From Pictures to Programs with Genetic Programming

Brad Zacher

Supervised by Brad Alexander

21/11/2012

Table of Contents

Acknowledgements

I would like to thank my supervisor Dr Brad Alexander for all his help throughout the year; for helping me troubleshoot difficult problems, helping to brainstorm new ideas, for being so patient and understanding during the difficult parts of the year, and for helping to give me a much needed push at the right times.

Thank you to my friends for providing me with a release for my many frustrations throughout the year. For being there with a helping hand, a cold drink or a game of DotA whenever I needed it. Your friendship kept me sane through seemingly insurmountable mountains of coursework which the year buried me under.

Thank you to my father for allowing me to take up so much of your time with helping to proof read my many reports, for letting me put you through the ordeal of listening to me test my presentations, and for letting me bounce ideas off of you. Thank you to my mother for helping to look after me so much throughout the year.

Abstract

Genetic programming (GP) is a powerful and widely used tool for evolving complex programs. In the context of GP, recursive programs have always been a difficult problem to tackle. Their compact and expressive form may seem like an ideal candidate for GP, although their characteristic structure causes standard evolutionary techniques to be unable to reliably produce results. This project presents a method to adapt GP (through the use of grammatical evolution) and inform the algorithm's fitness function using a number of small fragments of the recursive function's call tree.

1 Introduction

Genetic algorithms (GA) are naturally inspired heuristic search algorithms. They seek to use evolutionary-based processes and ideas in order to efficiently generate an optimal solution to a given problem. Genetic programming (GP) is a specialisation of GA, in which the individuals being generated are a representation of a programmatic solution to a problem. The most modern notion of GP was introduced by Koza [1] in 1989, and since then there has been an extensive amount of research into the development and application of the field - at the time of writing, there are over 9,300 papers written on the subject [2]. More specialised still is the technique known as grammatical evolution (GE) [3]. GE creates an intermediate step, mapping a numerical genome to elements of a grammar, allowing for huge flexibility and concisely generated individuals. The use of a grammar means that the GP search can be changed to suite any programming language and style easily, without needing to change the underlying search algorithm.

There are many reasons why GP is widely popular as an automated programming tool, two of which stand out as defining its popularity. First, the underlying algorithm uses a simple representation of the problem which allows the algorithm to be easily adjusted and adapted to the desired complexity and style. Other automated programming methods can use complex representations and operations within their algorithms which have to be finely tuned to a specific problem domain; whereas the GP algorithm can remain constant across different problems, only needing modifications to the representation. In the case of GE, the modifications are as simple as adjusting the grammar.

Second, unlike other automatic programming techniques, GP requires little prior information in order to attempt to find a solution; in some cases a set of "input-output pairs" are enough to evolve a solution. These reasons can also potentially be GP's greatest weaknesses however, as the generality of the algorithm often makes it hard to sufficiently inform and push the search in the right direction.

Recursive problems as a series form a set of rich, complex problems in both mathematical and computer science. Their seemingly simple form allows them to be compact and expressive. The essence of recursion is that it allows for the expression of global solutions through local processing of sub-problems. Which makes them great candidates for a search algorithm as the search space is a priori seems significantly smaller than a non-recursive algorithm, and it means that they all adhere to a basic structure which lends itself to the use of GP.

There is also the fascinating interaction between a recursive program and the genetic algorithm. A recursive solution must consist of two parts; there is base case or stopping condition, this is a (or a

set of) simple logic statements which determine when the recursive calls should stop. And there is the recursive call itself which slowly converges on the base case. Standard GP techniques generally involve treating the evolved program as a "black box" of sorts, and only assigning an individual a fitness value by looking at the output obtained from a given set of input. In general, this can work reasonably well for a non-recursive problem (and even trivial recursive problems); however due to the structure of recursive problems, this sort of testing can only go so far in evaluating the fitness of an individual because it is not very well informed. It fails to consider the fact that a solution can be partially correct (i.e. a base case is correct, but not a recursive case, and vice versa) because in general, a partially correct solution will give you completely incorrect output. For example for the factorial problem, having the correct recursive case of `return n*factorial(n)` but a base case that is even slightly incorrect, say `return 2` instead of `return 1` will cause all output to be out by a factor of 2, which will cause the individual to be scored poorly.

This interaction is what forms the main drive behind this project; the question we seek to address relates to how a fitness function can be sufficiently and efficiently informed about the inner structure of an individual, such that it is able to determine which individuals are "on the right track" – converting the fitness landscape from an "all-or-nothing" cliff, to a gradual and incremental curve.

When approaching a recursive problem, a developer naturally will have a number of input-output pairs that form the basis of the problem. In order to help guide their developmental efforts, intuitively the next step is to begin designing all (or parts of) a call tree to describe the way in which the function will work. A call tree is a powerful and descriptive piece of information which can implicitly describe large amounts of information about the problem. Even a set of small fragments of this tree can provide more than enough information to sufficiently inform the fitness function and create the desired incremental fitness curve. In this report we present a method of applying this set of easy to obtain, problem specific knowledge in order to help inform the fitness function as to the internal structure of the correct recursive solution. We found that by using the information provided by small fragments of the call tree in the search, we were able to greatly increase both the speed and accuracy of the algorithm.

This report begins a brief look at some of the algorithmic ideas the research is based off, as well as some of the recursive problems used for testing, and it then continues with a review of some of the literature relating to GP and GE. The report then presents a discussion of the methods used to solve the problem, followed by an analysis of the experimental results obtained from these methods. Finally, the report concludes with a look at some possible future research opportunities and additional work that may be done based on the results.

2 Background

2.1 Genetic Algorithms

In a GA, as with most search heuristics, a population of individuals is manipulated in order to find an optimal solution to a problem. It uses naturally inspired operators in order to breed candidate individuals together to generate the best solution.

The initial population is generally randomly generated over the possible problem space, with the number of individuals dependent on the problem being solved. In relation to GP, each individual is a programmatic solution to the given problem. Each individual is represented by a genome, which in the case of GE is a variable length string of numbers which, by some process, is converted into the individual's phenotype, which is the individual's true solution to the problem. Each phenotype is individually evaluated by a fitness function, which tests the solution's ability to solve the problem, and gives it a score based on how close its results come to the desired values.

Once all individuals of a generation have been evaluated, then new individuals may be bred from the fittest individuals of the population. A GA does this using any combination of three operators; selection, mutation and crossover. Selection is the simplest operator and involves selecting the best of the best solutions of the current population to keep unchanged for the new population. Mutation involves randomly changing part of a solution. This change can be performed on a fixed or a random portion of the genome; meaning that anything from a small fraction, to the entire genome itself can be changed. Finally, the crossover operator is what differentiates a GA from other search heuristics. Akin to breeding individuals together; it involves taking a number of fit parent genomes from the population, and combing them in some way such that a (or a set of) new child genomes are created containing parts of all of the genomes used in the crossover. This evolutionary process continues until a stopping condition is satisfied.

2.2 Grammatical Evolution

GE uses a standard bitstring GA underneath a genotype to phenotype mapping driven by a language grammar. This mapping creates an important characteristic that allows every genotype to create a syntactically correct phenotype. In normal GP you sort of have to restrict yourself to grammars and semantics which are really robust to random compositions. With GE you can define an ordinary grammar safe in the knowledge that your programs will conform.

For GE, the process by which a genotype is converted to a phenotype is called the mapping process. GE uses a Backus-Naur form grammar (henceforth referred to as a grammar) in order to define the

way in which this mapping process occurs; for an example of a GE grammar, see Figure 3. A grammar is a set of strict rules which can be used to define the exact structure of a program and the limit the possible variations it is allowed to have. From a GP perspective, a grammar is a powerful tool as provides two great utilities. Firstly it creates a simple way to map a genome to a program, which is discussed below. Secondly it acts as a form of constraint operator for the individuals of the algorithm in that only allows each individual to be mapped onto a functionally complete and syntactically correct program, regardless of the individual values of, or length of the genome, preventing the possibility of invalid and hence useless individuals from being evolved.

In order to map a genome to a grammar, each integer x_i of the genome is taken individually. Starting with the first rule of the grammar and the first integer of the genome, the choice an integer represents is the remainder of the division of the integer by the number of expressions of the given rule by taking the remainder of the division, i.e. $x_i' = remainder\left(\frac{x_i}{length(r_j)}\right)$. If the choice contains the name of a rule, then the next integer in the genome is taken and the same process is applied in terms of the new rule. This process repeats until either all of the integers of the genome are exhausted, or there are no more rules to match. This mapping process implicitly creates a tree as in Figure 5(a), where a parent node is created whenever a choice containing a rule is made.

Other GP representations and algorithms often use specialised constructs to map a genome to a functionally complete program. Most methods, for example in Koza's original paper [1], produce solutions in LISP or a similar functional programming language, and others have developed their own specific languages and representations in order to evolve solutions such as the Push language [4]. This means it can often require additional work in order to convert the evolved solution into a form which can be easily used. GE's use of a grammar allows us, or alternately a user of the system to choose exactly what language to produce solutions in. Additionally, by making a simple change to the grammar, we can drastically change form and structure of generated solutions, thus modifying the target and size of the search space without requiring the additional changes to the internal structure of the GP algorithm. Another benefit of using GE is the fact that no extraneous code is generated. By using a grammar we are able to control exactly what is allowed to be in a program, which means that there is very rarely a need to "clean up" unused code from a solution.

2.3 Recursive Solutions

Recursive solutions are an interesting set of solutions most commonly seen in maths and computer science. Simply put recursion implies that a function is defined in terms of itself. They are compact – usually spanning less than 10-20 lines of code, yet they are still expressive – allowing description of

complex and near infinite relationships and number sets. Within this report, there are two basic forms of recursive solutions that we will attempt to solve in this report; standard form and loop form. Whilst both solutions follow the same *basic* structure involving a base case which defines when to halt the recursive calls, and a (or a set of) recursive calls, the difference comes from the way in which they control the recursive calls. Standard form provides a fixed bound on the number of recursive calls made during each function invocation. Loop form uses, as its name suggests, loops in order to dynamically adjust the number of recursive calls that can be made each invocation based on the parameter passed into the function.

The factorial problem is a commonly used mathematical problem, generally seen in the probability and statistical work. It is usually represented by the notation $n!$ and its solution is defined as follows

$$fact(n) = \begin{cases} 1 & n = 1 \\ n \times fact(n-1) & otherwise \end{cases}.$$

The Fibonacci sequence is a famous mathematical relation that is present throughout the natural world. It is slightly more complex than factorial in that it has 2 recursive calls. Its solution is as follows

$$fib(n) = \begin{cases} 1 & n < 2 \\ fib(n-1) + fib(n-2) & otherwise \end{cases}.$$

We chose the factorial and Fibonacci problems as they are both relatively simple problems to solve, so they form a good platform for initial testing and development, and they also create a benchmark for the more complex problems.

An extension of standard Fibonacci is Fibonacci 3, which follows the same basic form, except it has three recursive calls. Its solution is defined as follows

$$fib3(n) = \begin{cases} 1 & n < 3 \\ fib3(n-1) + fib3(n-2) + fib3(n-3) & otherwise \end{cases}.$$

Similarly related to Fibonacci is the Lucas series, which has the same recursive calls, yet different base cases. Its solution is defined as follows

$$lucas(n) = \begin{cases} 1 & n = 2 \\ 2 & n = 1 \\ lucas(n-1) + lucas(n-2) & otherwise \end{cases}.$$

We chose Fibonacci3 and Lucas as they both build off of Fibonacci whilst adding different levels of complexity and without changing the basic structure of the problem.

```
function recurse(x)
  if (<base condition>) {
    return <base value>
  } else {
    return <recursive call>
  }
```

(a) Standard recursive form

```
function recurse(x, c)
  if (<base condition>) {
    return <base value>
  }

  result := <0 or 1, depending on operation below>
  for (i := <range lower> → <range upper>) {
    if(<guard condition>) {
      result := result <operation>
                <recursive call with respect to i>
    }
  }

  return result
```

(b) Loop recursive form

Figure 1 The recursive structures.

All of these problems can be solved using the basic programmatic structure as in Figure 1(a).

More complex than the standard form, loop form solutions have the elements of the standard form (a base case and recursive call); however they also have a loop implemented in order to dynamically adjust the parameters of, and number of the recursive calls each invocation. This allows them to capture more general patterns of computation. Unfortunately, the structure of these problems is not as clearly nor easily defined as the standard form. In order to solve these problems, we designed a prototype function shown in Figure 1(b). This prototype is discussed in depth in the next section. Whilst we do not explicitly use this prototype on more problems in this report, we took care to create it with a number of other problems in mind, and have used this form to solve other problems by hand (i.e. without the use of the GA).

Refactoring [5] is a more advanced problem which involves counting the number of ways in which a number can be written as a product of factors. Unlike the TopCoder problem however, we count $x \times 1$ as a factor. For example, $refactoring(24) = 7$ because 24 can be represented by $2 \times 2 \times 2 \times 3$, $2 \times 2 \times 6$, $2 \times 3 \times 4$, 2×12, 3×8, 4×6 and 1×24.

3 Related Work

In Koza's original paper [1], he uses basic GP with a constrained function set in order to generate solutions to the Fibonacci problem. He used lisp's s-expressions in order to symbolically represent the recursion. However he used the first *20* elements of the sequence as input-output for the problem in order to evolve the solution, which is a large amount of input to require a user to input into the system. In contrast our system is able to reliably generate a solution using the first 4 elements of the series. Given this much input however, he was able to successfully evolve the solution to the problem, which in itself shows an interesting result about how much information is required to make a standard GP system.

Koza [6] used GP to evolve automatically defined functions (ADFs); dynamically generated functions of indeterminate size and complexity that are wholly produced as the result of the evolution. By allowing the ADFs to reference themselves, they become able to solve recursive problems. ADFs allow GP to more closely imitate human programming, allowing them to decompose problems into smaller, separate problems. However this method is better suited to, and hence often applied to more a different set of recursive problems such as the even-n-parity problem (i.e. not the problems we are seeking to solve in this paper), due to the fact that these types of problems can be built off of solutions to smaller versions of the same problem.

One example of a group of solution to evolving recursive programs using GP is the work done by Yu, which evolves solutions to recursive problems through the use of lambda abstractions and higher-order functions, such as map, foldl and foldr, which provide a mechanism to perform recursive operations on lists of numbers implicitly, which helps to solve the problems of infinite recursion. Whilst Yu's earlier work [7][8] is focused on solving recursive problems that are applied to lists (in particular the even-*n*-parity problem), in [9] Yu defines a new implicit recursion function foldn and applies it to the Fibonacci problem. Yu is able to evolve the correct solution with a 97% success rate. However this approach does not strictly define recursive function as it produces solutions that calculate using a bottom-up approach and are thus allowed to *directly* reference previously calculated values and are run iteratively. Additionally as this approach was not tested on any other relationships, it is unclear how well the approach would work on more complex relationships, such as additional recursive calls and/or base cases.

The Push programming language is an interpreted language that has been developed specifically for use with GAs. A specialisation of this system used for GP is PushGP [4]. Push uses a number of stacks in order to store and manipulate data values as well as program instructions. It is highly dynamic, allowing programs to directly manipulate and create new instructions to be pushed onto the

execution stack. In terms of GP, this allows programs to be highly dynamic and flexible. It can even allow for autoconstructive evolution, in which programs directly generate their own programs by some means, rather than the using GA's standard crossover and mutation operators. In relation to generating recursive problems, the system is able to generate correct, generalizable solutions to a number of recursive problems, such as factorial and Fibonacci. Whilst they do achieve some success with these problems; being able to produce correct solutions to factorial using the first 8 numbers of the sequence, and similarly for Fibonacci using the first 12 numbers, and whilst they summarise the environments in which all of their experiments are performed, the authors do not explicitly define the environment and results of individual tests, thus it is difficult to properly analyse their results. Additionally, due the stack-based nature, execution style and syntax (or lack thereof) of the language means that the generated solutions are sometimes hard for humans to properly interpret and require a lot of work in order to convert into other languages and applications. For example one solution to the factorial problem is `(1 EXEC.DO*RANGE INTEGER.*)`, and one solution to Fibonacci is `(EXEC.DO*TIMES (CODE.LENGTH EXEC.S) INTEGER.STACKDEPTH CODE.YANKDUP)`. Whilst both these solutions represent correct and completely generalisable solutions to their respective problems, they do not match any conventional solutions to the problem, which means that for an average user of an application who is not likely to have sufficient knowledge of the Push3 language, nor the ability to trace the processing steps of the solution, these solutions do not clearly solve the problem and additionally would be largely difficult to convert to standard programming languages, which greatly reduces the usefulness of generated individuals.

Cartesian GP (CGP) [10] is Miller and Thomson's approach to GP. It represents candidate individuals as a series of interconnected nodes in a graph, with each node representing either a specific function call or an input for the system to use. It is similar to GE in that it represents an individual's genome as an integer string, which is later iterated through in order to generate the phenotype candidate program. Self-Modifying CGP (SMCGP) [11] is an extension of standard CGP that adds a number of "modification" operators to the allowed function set that enable individuals to directly modify their phenotype during execution. These self-modifications can, given the correct conditions, create a form of implicit recursion, allowing an individual to emulate the recursive process via duplication of sections of the function graph and similar. They test their system on the Fibonacci sequence with two tests using the first 12 and 50 numbers of the sequence as respective inputs. They manage to achieve good success rates in relation to the program inputs as well as good generalisation rates. It is worth noting that they found that the high number of input values produce a better chance of a successful individual being able to generalise, which is to be expected.

4 Method

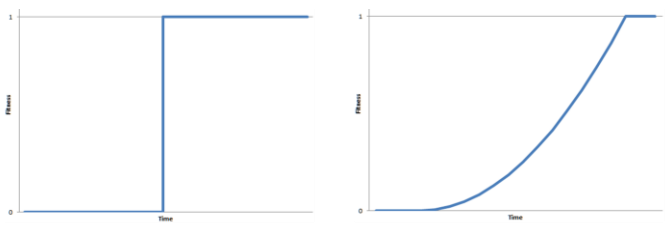
The problem with basic input-output pairs testing only, as previously mentioned, is that partially correct solutions are often going to fail these tests, and thus are given a poor fitness score, which turns the fitness landscape into an "all-or-nothing" search as in Figure 2(a). However the basic drive behind a genetic search requires that the population with the best fitness are selected for breeding and mutation so that better solutions can be produced from them. For this to work most efficiently, the fitness landscape must look more like Figure 2(b), which means that partially correct solutions must get allocated some fitness, or else the algorithm by definition will not select them.

The main research question of this project can be summarised in a single sentence: What does it take to sufficiently inform the fitness evaluations of a GP algorithm as to the completeness of a solution to a recursive problem? In solving this question, one must strike a careful balance between too much and not enough information. If we require that the user input too much information, too much problem specific knowledge into the algorithm, then the algorithm collapses into more or less a useless piece of software, as the user has more than enough information to simply solve it themselves. Not enough information and the algorithm devolves into a blind search of the space, which can often result in incorrect solutions and individuals that can't generalise well, if at all.

4.1 "…to Programs" – Adapting GE to recursion

4.1.1 Basic input-output pairs

In the beginning we started with the smallest amount of information that we could – simple sets of input-output pairs. Initially we started with a basic GA implemented using two the open-source C++ libraries; GALib [12] for the underlying GA algorithm, and libGE [13] in order to allow us to use the GE representation. Although, as previously discussed, we had theorised that this would be insufficient information in order to efficiently generate solutions, it was a good first step to build up familiarity with the inner working of the libraries, and it allowed us to generate a set of "control" results to try and improve upon. As expected, the results were less than astounding. The grammar that was used is shown in Figure 3. It's worth noting that this grammar has been restricted in that it only allows for the integers between 1 and 3, as it was found that the search space was too large



(a) Difficult fitness landscape (b) Good fitness landscape

Figure 2 Two fitness landscapes.

```
<individual>::=\
if (<condition>) {\n\
    return <base>;\n\
} else {\n\
    return <expr>;\n\
}\n

<condition>  ::= <var> <logic_op> <digit>

<logic_op>   ::= "<" | ">"

<op>         ::= - | * | +

<digit>      ::= 1 | 2 | 3

<var>        ::= x

<base>       ::= <digit> | <var>

<expr>       ::= recurse(<rec_expr>)
               | recurse(<rec_expr>) <op> <maths_expr>
               | <maths_expr> <op> recurse(<rec_expr>)

<maths_expr> ::= <digit>
               | <var>
               | recurse(<rec_expr>)

<rec_expr>   ::= <var> <op> <digit>
               | <digit> <op> <var>
```

Figure 3 The initial grammar used for basic input-output testing.

with more integers, which prevented the algorithm from finding a solution to factorial. Similarly, in order to reduce the complexity of programming, the division operator was excluded as it adds extra complexity in terms of not only dealing with floating point numbers, but also in terms of dealing with division by zero. These results perfectly illustrate the point that a basic, naïve approach cannot reliably solve even marginally complex recursive problems. The algorithm was unable to consistently find a solution for the simplest of recursive problems, factorial, and was not able to consistently find the correct solution to the marginally more complex Fibonacci problem.

4.1.2 Tree Fragments

From this basic implementation and results, we looked at the problem domain – recursive problems – and looked at what information a user could easily obtain, namely the information they would have handy if thinking about a recursive problem. We settled on an old design tool that is sometimes taught when a student is first learning recursive programming – a call tree. A call tree is an incredibly descriptive and concise piece of information that allows a user to logically describe the way in which a recursive program internally structures its recursive calls (an example tree for Fibonacci(4) is shown in Figure 4). However, we felt it would be unreasonable and unnecessary to require a user to supply an *entire* call tree down to a base case; even from a relatively low initial call value, such as Fibonacci(4) down to Fibonacci(1 or 0), as with this much information, the user would be able to

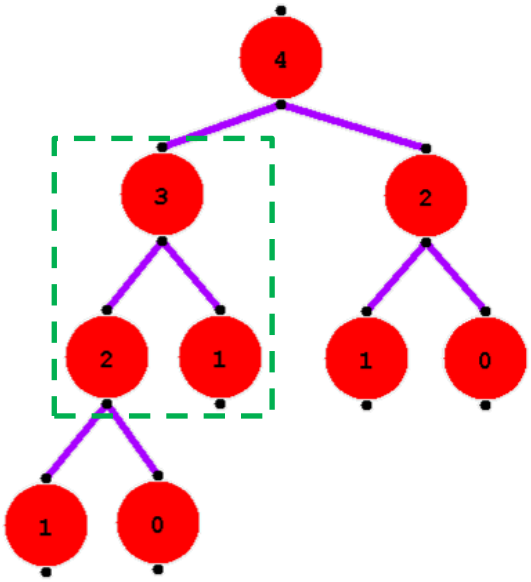
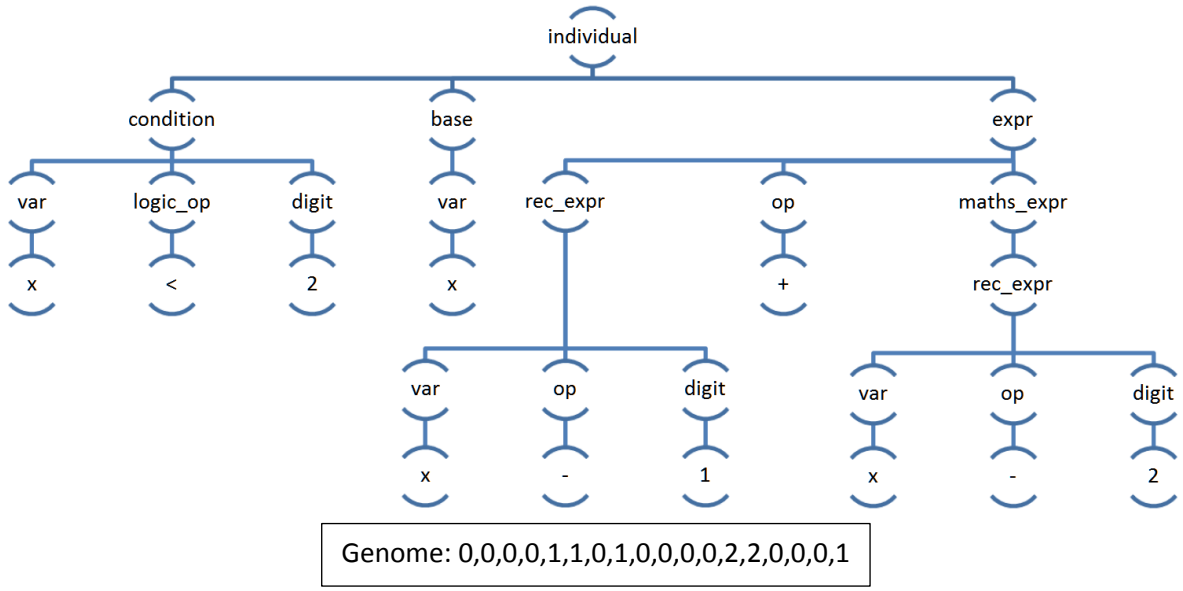


Figure 4 The call tree for Fibonacci(4), with a fragment of the tree highlighted within the dotted square.

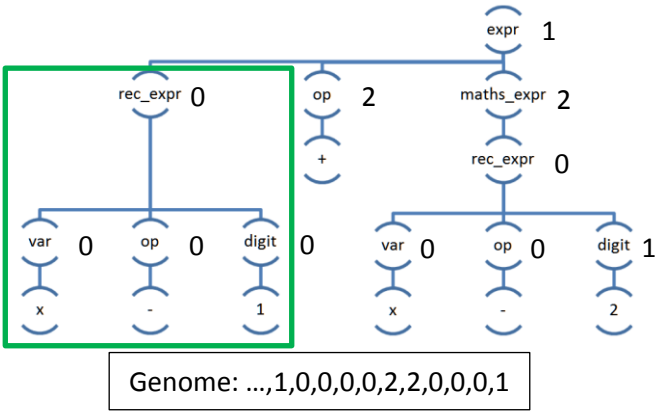
easily, manually solve at most of the solution (i.e. the user needs to have *too much* information to use the algorithm). Thus we formulated the notion of *fragments* of a call tree. A *call tree fragment* is simply a parent-child relationship from somewhere within the complete call tree of the solution, such as the one shown within the green square in Figure 4. This is fragment depicts Fibonacci(3) as the parent, and Fibonacci(2) and Fibonacci(1) as the children, meaning that Fibonacci(3) recursively calls Fibonacci(2) and Fibonacci(1). With a number of these fragments, it becomes relatively easy to inform the fitness function about what each recursive call should look like.

Given the decision to use fragments, the next question was how to use it to inform the search. The first approach used was to integrate it as part of the input-output fitness function, meaning the recursive calls were individually tokenised out of the phenotype and tested against the fragments from the call tree. This seemed like a good choice as it required the least amount of modification to the underlying algorithm whilst still allowing for full integration of the new fitness tests. However whilst the algorithm was able to more consistently evolve solutions to Fibonacci and factorial, it was not able to do it in a shorter number of generations. This was because our implementation so far had revealed a weakness that comes from the underlying GE representation, called the ripple crossover [14].

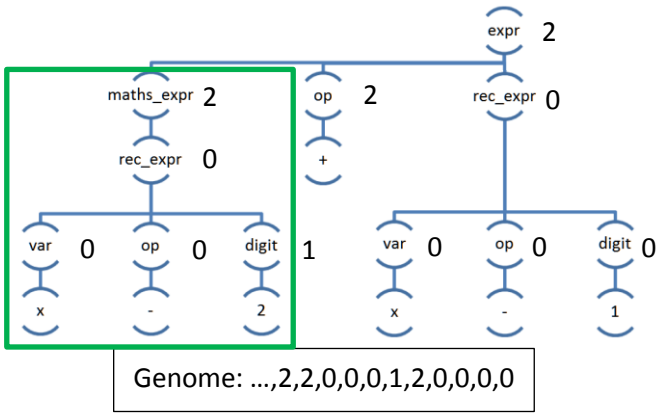
GE, like all other GP representations, treats the genome as a tree or a graph. In most representations this means that a crossover of two partially correct solutions (i.e. to create a completely correct solution) is as simple as replacing the incorrect node and its children from one solution with the correct node and children from another solution. However due to the way in which the integer genome is mapped onto the grammar, the symbol that the value of the genome represents at a



(a) The correct expansion of the grammar from Figure 2 to solve the Fibonacci problem.



(b) An expr branch



(c) Another `expr` branch

Figure 5 The weakness of grammatical evolution's tree mapping.

given point is entirely dependent on what the previous values of the genome have been mapped to, which in turn depends on what the rules of the grammar allow. To illustrate this, an example has been shown in Figure 5, which uses the grammar in Figure 2 to solve the Fibonacci problem.

Figure 5(a) shows the complete, correct solution. For the `expr` branch, there are three possible choices, the latter two of which will give the correct result as they are the reverse of one another; both are shown in Figure 5(b) and Figure 5(c). In this example, if you wish to replace the first branch in (b) with the first branch in (c) (the branches within the green squares), you would not be able to do this for the reason above; as the `expr` node from (b) has a value of 1, it forces the following genome integers to be mapped to the `<rec_expr> <op> <maths_expr>` rule, which also means that as there is an extra node in (c)'s subtree, this will also affect the value of the `op` and `maths_expr` nodes. Unfortunately the only way to reduce the impact of this is by developing grammars with smaller tree depths.

4.1.3 Splitting the Search

From this obstacle, a new method of integrating the fragments was needed, as well as a more robust and lower depth grammar. This lead us to the final approach; splitting the search into two phases. Unfortunately this approach required significant code refactoring, however it allowed us to easily remove much of the additional depth from the grammar.

Phase 1 of the search involves evolving the individual recursive calls of the solution and nothing else. The grammar for this phase can be seen in Figure 6. Note that the digits are separated into 4 separate rules. This is done based on the observation that the recursive problems we were solving did not contain the larger single-digit numbers, and whilst we wanted to not only make it harder for the algorithm to evolve solutions, we also wanted the grammar to be ready for problems that involved the higher digits. This form gives the grammar a bias toward the smaller digits (i.e. there is a $\frac{2}{3}$ chance of producing a 1 or a 2, but there is only a $\frac{1}{36}$ chance of evolving an 8 or 9).

The fitness of a candidate individual in this phase is based exclusively off of its ability to match the given tree fragments. The fitness of an individual is given by the following equation

$$fit = \frac{m}{\sum_{i=0}^{m}\left(\min_{(j=1,\dots,n_i)}\left(\mathrm{dist}\left(f(p_i), c_{ij}\right)\right)\right) + m}$$

where $f(x)$ is the candidate individual, p_i represents the ith parent value with n_i children, c_{ij} represents the jth child of the ith parent and m represents the total number of parents. Note that because there are no constraints on the ordering of the children of a fragment, a candidate

```
<expr_root>     ::= <var> <op> <digit>
                  | <digit> <op> <var>

<op>            ::= - | * | +

<var>           ::= x

<digit>         ::= 1 | 2 | <big_digit>
<big_digit>     ::= 3 | 4 | 5 | <bigger_digit>
<bigger_digit>  ::= 6 | 7 | <huge_digit>
<huge_digit>    ::= 8 | 9
```

Figure 6 The grammar for Phase 1

individual must be tested against every child. The choice of distance functions is discussed in detail within the next section. Intuitively, if the individual matches one child, then the result of the min term will be 0. Trivially, if the individual matches one child of each parent, then the sum term becomes zero, causing the fitness to be 1.

Given the small size of this phase's grammar, it meant that the search space was very small, which allows the search to find a correct individual within a small number of generations. However, due to nature of a GA as well as the implementation that was used, it was not feasible to attempt to evolve all recursive calls at once; as it is not possible to guarantee that the algorithm will evolve all cases within a single population. In lieu of this, this phase is repeated once for each recursive call within the algorithm, with the found solutions being given reduced fitness to lessen their influence on the evolutionary process. This approach worked exceedingly well for all of the problems. We found that in some cases where there was an exceedingly simple relationship to derive from the fragments, the algorithm was able to generate one or all of the correct recursive calls by "dumb luck" in the initial population. Whilst this sped up our runtimes because it caused phase 1 to complete quickly, we felt that it could cause problems down the road when there may be better solutions, thus we force the algorithm to run for at least 5 generations before it is allowed to return with a perfect individual.

As a side note, this repetition means that there are a considerable number of extra individuals to process, which impacts heavily on the runtime of the algorithm. Even though a large number of optimisations were made such as the rewriting and simplification of data structures and removal of extraneous operations, the runtimes achieved were still subpar. The cause of this problem was later discovered to be the choice of compiler. As the grammar is written in standard C, an individual must be compiled before it can have its fitness evaluated. Originally due to some object-orientated data structures, the standard GNU C++ compiler (g++) was used, and later after these were simplified, the standard GNU C compiler (gcc) was used. Due to the large amount of functionality available to these compilers – much more than is required to compile the simple individuals, they are incredibly slow when they are required to compile a large number of unrelated programs in series. For g++ it took between 140ms and 160ms per individual, and for gcc it took between 40ms and 60ms per

individual. Compared to the 1-5ms evaluation time, this was a disproportionately large amount of time. This problem was rectified by switching to the tiny C compiler (tcc). This reduced the compile times to a significantly more manageable 8-10ms.

Phase 2 follows the same ideas as earlier in that it evolves the entire function using input-output testing. Contrary to earlier methods, from phase 1 we have already figured out the recursive call values, meaning a lot of the work has already been done, which removes some of the depth from the grammar. This allowed us to extend the grammar to a slightly larger set of programs. Note that from phase 1 we know how many recursive calls there are, and the grammar is simply modified to reflect this. Figure 7 shows the grammar for a function with two recursive calls. The fitness of an individual is calculated as follows

$$fit = \frac{m}{\sum_{i=0}^{m} \left(\text{dist}\left(f(x_i), y_{ij} \right) \right) + m}$$

where x_i is the ith input value, y_i is the related ith output value, m is the total number of input-output pairs, and $f(x)$ is the candidate individual. The fitness equation is chosen to be of this form for the same reasons above, however in this phase there is only one possible output for each input, ergo there is no need for a minimisation term.

During testing we naturally had some problems with infinite recursion. Whilst we did our best in order to design our grammars such that this was able to happen as little as possible, there was always the chance that infinite individuals would be generated and cause our algorithm to slow whilst it waited for the evaluation to naturally error out and terminate. In order to prevent this, we followed the example of [15]; we looked at the evaluation times that were being achieved for non-infinite solutions – around 1-5ms – and we imposed a physical runtime limit of 20ms on the evaluation. We felt that 20ms would be adequately long to allow seeming infinite solutions to converge to the base case. Any individuals that have not terminated by this time are considered incorrect and are thus considered to be incorrect and are assigned minimum fitness.

4.1.4 Handling loop structures

From our success with basic structured recursive problems, we turned our attention to more complex problems with a loop based structure. Initially, these problems proved to be a problem because of their unusual and inconsistent structure. It was difficult for us to determine how to best approach these problems from a GE point of view. We originally looked at using a grammar which was a hybrid, containing the elements of both standard and loop structures. However this was unnecessarily complicated and verbose, and often it caused problems with the evolutionary process

```
<expr_root>     ::= \
if (<var> "<" <digit>) {\n\
 return <lit>;\n\
} else if (<var> "<" <digit>) {\n\
 return <lit>;\n\
} else {\n\
 return <expr1> <op> <expr2>;\n\
}

<expr1>         ::= <rec1>
                  | <rec1> <op> <lit>
                  | <lit> <op> <rec1>
<rec1>          ::= recurse(param(<var>, 1))

<expr2>         ::= <rec2>
                  | <rec2> <op> <lit>
                  | <lit> <op> <rec2>
<rec2>          ::= recurse(param(<var>, 2))

<op>            ::= - | * | +

<lit>           ::= <digit> | <var>
<var>           ::= x

<digit>         ::= 1 | 2 | <big_digit>
<big_digit>     ::= 3 | 4 | 5 | <bigger_digit>
<bigger_digit>  ::= 6 | 7 | <huge_digit>
<huge_digit>    ::= 8 | 9
```

Figure 7 The grammar for Phase 2 for a problem with 2 recursive calls.

because the phase 1 evolution would generate a standard form recursive call which would ultimately then throw off the result of the run. Eventually we decided to keep the two structures separate and we arrived on the structure previously discussed and shown in Figure 1(b). This structure allowed us to put numerous constraints on the grammar, whilst still leaving the solution space open enough to be able to solve a more general series of problems.

Due to the fact that the recursive calls are made in terms of both x and an iteration variable i, a new phase 1 grammar needed to be created, seen in Figure 8(a). Early experiments involved a method similar to the process from the standard phase 1, in that we evolved just the recursive call, however this method proved to be ineffective and often caused the algorithm to pick `<var> - i` as this can more or less readily match any tree structure with a loop. From the new structure we noticed that similarly related to the tree structure is the `guard` rule, which determines when a recursive call is made in the loop. By evolving both terms at the same time, we achieved a much better result. Note that in Figure 8(a), the "`/**/`" terms are included to delimit the start and end of each term so they can be easily split out of the same phenotype.

Given this new structure of the recursive problems, a new phase 2 grammar was defined to accommodate not only the vastly different structure, but also the myriad of extra evolutionary variables, shown in Figure 8(b). This grammar – as with all grammars in the report – is written in standard C and hence makes use of pre-compiler definitions in order to define the op and related

```
<expr_root> ::= <loop_expr>
<loop_expr> ::= /**/(<var> "<" i)/**//**/(<var> - i)/**/
              | /**/<guard>/**//**/(i <loop_op> <var>)/**/
              | /**/<guard>/**//**/(<var> <loop_op> i)/**/
<loop_op>   ::= * | + | /
<guard>     ::= (<var> % i == 0) | (<var> "<" i)
<var>       ::= x
```

(a) Phase 1

```
<expr_root>    ::= \
#define op <op>\n\
int i, result;\n\
if (<pred>) {\n\
  return <ret>;\n\
}\n\
result = base_op_val;\n\
for (i = <rl>; i "<" <ru>; i++) {\n\
  if (<guard>) {\n\
    result = result op recurse(<loop_expr>, i);\n\
  }\n\
}\n\
return result;\n

<op>           ::= + \n#define base_op_val 0
                 | - \n#define base_op_val 0
                 | * \n#define base_op_val 1

<pred>         ::= <var> "<" <digit>
                 | <var> ">" <digit>
<ret>          ::= <digit> | <var>

<rl>           ::= c | <digit> | <var>
<ru>           ::= (<var> + <digit>)
                 | (<var> - <digit>)
                 | (<digit> - <var>)
                 | <var>

<var>          ::= x

<digit>        ::= 1 | 2 | <big_digit>
<big_digit>    ::= 3 | 4 | 5 | <bigger_digit>
<bigger_digit> ::= 6 | 7 | <huge_digit>
<huge_digit>   ::= 8 | 9
```

(b) Phase 2

Figure 8 The grammars for loop structures

neutral value (called the `base_op_val`). Note that the rules `trans_i` and `guard` are excluded from the phase 2 definition as they are directly replaced with the expressions evolved in phase 1. For example, the refactoring problem can be solved as in Figure 9.

4.2 "From Pictures…" – Creating an intuitive way to control the algorithm

With our new, specialised adaptation of GE, we felt we needed a more intuitive way of controlling the algorithm, and changing the various data structures and parameters. Whilst it is adequate to manually adjust the two grammars as well as the data structures holding the tree fragment data based on the problem, it is a tedious process and can sometimes be affected by human error.

We developed a simple, yet elegant GUI to allow for a user to input a tree, which has two main purposes. Firstly it facilitates the inputting of the call tree fragments as well as the input-output pairs, and secondly it provides the framework for analysing the tree fragments and automatically selecting and generating the required grammars, as well as coordinating the separate execution of the phases of the algorithm.

The input system has been designed to be as simple and intuitive as possible. Tree fragments can be input by creating nodes and linking them to their related parents and/or children. The fragments can be completely unlinked and separated as in Figure (a), they can be logically linked together based on how the call tree actually looks as in Figure 10(b), or they can be linked with no node repetition as in Figure 10(a). The node labels represent the input parameter at that node and the value within brackets represents the output value for that specific node.

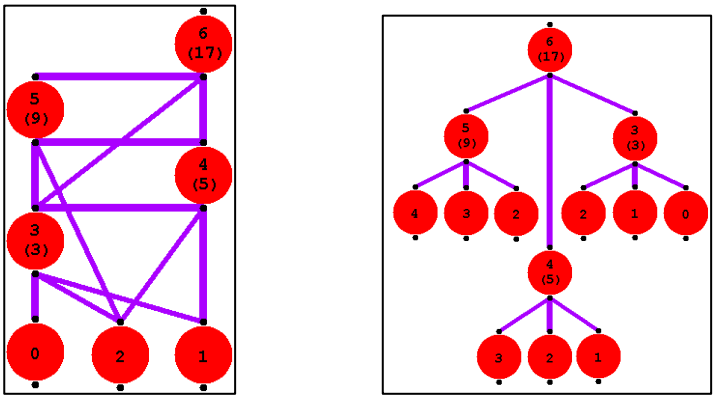
Once all of the desired tree fragments have been input, then the system can be instructed to build the files required by the GA; the data structure containing the inputted tree data, the phase 1 grammar and the phase 2 grammar. During this process, the algorithm will perform a simple analysis of the number of children per parent order to determine the number of base cases that are possibly required, the number recursive cases needed and similarly the number of phase 1 searches required. It also analyses the consistency of the number of children to determine if the problem may need loop structures in order to solve. It also adjusts the parameters of the GA, such as the maximum number of generations to run for as well as the population size.

We had originally aimed to develop a system to allow for hand-drawn call trees to be used as input into the system. However after some research, we realised that the various aspects of image recognition required in order to be able to reliably detect not only the individual nodes, but also the links between the nodes and, more importantly, the hand-written integers representing the node

```
int recurse(int x, int c) {
  #define op +
  #define base_op_val 0

  int i, result;

  if (x < 2) {
    return x;
  }

  result = base_op_val;
  for (i = c; i < (x+1); i++) {
    if (x % i == 0) {
    result = result op recurse(x / i, i);
    }
  }

  return result;
}
```

Figure 9 An example solution to refactoring using the grammars from Figure 8

values constitutes a separate research project in itself. It would be interesting as a possible future research path to increase the ease of which the system can be used.



(a) Highly interconnected (b) Logically connected

Figure 10 Fibonacci3 tree fragments.

5 Results

We ran each of factorial, Fibonacci, Fibonacci3, Lucas and refactoring problems through our algorithm 50 times, each with 4 input-output pairs and 3 or 4 fragments, the exact inputs are shown in 10(b) and 11. We allowed the algorithm itself to control the population size and maximum number of generations based on the input fragments. The results of these experiments can be seen in Table 1 and are shown graphed in Figures 12, 13, 14 and 15. For each problem we ran experiments with three different distance metrics, to test the effects and performance of different forms of penalisation on the various evolutionary results and also to help inform us of the best metric to use by default. In order to compare our results with standard GP methodologies; for factorial, Fibonacci and Fibonacci3 we ran additional tests without the use of tree-fitness (i.e. just using input-output pairs). These results are shown in Table 2 and are similarly graphed in the above figures. Note that after producing the results for Fibonacci3 and noting not only the terrible runtime, but also the fact that it was unable to evolve a useful solution to the problem, we felt that there was not any need to attempt to evolve solutions for the remaining problems.

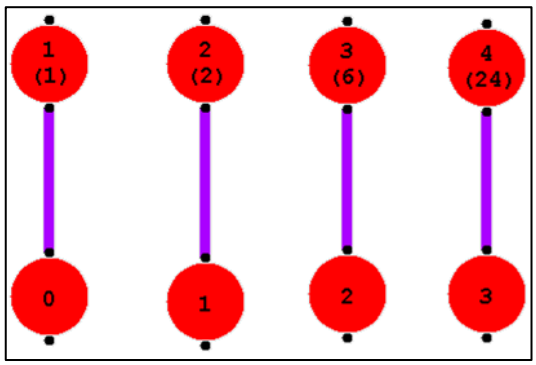
5.1 Distance metric results

The all-or-nothing metric assigns calculates distance as per its namesake; it returns a distance of 0 if and only if the two numbers are equal, otherwise it gives a distance of 1. The formula is as follows

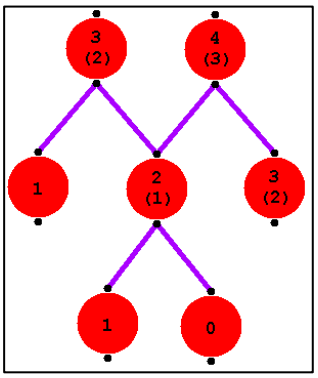
$$\mathrm{dist}(a,b) = \begin{cases} 0 & a = b \\ 1 & otherwise \end{cases}.$$

The metric performed well for the Lucas problem; not only was it able to solve the problem with the lowest runtime and number of generations, it was also the most accurate. However it struggled to perform well for most of the other problems, consistently achieving the slowest and least accurate results. This is what we expected from its design, as it does not help inform the evolutionary process of how close to correct a solution is. These results suggest that this metric appears to work best when the problem's complexity comes from its base case; such as the Lucas series' reversed base cases. This is because less emphasis on the evolution of the recursive case, which means that a more informative metric, such as the ones below, will report that incorrect individuals are closer than they should, which means they are more likely to cause the algorithm to move toward a local optima.

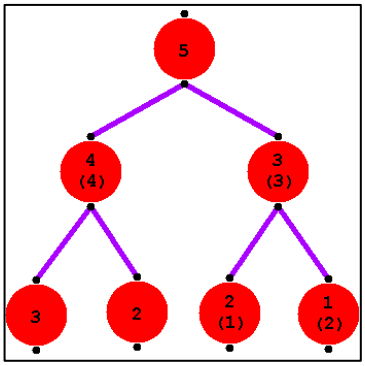
The standard metric is designed to return distances based on how close an individual is to the real value. We included this metric as it is intended to create a fitness environment which allows incorrect individuals to be assigned some fitness, creating a less harsh fitness environment and hopefully allowing for a more gradual fitness improvement. The formula is as follows



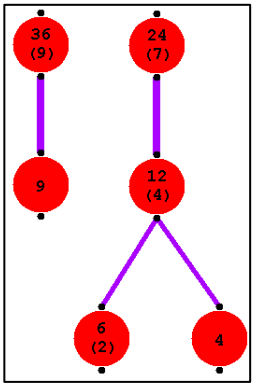
(a) Factorial tree fragments.



(b) Fibonacci tree fragments.



(c) Lucas tree fragments.



(d) Refactoring tree fragments

Figure 11 Various Tree Fragments

$$\text{dist}(a, b) = \text{abs}(a - b).$$

Unlike the other metrics, the standard metric did not perform poorly for any of the problems; rather its performance was consistently average, except for the refactoring problem where it was able to run faster. This does not suggest that it is necessarily a bad metric; on the contrary this suggests that it is an ideal metric to use, as the results imply that it is in fact the better metric to use in a general sense where there are a lot of problems with an unknown structure, as it should not perform overly badly for any problem.

Finally, the squared metric is designed based on the same ideas as the standard metric; however it was intended to create penalties for individuals that are heavier the further from the correct value they are. The formula is as follows

$$\text{dist}(a, b) = (a - b)^2.$$

Table 1 Results with tree-fragment fitness

Distance Metric		All-or-nothing			Standard			Squared		
Problem		*Correct runs*	*Average number of generations*	*Average runtime (seconds)*	*Correct runs*	*Average number of generations*	*Average runtime (seconds)*	*Correct runs*	*Average number of generations*	*Average runtime (seconds)*
Factorial	Phase 1		5	3.8740		5	3.9503		5	3.8667
	Phase 2		5	7.3491		5	7.3504		5	7.4357
	Total	100%	10	11.2231	100%	10	11.3007	100%	10	11.3024
	Std Dev.		0	0.6760		0	0.2695		0	0.5833
Fibonacci	Phase 1		10	7.6728		10	7.7045		10	7.5935
	Phase 2		5	7.8823		5	7.7903		5	7.7338
	Total	100%	15	15.5551	100%	15	15.4948	100%	15	15.3273
	Std Dev.		0	0.3970		0	0.3090		0	0.6157
Fibonacci3	Phase 1		15	11.5853		15	11.2272		15	11.1638
	Phase 2		97	112.9543		73	82.3500		65	72.6049
	Total	78%	112	124.5396	86%	88	93.5772	96%	80	83.7687
	Std Dev.		75	88.8708		63	75.3035		65	76.8769
Lucas	Phase 1		10	7.6448		10	7.5723		10	7.5713
	Phase 2		78	82.6677		87	87.2065		88	88.7766
	Total	82%	88	90.3125	80%	97	94.7788	74%	98	96.3479
	Std Dev.		46	49.9038		46	49.7765		46	48.4321
Refactoring	Phase 1		5	3.7477		5	3.8759		5	3.9058
	Phase 2		9	9.1274		6	6.2518		7	7.4973
	Total	100%	14	12.8751	100%	11	10.1277	100%	12	11.4031
	Std Dev.		9	7.8248		3	2.9819		5	4.6066

Table 2 Results without tree-fragment fitness

Distance Metric		Standard		
Problem		*Correct runs*	*Average Number of generations*	*Average runtime (seconds)*
Factorial	Total	100%	19	8.6189
	Std Dev		13	6.2196
Fibonacci	Total	20%	216	104.8162
	Std Dev		191	94.0523
Fibonacci3	Total	0%	413	207.9414
	Std Dev		144	74.2822

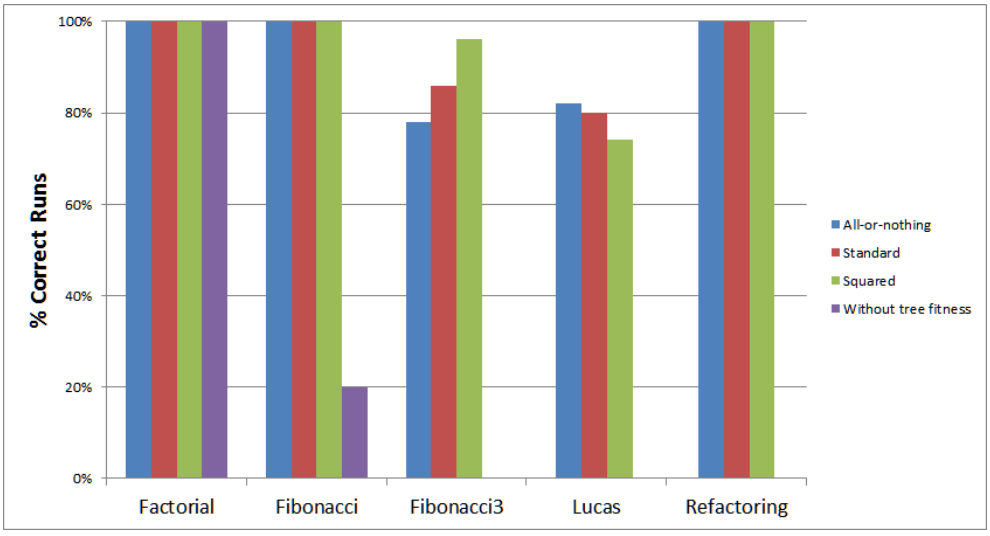


Figure 12 Correct runs for each problem.
No correct runs were obtained for Fibonacci3 without tree fitness.
Results were not performed without tree fitness for Lucas and Refactoring.

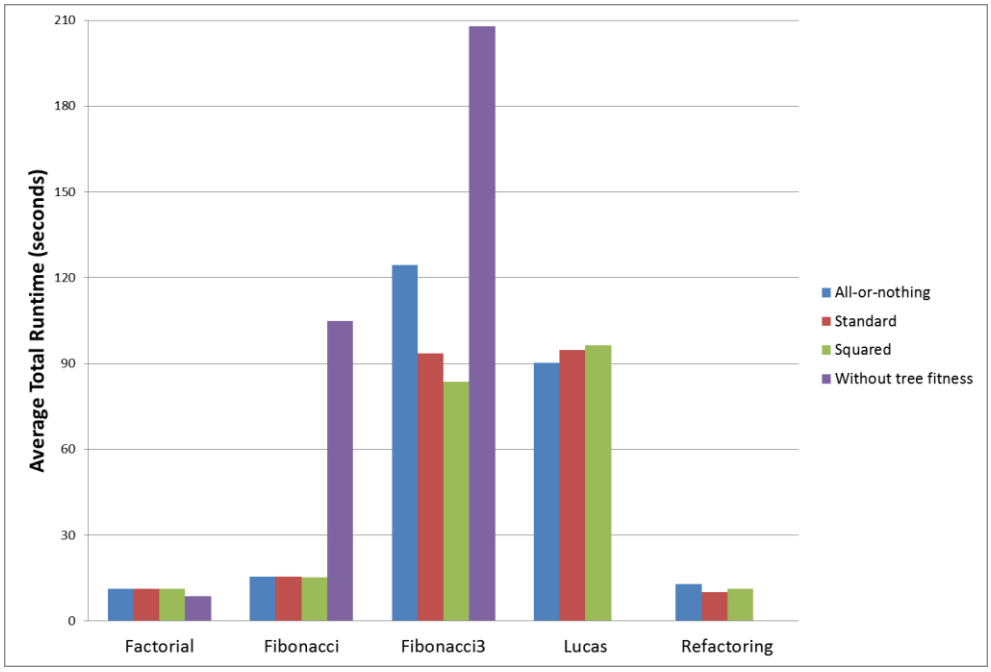


Figure 13 Average total runtime for each problem

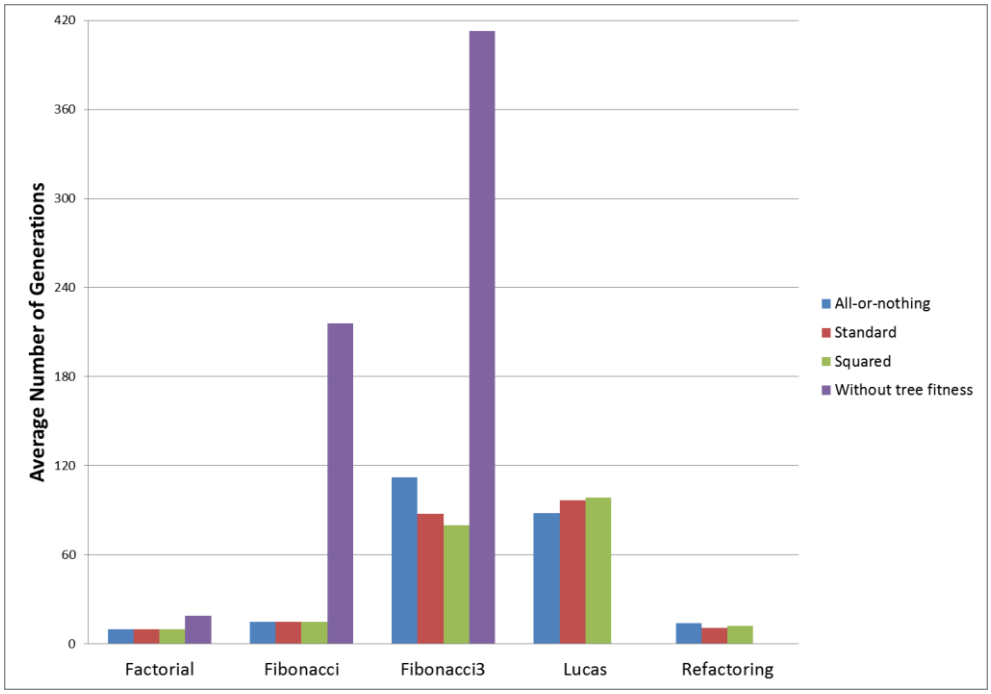


Figure 14 Average number of generations to solve each problem

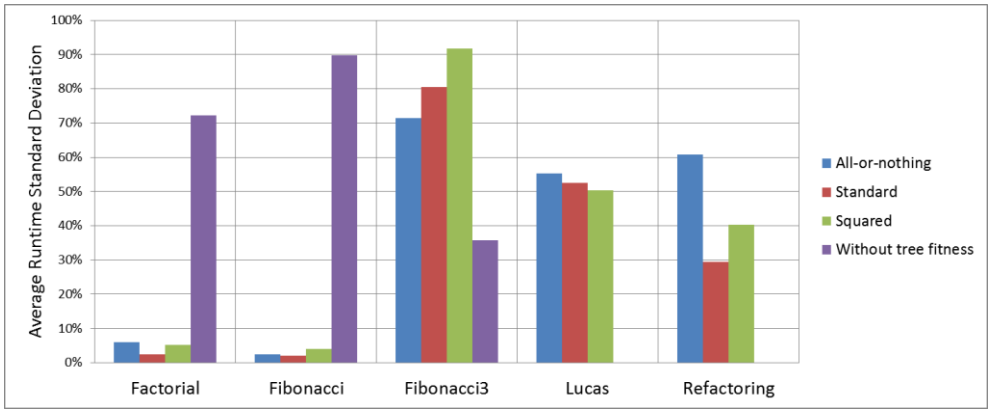


Figure 15 Standard deviations of the average runtimes for each problem

This metric performed best for Fibonacci and Fibonacci3, the problems with a majority of the complexity in the recursive call, which is as to be expected based on its design and intended purpose. As there is less emphasis on the base cases of those problems – in the case of both they can easily be represented with a single base case, thus much of the output variation would come from differentiation amongst the recursive case operators, which means that heavier penalties for a more incorrect solution will help steer the evolution in the right direction.

5.2 Problem set results

The results for factorial are as to be expected due to the fact that it is a trivial problem to solve. It can easily be solved in a short time and with a small number of generations regardless of the use of the use of tree fitness. In fact, the additional calculations and setup times for the additional GA run to calculate the recursive case causes our algorithm to be marginally slower than without tree fitness. However we are able to obtain the correct result in almost half the number of generations. Additionally, it is worth noting that as it is such a simple problem, there is very little differentiation amongst the different distance metrics.

The results of the Fibonacci experiments with tree fitness are very similar in nature to those of factorial. Again they are as expected due to the problem's simplicity and there is very little differentiation amongst the distance metrics. We are able to find a solution almost as quickly as with factorial. In contrast to factorial however, and even though Fibonacci is still a relatively simple problem, basic input-output testing shows how poorly it performs, running almost seven times slower than our algorithm with almost 15 times as many generations required to find a solution that can solve the given input-output pairs. Whilst our algorithm is able to consistently generate solutions that match the actual Fibonacci algorithm, when only using input-output testing, only 20% of the evolved solutions were able to generalise past the given input-output pairs.

The Fibonacci3 results show how much more difficult the extra recursive calls make the evolutionary process. Even with the use of tree fitness, the runtimes are significantly longer than those achieved for standard Fibonacci, and they are nowhere near as consistent; with some runs being able to find the solution relatively quickly, in times below a minute, and other runs struggling and unable to find a solution at all within the generation limit. This is an unfortunate side effect of the heuristic nature of a GA and GP in particular; with a problem space as chaotic and inconsistent in nature as a space of programs, it is very difficult to prevent the algorithm from getting stuck in local minima. With the addition of an extra recursive case, it creates a combinatorial explosion which makes the space large and much more difficult to control. That being said, the runtimes and number of generations that we achieved were still significantly better than those without the use of tree fitness. It is worth noting,

however, that the runtimes without tree fitness for Fibonacci3 are longer partly due to the fact that for this problem we allowed the algorithm to run for a larger number of generations with the hopes that it might be able to eventually converge on the solution. However as the results show, even with the extra allowance - some runs with up to 300 extra generations – it was still unable to find a solution to the problem which even matched the input-output pairs.

The Lucas series surprised us a bit. Whilst we expected it to take longer to solve than the related Fibonacci, we did not expect that the accuracy would be as low as it was. Unfortunately upon analysis this is because even with the use of the tree fitness to discover the recursive calls in phase 1, the problem's complexity comes from its backwards base cases, which means that phase 2 still has to do a majority of the work. Unfortunately this means that the majority of the problem degrades into a relatively uninformed search, which attributed the lower accuracy and higher runtimes.

The experimental results from evolving a solution to refactoring have shown an interesting finding. Even though the problem itself is inherently more complex than that of factorial and Fibonacci, the results show that we are able to find a solution in a comparably small run time and number of generations. This is partially due to the reasonable constraints that we placed on the loop structure grammars, allowing us to shrink the search space substantially, whilst still maintaining a degree of generality. It is also influenced by the fact that the grammar that we developed largely plays to the strengths of GE, in that whilst there are a number of different rules that have to be evolved at once, most of these rules have a shallow tree depth, allowing for easier evolution.

6 Future research directions

After developing this algorithm to work reliably for the given subset of single parameter recursive problems, it is clear that some there are some future research prospects that are presented by some further, more advanced applications. First and foremost, there are numerous more complex single parameter recursive problems that exist in the world that we could seek to develop the system to solve, such as the cannonballs problem [16], involving counting the minimum number of pyramids that can be formed from n cannonballs, or the counting change problem [17], which involves counting the number of ways that you can make up a given monetary value from a number of different value coins. Being able to reliably evolve these would provide a greater usability to the application. Next there is the extension of the algorithm such that is able to handle the evolution and evaluation of multiple-parameter recursive problems such as the greatest common divisor problem. This task would require a significant amount of modification to the grammars, the evaluative functions and the GUI analysis algorithm, although again it would benefit the research greatly by proving the wider variety of applications of this method of recursive evolution. Looking past these mathematical applications is the adaptation of the algorithm to recursive problems that are not explicitly based on integers, such as the famous even-n-parity problem, for which has often proven to be difficult to reliably evolve solutions that can generalise. Finally, there is one research pathway, somewhat unrelated to the field of GP; the development of an image recognition algorithm that is able to read in hand-drawn call tree fragments that have been scanned into the application. This would allow the system to be more powerful a tool for producing recursive solutions based on call trees.

References

[1] J. R. Koza, “Hierarchical genetic algorithms operating on populations of computer programs,” in *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 1*, ser. IJCAI’89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 768–774. [Online]. Available: http://dl.acm.org/citation.cfm?id=1623755.1623877

[2] W. Langdon, S. Gustafson, and J. Koza. (2012) The genetic programming bibliography. Accessed 10/11/2012. [Online]. Available: http://www.cs.bham.ac.uk/~wbl/biblio/

[3] C. Ryan, J. J. Collins, and M. O’Neill, “Grammatical evolution: Evolving programs for an arbitrary language,” in *Proceedings of the First European Workshop on Genetic Programming*, ser. EuroGP ’98. London, UK, UK: Springer-Verlag, 1998, pp. 83–96. [Online]. Available: http://-dl.acm.org/citation.cfm?id=646806.706289

[4] L. Spector, J. Klein, and M. Keijzer, “The push3 execution stack and the evolution of control,” in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, ser. GECCO ’05. New York, NY, USA: ACM, 2005, pp. 1689–1696. [Online]. Available: http://doi.acm.org/10.1145/-1068009.1068292

[5] TopCoder, Inc. (2010) TopCoder problem "Refactoring" used in SRM 216. Accesed 14/11/2012. [Online]. Available: http://topcoder.bgcoder.com/print.php?id=669

[6] J. R. Koza, “Hierarchical automatic function definition in genetic programming,” in *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems*. Morgan Kaufmann Publishers Inc, 1992.

[7] T. Yu and C. Clack, “Recursion, lambda abstractions and genetic programming,” in *Genetic Programming 1998: Proc. of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin*, 1998. [Online]. Available: http://ci.nii.ac.jp/naid/10019457709/en/

[8] T. Yu, “Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction,” *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 345–380, Dec. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1012926821302

[9] T. Yu, “A higher-order function approach to evolve recursive programs,” in *Genetic Programming Theory and Practice III*, ser. Genetic Programming, T. Yu, R. Riolo, and B. Worzel, Eds. Springer US, 2006, vol. 9, pp. 93–108. [Online]. Available: http://dx.doi.org/10.1007/0-387-28111-8_7

[10] J. Miller and P. Thomson, “Cartesian genetic programming,” in *Genetic Programming*, ser. Lecture Notes in Computer Science, R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin, and T. Fogarty, Eds. Springer Berlin Heidelberg, 2000, vol. 1802, pp. 121–132. [Online]. Available: http://-dx.doi.org/10.1007/978-3-540-46239-2_9

[11] S. Harding, J. F. Miller, and W. Banzhaf, “Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing,” in *Proceedings of the 12th European Conference on Genetic Programming*, ser. EuroGP ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 133–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01181-8_12

[12] M. Wall. (2009) GAlib: A C++ Library of Genetic Algorithm Components. Accessed 10/11/2012. [Online]. Available: http://lancet.mit.edu/ga/

[13] M. Nicolau. (2007) libGE Project Homepage. Accessed 10/11/2012. [Online]. Available: http://bds.ul.ie/libGE/index.html

[14] M. Keijzer, C. Ryan, M. O'Neill, M. Cattolico, and V. Babovic, "Ripple crossover in genetic programming," in *Proceedings of the 4th European Conference on Genetic Programming*, ser. EuroGP '01. London, UK, UK: Springer-Verlag, 2001, pp. 74–86. [Online]. Available: http://dl.acm.org/-citation.cfm?id=646809.759655

[15] M. L. Wong and K. S. Leung, "Evolving recursive functions for the even-parity problem using genetic programming," in *Advances in genetic programming*, P. J. Angeline and K. E. Kinnear, Jr., Eds. Cambridge, MA, USA: MIT Press, 1996, pp. 221–240. [Online]. Available: http://dl.acm.org/-citation.cfm?id=270195.270216

[16] TopCoder, Inc. (2010) TopCoder problem "CannonBalls" used in SRM 288. Accesed 14/11/2012. [Online]. Available: http://topcoder.bgcoder.com/print.php?id=1154

[17] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.