Only 4 out of 29 problems were solved consistently[4], and had difficulty in simple problems, such as printing a sequence of numbers.

The genetic approach we are using is grammatical evolution (GE). Computer languages can easily be translated into grammars, as they follow relatively strict rules to spoken languages. Grammars provide a strong framework for the evolutionary algorithm to translate genotypes to phenotypes, as well as to design the structure of those genotypes. Researchers improved upon GE and came out with Structured Grammatical Evolution (SGE)[2].The goal of SGE was to decrease the random variance of GE by increasing the structure of the genotypes, so that the recombination and translation of genotypes was more stable. SGE performed better than GE on various algorithms, and in this paper we look at whether SGE can perform better in program synthesis.

2. Research Question

The goal of this thesis is to improve the success rate of program synthesis using genetic algorithms. We aim to use SGE to make improvements upon GE specifically for program synthesis. Because SGE on its own does not work with program synthesis, we make changes and improvements on SGE to solve our problem. The results from SGE are analyzed to discover more ways to improve program synthesis.

3. Background

3.a. Genetic Algorithms

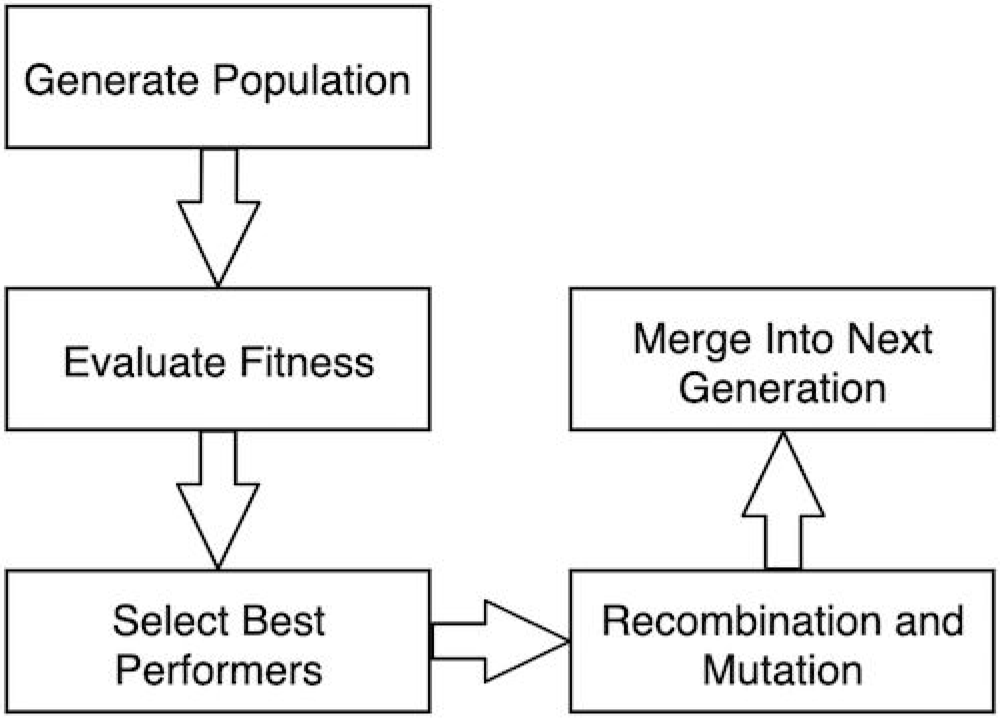


Figure 1. Genetic algorithm flowchart

Genetic algorithms solve problems by generating a set of initial solutions that can be thought of as gene pools and then repeatedly improving upon those solutions in iterations. The problem is required to have a method to measure the performance of the solution. This way, top performing solutions, or genomes, can be combined to create the next set of solutions. Each solution is represented by a list of bits that functions similarly to DNA in that it can be translated to a list of genotypes. The genotypes can then be translated into phenotypes which represent the solution. The genotypes of two solutions can be combined into a child solution by using genetic techniques such as crossover, which takes the parts of two different solutions to make a child solution. In order for different solutions to appear, random parts of solutions can be assigned random numbers, which functions similarly to mutations in DNA. The new solutions form a new population, and the process gets repeated. Figure 1 shows that the cycle increases fitness with each generation. Evolutionary computation can solve a wide variety of problems, as it only needs a way to measure fitness to be able to produce better solutions. However, the solutions are typically less optimal than a solution specifically tailored to each problem. Thus, the evolutionary algorithm's ability to adapt to different problems can be used to an advantage in solving open ended problems where there does not exist an exact solution.

3.b. Grammatical Evolution

Grammatical evolution (GE) uses a context free grammar to construct a list of integers that can be translated into genotypes and phenotypes. The phenotypes represent the solution to the problem.Here, we have a grammar that represents a string (of characters). The values on the left are nonterminals, which can be decomposed into the pieces on the right. On the right, we have different options, called productions, separated by "|". This example is recursive, so a string can either be a letter, or a letter followed by a string, which allows us to form any string from the grammar. We take the modulo of integers in the genotype to decide which production is chosen.

1: $< string >::=< letter > \mid < letter >< string >$
2: $< letter >::=< vowel > \mid < consonant > \mid < space >$
3: $< space >::=' '$
4: $< vowel >::=' a'|'e'|'i'|'o'|'u'$
5: $< consonant >::=' b'|'c'|'d'|'f'|'g'|'h'|'j'|'k'|'l'|...$

Figure 2. Grammar example of english words

For example, in the grammar above, we can have a genotype of 1,0,3,4,1,0,1. We start with <string>, which has 2 productions. 1 = 1 (mod 2), so our string becomes <letter><string>. The genotype gets translated into a phenotype from left to right. The leftmost nonterminal is <letter>, and taking the next number 0, it becomes <vowel><string>. Continuing this pattern, we will get 'i'<string> => 'i'<consonant> => 'i''b'.

The top performing genotypes of a population will be chosen through tournament selection to mate to produce the next population. In tournament selection, we set a tournament size and probability. For each of the two parents of each child, we select the tournament size number of genomes from the population randomly. We then order the genomes in terms of fitness, and then iterate through the list, picking next genome with the tournament probability. Tournament selection aims to select the better performing genomes to create the new child generation, but also on occasion choose low performing genomes. The purpose of this is to prevent the decrease of diversity among the genomes. If we do not implement this, then later populations start looking like just slight variations of the top performing genome, and severely limits the possible solutions that GE and SGE can create. The theory is that by combining the high fitness genotypes, we may find higher fitness offspring. Two parents are selected and produce offspring through recombination and crossover, which is shown in figure 3.

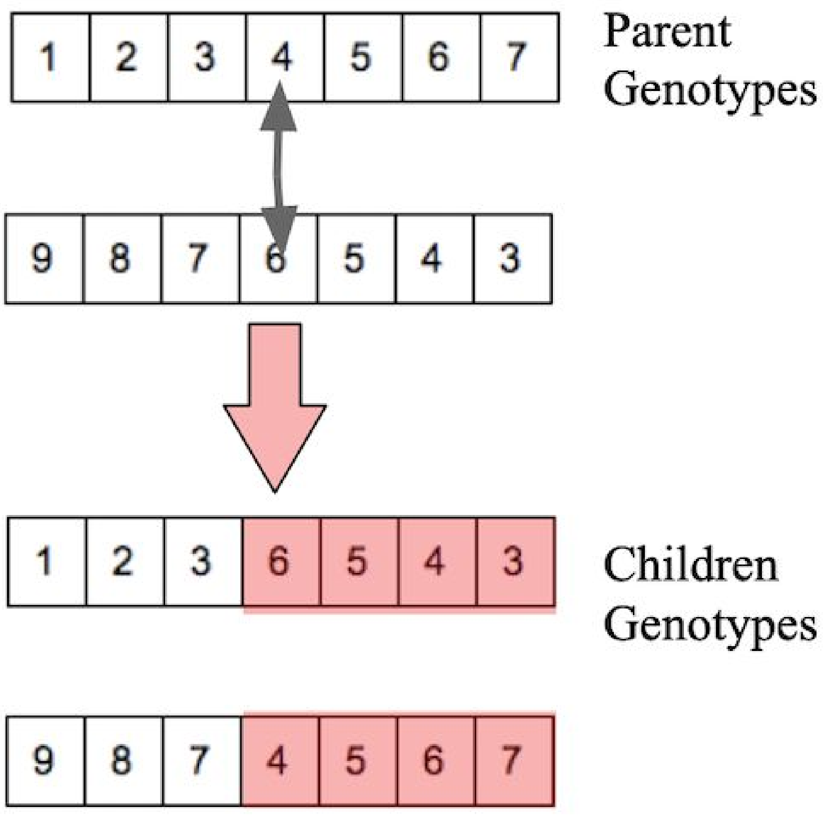


Figure 3. Example of recombination between two genotypes

Genetic algorithms also use mutations as the force to bring new traits and solutions to a problem. Each gene has a small chance of becoming a different random number. Changes that result in higher fitness will be propagated through the population through survival.

The advantage of creating the solution based on a grammar is that the solution has a structure1. As long as the grammar is correct, solutions from GE will always be valid solutions, while other evolutionary algorithms could evolve invalid solutions. Furthermore, the evolution process or even the solution can be modified by people by modifying the language1. Usually, evolutionary solutions are too complex and nonstandard for humans to understand. Modifying the structure of the grammar instead of the actual solution makes it less of a "black box" and allows a human to make modifications to better fit the problem. With these advantages come disadvantages as well. GE algorithms suffer from having a low locality. A low locality means that a change to the list of integers generally results in a larger change in phenotypes. This is because integers may by modified so that different productions are chosen, changing the structure of the solution and nullifying other purposefully selected numbers. In the figures below I demonstrate an example of bad locality using this grammar:

< start > ::= < line > | < line > / < line >

< line > ::= <var> | < var > ∗ < var > | y

< var > ::=x1|x2|x3

Figure 4. Poor locality grammar example

Suppose we have the following two sequences.

	first expansion	second expansion	third expansion	fourth expansion
sequence1	1	2	1	2
production chosen	<line>/<line>	y	<var>	x3
phenotype	<line>/<line>	y/<line>	y/<var>	y/x3

	first expansion	second expansion	third expansion	fourth expansion
sequence2	1	0	1	2
production chosen	<line>/<line>	<var>	x2	y
phenotype	<line>/<line>	<var>/<line>	x2/<var>	x2/y

The sequences only differ by one number. However, the change from 2 to 0 causes the terminal y to become a nonterminal <var>, so the last two numbers "1" and "2", are used for different nonterminals.

In exaggeration, a high locality algorithm will gradually modify the solutions and slowly increase fitness, while a low locality algorithm will make large changes in solutions, relying on luck that a large deviation would result in a better solution. GE algorithms have low locality because changing the length of the representation or of genotypes may cause the entire structure to change.

3.c. Structured Grammatical Evolution

Structured grammatical evolution (SGE) aims to reduce locality by enforcing more constraints on the grammar as well as the crossover of two lists of genotypes to ensure that the modification of one phenotype does not modify the others[2,3]. As we saw in the example grammar above, the nonterminals expand into other nonterminals until they finally become terminals. We count the number of total possible expansions possible for each terminal by using a recursive algorithm. Instead of storing the genome as a list of integers, SGE uses separate lists of integers for each nonterminal. This should theoretically improve locality because changing one list will not change values of another list to be used for a different nonterminal.

A recursive grammar, like the string grammar, is where a nonterminal is both on the left hand side and on the right hand side. For structured evolution, because we count the number of expansions, we must limit the levels of recursion. We can do this by renaming the variables, such as "<var_lvl_1>", "<var_lvl_2>", to remove the recursion. The last level of the recursion must include all combinations of productions that do not contain any recursion, where the recursive nonterminals are replaced with other non-recursive productions. Limiting how deep recursion also further stabilizes the structure, unlike GE where lists may generate long results or short results.

To do recombination, the paper by Lourenco uses a method where for each genotype, one parent is randomly chosen for the child to inherit that genotype. The genotype is a list of numbers, where each number represents the choice of an option from a production. We call this method binary mask[2], as a random binary string is produced to determine from which parent each gene is selected. This is different from GE, which uses a point crossover. Rather than exchanging parts of the entire genotype, only the expansions for a nonterminal are exchanged. For mutation, each genotype has a probability of being mutated by selecting one value from the genotype list and changing it with a new random number.

3.d. SGE for Program Synthesis

In the benchmark for 29 problems, there is a defined structure to the problems. Each problem has a list of training inputs and outputs, used to train the GE. Each problem also has a list of outputs used for testing. The benchmark includes the grammar, as well as that will test GE's code and measure the fitness, which is usually some sort of distance between GE's output and the correct answer. Note that we will not be examining the code created by GE as part of the fitness. The grammars are fully fleshed out to be able to express as many different programs as possible. This makes the search space very large, which further tests how effectively GE and SGE can converge on the correct functions to use. On top of this, the problems range across a variety of different programming techniques. Some of the problems used in the benchmark are described here[4]:

1. Checksum: Given a string, convert each character in the string into its integer

ASCII value, sum them, take the sum modulo 64, add the integer value of the space character, and then convert that integer back into its corresponding character (the checksum character). The program must print Check sum is X, where X is replaced by the correct checksum character.

2. Number IO: Given an integer and a float, print their sum

3. Median: Given 3 integers, print their median.

4. Mirror Image: Given two vectors of integers, return true if one vector is the reverse of the other, and false otherwise.

5. Negative to Zero: Given a vector of integers, return the vector where all negative integers have been replaced by 0.

Small Or Large: Given an integer n, print “small” if $n < 1000$ and “large” if $n \geq 2000$ (and nothing if $1000 \leq n < 2000$).

6. String Length Backwards: Given a vector of strings, print the length of each string in the vector starting with the last and ending with the first.

7. Syllables: Given a string containing symbols, spaces, digits, and lowercase letters, count the number of occurrences of vowels (a, e, i, o, u, y) in the string and print that number as X in The number of syllables is X.

8. Last Index of Zero: Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.

9. Vector Average: Given a vector of floats, return the average of those floats. Results are rounded to 4 decimal places.[4]

One difficulty is that some of the problems do not provide a gradual improvement of fitness. For example, for the problem Small or Large, where the program must distinguish whether one integer is less than, equal to, or greater than another integer, the fitness will make 3 jumps, one jump for each comparison. Thus, the program is looking for an exact solutions, and does not benefit from genetic evolution's advantage of gradually being able to discern which combinations of nonterminals relate to greater fitness.

The goal of applying SGE to program synthesis is twofold. The first goal is to reduce locality to try to improve convergence time. Because code has a clear structure, and a change of any nonterminal will drastically change the code's calculations. Think of trying to change a loop to a while, or a variable to some other variable. GE may perform recombination and mutations that seem much more random than guided by analysis. SGE should be able to preserve much more functionality than GE, as recombination may be seen as moving chunks of code without distorting its functionality. The second goal of SGE is to remove the case when GE does not complete its phenotype expression with the limited amount of genotypes it has. There are two options for GE, either wrap around the genotypes so after using the last one, we start using the genotypes from the beginning again, or to just end the sequence before finishing the expression. Both options appear flawed. By wrapping around, a genotype may be responsible for expression multiple phenotypes, and is highly subject to the variance being more random than structured. With the other option, choosing to stop the code and just inserting blanks for the rest of the program is very likely to result in code that ends abruptly with an error. This just creates useless code that GE will not be using in future generations, and decreases the diversity of each generation. Because SGE has a structure where we can set the maximum amount of recursion, SGE can much more reliably get a solution that without code errors.

4. Methods

SGE had only dealt with recursion when the nonterminal on the left hand side also appeared on the right hand side. However, in program synthesis, there will be much deeper cycles because of loops, ifs, arithmetic, etc. Thus, we devised a solution to prevent cycles from happening. Cycles cause the phenotype to never finish like GE, and will result in code that will run into an error and not finish running. Initially, we first go through the grammar tree to discover all the cycles and keep note of which nonterminals are in the cycle.

We then need to find the productions in each right hand side that can terminate in a cycle. The grammar can only terminate if all the nonterminals become terminals. Thus, we can do depth first search through the grammar tree to find out which paths from the start symbol lead to only nonterminals. Any production that contains a nonterminal that does not have any productions that terminate is considered as part of a cycle. This method is shown in Figure 5.

Algorithm 1 Determine Cyclical and Noncyclical Productions

```
procedure CycleDFS(nt, visitedNT)
    $cyclicalProductions = \{\}$
    $noncyclicalProductions = \{\}$
    $ntTerminates = false$
    for $production$ in $grammar[nt]$ do
        $productionTerminates = true$
        for $option$ in $production$ do
            if $option \in nonTerminalsSet$ then
                if $option \notin visitedNT$ then
                    $visitedNT.add(option)$
                    flag = CycleDFS(option, visitedNT)
                    if $not flag$ then
                        $productionTerminates = false$
                    $visitedNT.remove(option)$
                else
                    $productionTerminates = false$
        if $productionTerminates$ then
            $noncyclicalProductions.add(production)$
        else
            $cyclicalProductions.add(production)$
```

Figure 5. Determining cyclical and noncyclical productions pseudocode

After finding the productions that would not terminate if they were expanded in the last level, we remove them from the grammar. For the very last level of recursion in our tree, we remove all the productions that have nonterminals that do not terminate, and also remove the parent productions connected to such productions. Note that we only do this on the last level. Thus, this algorithm is not perfect, as a perfect algorithm would measure every single production of the last level. However, we believe these two methods of removing the grammar of cycles should remove a significant number of cycles, and thus have lower error rates and gene diversity than GE. The pseudocode is shown in Figure 6.

Algorithm 2 Remove recursion from grammar

```
procedure GRAMMARREMOVERECURSION
    $queue = [startSymbol]$
    while $queueisnotempty$ do
        $nt = queue.pop()$
        if $nt$ in $cycle$ then
            for $i = 1$ to $recursionMax - 1$ do
                $newNT = nt + "lvl" + (i)$
                $grammar[newNT] = copy(grammar[nt])$
                $nonterminals.add(newNT)$
                for $prod$ in $grammar[newNT]$ do
                    for $opt$ in $prod$ do
                        if $opt \in cycle$ then
                            $opt = opt + "lvl" + (i + 1)$
            $newNT = nt + "lvl" + recursionMax$
            $grammar[newNT] = copy(grammar[nt])$
            for $prod$ in $grammar[newNT]$ do
                if $prod$ in $cyclicalProductions$ then
                    $grammar[newNT].remove(prod)$
```

Figure 6. Pseudocode for converting a grammar with cycles into noncyclical grammar

Because any nonterminal may be in a cycle with any other terminal, we also changed the level structure from how it was done in SGE[2]. Instead of having a variable go to the next level if it calls itself, in the program synthesis case, the variables in the production of a nonterminal that are in any cycle must be one level higher than the left hand side non terminal. This prevents cycles from happening. Thus, the recursion level limit represents the actual number of nonterminal expansions in the grammar. Because of this, the recursion limit must be carefully set for each problem, because different grammars may require a different length of expansion to terminate.

5. Experiment

SGE was written in Python 3, and the experiments were run on CSAIL's Openstack servers. The benchmark grammars, fitness and execution codes, and train and test cases were copied from the General Program Synthesis Suite. We only selected 10 out of the 29 problems because of a time constraint. We specifically chose the problems where GE solved the problem at least once to be able to compare efficiency.

In SGE, we use the same parameters as the Benchmark Program Synthesis Suite, with a population size of 1000, tournament size of 7 with probability 0.7[4]. The mutation chance is 0.15, so for every generated child, each of its genotypes has a 0.15 chance to change. The top 3 of each generation will always be carried on to the next generation, and the other 997 genotypes are generated by combining parents of the previous generation. The maximum recursion level was set to 8, so each nonterminal could be expanded at most 8 times. SGE is run for 300 iterations, and recorded for at which iteration it achieved the correct solution. For each problem, we run SGE 100 times and record the number of times SGE's code solved the outputs without any errors.

Source Code: https://github.mit.edu/ALFA-FlexGP/sge_andrew_zhang

6. Results

Problem	SGE Tournamen t	GE Tournamen t	GE Lexicase	Average SGE Generations
Checksum	0	0	0	-
NumberIO	100	68	98	8
Median	2	7	45	173.5
Mirror Image	100	46	78	67
Negative to Zero	0	10	45	-
Small Or Large	1	3	5	282
String Length Backwards	22	7	66	36.4
Syllables	4	1	18	245.5
Last Index of Zero	13	8	21	98
Vector Average	0	14	16	-

Figure 7. SGE on program synthesis results table. The first column is the problem from the benchmark suite that we are solving. The second to fourth columns describe the successful runs out of 100 runs in solving the problem. The fifth column shows the average number of generations it took SGE to solve the problem.

For the results, we compare the performance of SGE with tournament selection and GE with tournament selection. For reference, we include the data on GE with lexicase selection which is a better performing parent selection algorithm. We also include the average number of generations it took SGE to solve each specific problem. SGE solved 76 more cases than GE, and performed on average 16% better than GE. The problems that SGE did solve, it did so in fewer than 100 iterations.

For simple problems such as NumberIO and Mirror Image where GE solved a majority of the 100 trials, SGE was able to solve it every time, performing better than even GE with lexicase selection. Unfortunately, for problems that were solved by GE for less than 5 cases, SGE was also not able to solve. The problems Last Index of Zero and String Lengths Backwards, SGE was able to perform better. In general, SGE was able to solve problems involving for and while loops. But for Median and Small or Large, SGE performed worse than GE. The median problem requires finding the median of three numbers, and the Small or Large problem involves printing if one of the numbers if greater, equal or less than the other number. These problems use multiple layered if commands in their solutions, so SGE still has problems producing solutions that are more complex. SGE was still unable to solve problems where the output answer tend to be either right or wrong. Changes to the code will tend to not increase fitness for these problems, and thus make it difficult SGE to find parts of solutions that it use recombination to produce better code. From the results, for using tournament selection, it seems that SGE only performs slightly better than GE.

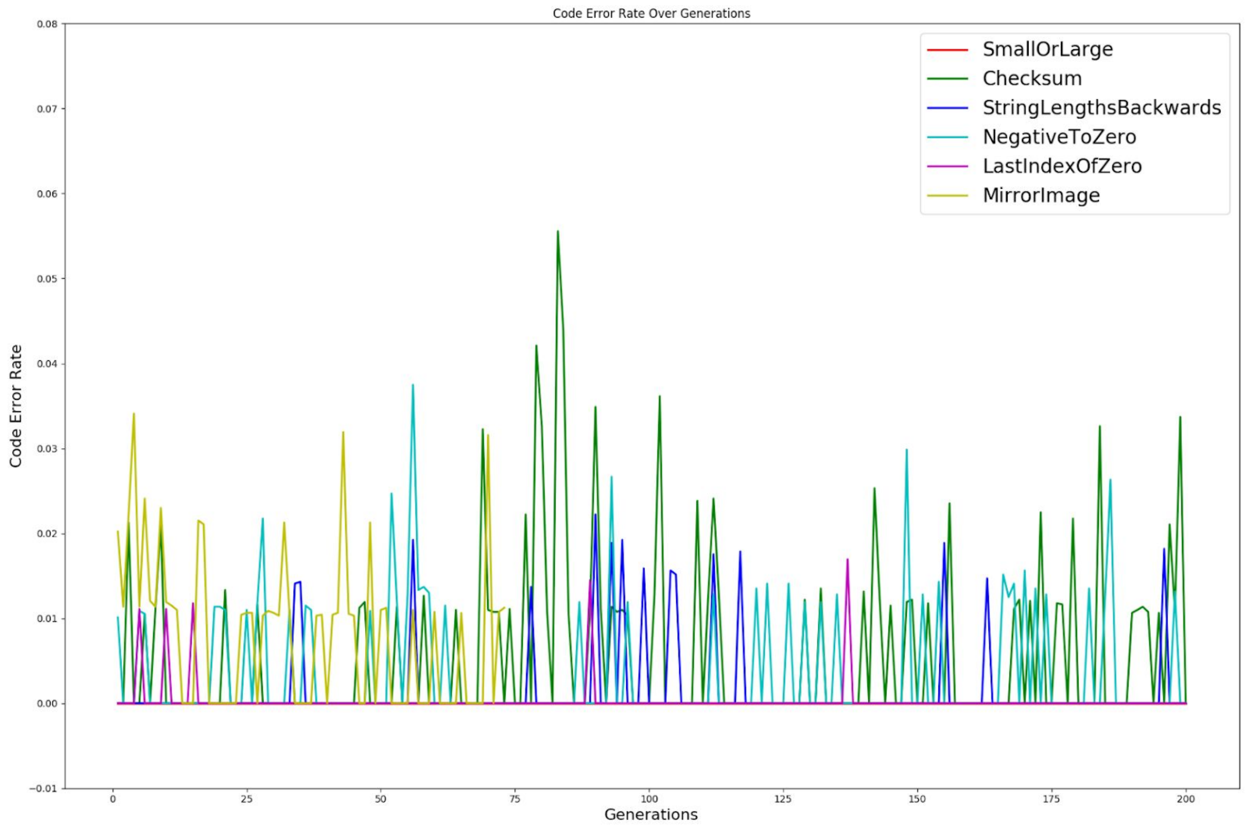
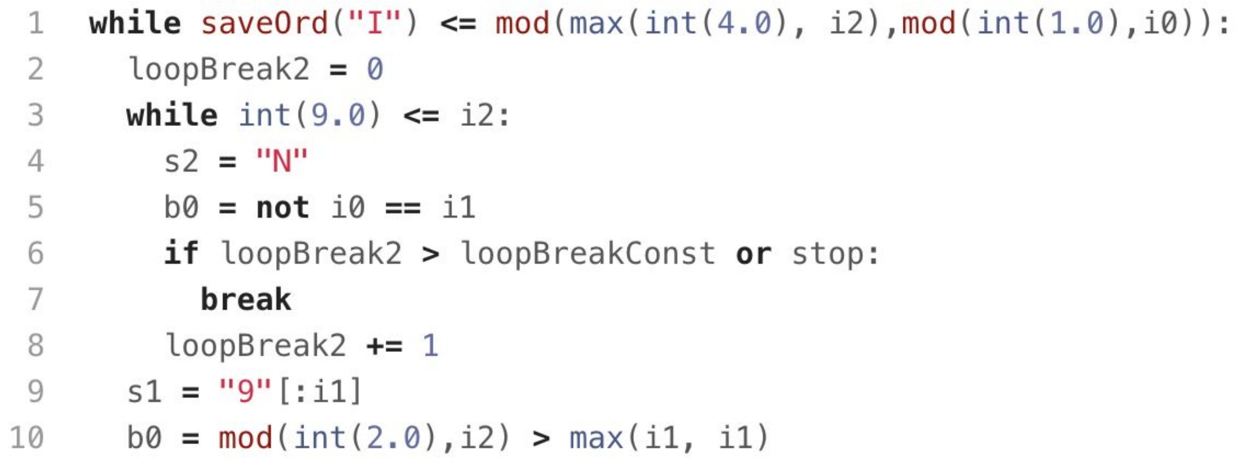


Figure 8: The percent of code for each generation that encounters an error.

We see from Figure 8 that the code produced by SGE is relatively error free. The error rates on average are just 1% of the population. Thus, we are able to remove the need to let the genotype wrap around, while maintaining the completion of running the generated code.

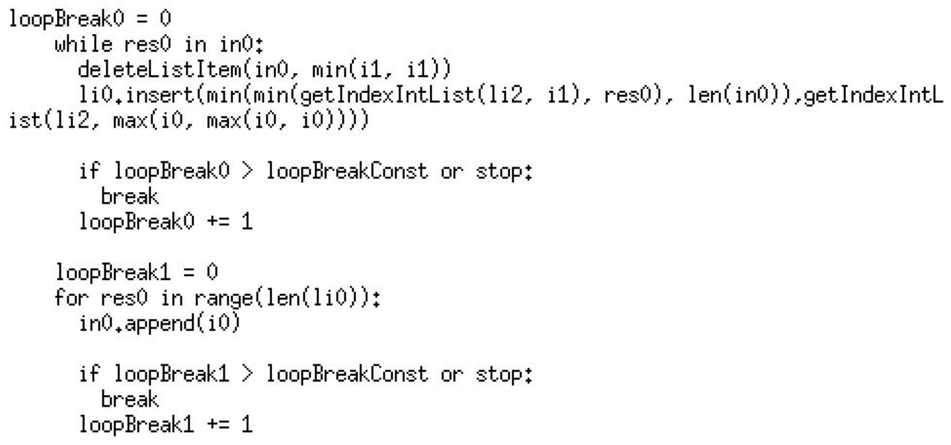
Most of the code that was generated in the process of SGE looked very random and unhuman-like. Usually, the code had a large number of extra random letters and functionality. Thus, a significant amount of code was useless, but was carried on throughout generations. An example is shown in Figure 9.



```
while saveOrd("I") <= mod(max(int(4.0), i2),mod(int(1.0),i0)):
  loopBreak2 = 0
  while int(9.0) <= i2:
    s2 = "N"
    b0 = not i0 == i1
    if loopBreak2 > loopBreakConst or stop:
      break
    loopBreak2 += 1
  s1 = "9"[:i1]
  b0 = mod(int(2.0),i2) > max(i1, i1)
```

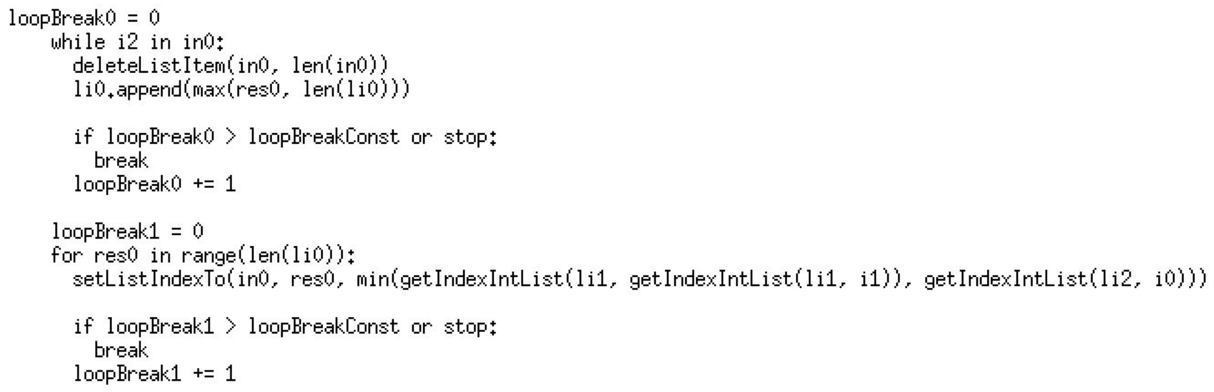
Figure 9. Example of generated code

Two examples of solutions to the problem Last Index of Zero are shown (the code must print the index of the last zero in a list). Note that the code is different from what a human would write.



```
loopBreak0 = 0
    while res0 in in0:
      deleteListItem(in0, min(i1, i1))
      li0.insert(min(min(getIndexIntList(li2, i1), res0), len(in0)),getIndexIntL
ist(li2, max(i0, max(i0, i0))))

      if loopBreak0 > loopBreakConst or stop:
        break
      loopBreak0 += 1

    loopBreak1 = 0
    for res0 in range(len(li0)):
      in0.append(i0)

      if loopBreak1 > loopBreakConst or stop:
        break
      loopBreak1 += 1
```

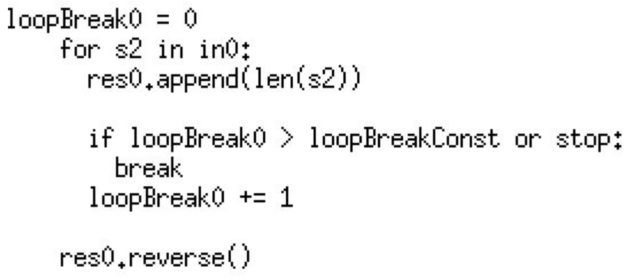
Figure 10. Example 1 of solution to Last Index of Zero problem. The incorrect tab on the first line is the a format byproduct of inserting code. The loop breaks are for breaking infinite loops if SGE incorrectly makes one.



```
loopBreak0 = 0
    while i2 in in0:
      deleteListItem(in0, len(in0))
      li0.append(max(res0, len(li0)))

      if loopBreak0 > loopBreakConst or stop:
        break
      loopBreak0 += 1

    loopBreak1 = 0
    for res0 in range(len(li0)):
      setListIndexTo(in0, res0, min(getIndexIntList(li1, getIndexIntList(li1, i1)), getIndexIntList(li2, i0)))

      if loopBreak1 > loopBreakConst or stop:
        break
      loopBreak1 += 1
```

Figure 11. Example of different solution to Last Index of Zero problem.

The example of solution from String Length Backwards (the code must return a list of the string lengths in reverse) is simple and looks more similar to what a human would write. When many unnecessary nonterminals and terminals were taken out of the grammar, SGE performed with much better success. The fewer possible solutions, the less chance that SGE will generate some nonsense code that does not help with increasing fitness.



```
loopBreak0 = 0
    for s2 in in0:
      res0.append(len(s2))

      if loopBreak0 > loopBreakConst or stop:
        break
      loopBreak0 += 1

    res0.reverse()
```

Figure 12. Example of solution from String Length Backwards.

7. Conclusions and Future Works

From the results, we only see a slight improvement in performance by SGE. However, SGE did manage to effectively remove the use of wrap around in GE while maintaining the correctness of the code. While only marginally better, SGE appears to be a better framework to approach further improvements than GE, because the added structure reduces unnecessary variance in producing solutions. Because GE with lexicase selection performed much better than GE with tournament selection, it would be useful in the future to test SGE with lexicase selection to get a better understanding of if SGE actually does perform better. It would also be interesting to incorporate other forms of selection, such as novelty search.

Looking at the solutions that SGE has created sheds a lot of insight on how to prove program synthesis for grammatical evolution in general. Reducing the terminals and nonterminals that are not necessary for the solution would increase performance. Thus, a method that could understand what parts of the grammar the solution actually needs would increase performance. Novelty search takes a step in the direction of incorporating the solution’s code to the fitness. We think it is possible to take other steps, such as monitor whether lines of code actually have any effect on the final solution. Another way would be to try to limit the number of different nonterminals per line of code. Many lines of code created by SGE were performing functions on many different things, and reducing this, or just splitting the functions over multiple statements may increase performance.

Although the increases in performance from SGE are not compelling, this experiment raises multiple questions whether SGE can be used to make improvements on program synthesis in the future.

Bibliography

1. Painter, Tim. "Grammatical Evolution in Python." . , University of Bath, 2006, www.cs.bath.ac.uk/~mdv/courses/CM30082/projects.bho/2005-6/painter-t-dissertation-2005-6.pdf.

2. Lourenço N., Pereira F.B., Costa E. (2016) SGE: A Structured Representation for Grammatical Evolution. In: Bonnevay S., Legrand P., Monmarché N., Lutton E., Schoenauer M. (eds) Artificial Evolution. EA 2015. Lecture Notes in Computer Science, vol 9554. Springer, Cham.

3. Eric Medvet, Hierarchical grammatical evolution, Proceedings of the Genetic and Evolutionary Computation Conference Companion, July 15-19, 2017, Berlin, Germany.

4. Helmuth, Thomas. Spector, Lee. "General Program Synthesis Benchmark Suite"., 2015, GECCO. http://dx.doi.org/10.1145/2739480.2754769