COMP390 Honours Year Computer Science Project

Robin Lockyer

201148882

Evolving a Sorting Algorithm with SNGP

Specification

1. Project Description

This project is being done for my project supervisor Dr David Jackson. The aim of this project is to attempt to evolve a sorting algorithm using SNGP and, if successful, compare the efficiency of evolving sorting algorithms using GP with evolving sorts with SNGP.

Genetic programming (GP) is a technique for creating programmes not by writing them by hand, but instead by creating a population of random programmes and modifying them using an evolutionary algorithm [1]. The desired result is that after several generations a programme that performs well at a given task is generated. GP has previously been used to successfully evolve sorting algorithms [2].

Single node genetic programming (SNGP) is a variation on GP invented by Dr Jackson [3] which structures the population of programmes in a manner that allows the use of dynamic programming when computing the result of the programmes in an effort to more efficiently generate a working solution.

2. Statement of Deliverables

2.1. Anticipated Documentation

I anticipate the documentation that will be required will include:

- Specification
- Design Document, including:
  - Description of anticipated experiments
  - Descriptions of algorithms used for SNGP implementation
  - Description of data structures used
  - Description of methods of evaluation
- Project report, including :
  - Analysis of the effectiveness of SNGP in evolving sorting algorithms
  - Summary of results
- Instructions on how to use SNGP implementation
- Commented source code for final SNGP implementation
- Full list of experiments and results

If I am successful in evolving an SNGP sort I will then need to evolve a sorting algorithm using GP for comparison, so the design document will also include:

- Commented source code for final GP implementation
- Descriptions of algorithms used for GP implementation
- Analysis and comparison of SNGP and GP in regards to sorting algorithms

2.2. Anticipated Software

The software I anticipate producing is an implementation of SNGP that is capable of evolving a sorting algorithm. I may also produce several variations of this implementation with different initial conditions, measures of fitness, maximum number of generations, etc. to compare and evaluate the best way to implement SNGP to evolve a sort.

If I am successful in evolving a sort I will also re-implement previous work in evolving a sorting algorithm in GP as a comparison to SNGP.

2.3. Anticipated Experiments

The exact parameters for SNGP to successfully evolve a sort are unknown it has not been done before. As such I will have to experiment with several variables to find values that work. These variables include:

- Fitness test
- Measure of sortedness
- Population size
- Number of generations
- Number of runs
- Set of terminals
- Set of operations

I may also experiment with evolving different kinds of sort, such as recursive or iterative algorithms, or giving the algorithm working memory in the form of a stack to allow it to perform more complex sorts such as merge sort.

2.4. Methods of Evaluation

If I succeed in evolving a sorting algorithm I will evaluate it by comparing the SNGP implementation to both previous SNGP implementations for different problems and to GP implementations that evolve sorting algorithms.

The criteria for evaluation include:

- The rate at which runs successfully produce solutions
- The time taken to perform 100 runs
- Minimum, maximum and average size of solutions generated
- Average number of programme nodes evaluated per solution
-

No third party evaluation will be necessary, and as such no human participants are needed.

3. Conduct of the Project and Plan

3.1. Preparation

For background research I have read:

- A Field Guide to Genetic Programming [1]
- A New, Node-Focused Model for Genetic Programming [3]
- Single Node Genetic Programming on Problems with Side Effects [4]
- Generality and Difficulty in Genetic Programming: Evolving a Sort [5]
- Evolving a Sort: Lessons in Genetic Programming [2]

The first of these is an introductory book to genetic programming. I have read Part 1 of this book, which explains what genetic programming is and the algorithms and data structures used. The

remaining parts go into detail on alternate representations of programmes and advanced techniques which are not necessary for this project as I am sticking to SNGP.

The second and third sources are papers written by Dr Jackson on SNGP. The first explains what SNGP is and how it differs and compares from GP. The second reiterates some of the same information but focuses on comparing the two techniques in problems with side effects.

The last two sources detail previous work on evolving a sort using regular genetic programming, which gives me a starting point for the SNGP implementation as I can use similar operations and fitness measures.

3.2. Data Required

The evaluation of the generated programmes will require test data. This data will most likely be randomly generated arrays of integers of varying lengths and sortedness. This variation is needed to ensure that the programmes generated are able to sort any input, not just the specific test data the algorithm is given.
No human data will be used.

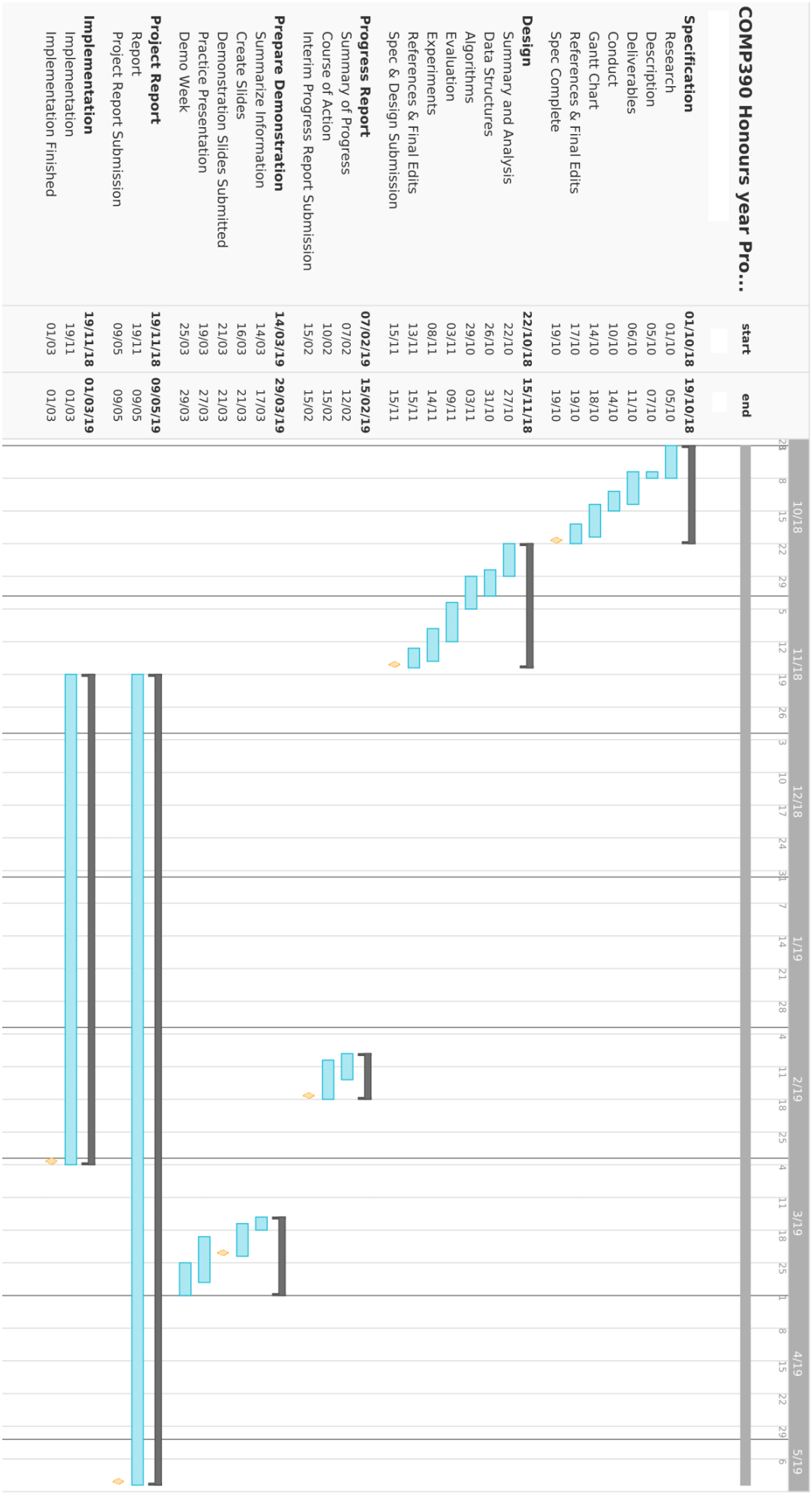
3.3. Design

The design methodology will mostly be in line with the functional design methodology as I will be using the C programming language, which does not easily support object oriented design. There is also no database involved and so the traditional (i.e. web based with DB backend) design methodology is also not applicable.

3.4. Implementation

For the implementation of the project I will be using the C programming language. This is because previous work done on SNGP has also been implemented in C and using the same language will make it easier to compare my results. In addition, Dr Jackson has allowed me to use some of his C code from his work on SNGP as a basis for my own.
No special hardware is required for this project. All that is needed is a computer able to compile and run C code. The software I will be using the Microsoft Visual Studio IDE for its ability to compile and debug C code.

3.5. Project Plan



3.6. Risk Assessment

One of the major challenges I anticipate encountering during this project is the uncertainty of whether SNGP is capable of creating a sorting algorithm in a reasonable number of generations. However, if it turns out SNGP is not suitable for evolving sorting algorithms I can modify my project to analyse why this is the case.

The randomness inherent to GP makes it difficult to replicate results. Luckily the code provided by Dr Jackson allows for runs to be repeated by providing an identical seed number of the random number generator, so I can use his methods to implement a similar system.

Another risk is that a single run of any form of GP is not sufficient to determine how successful the implementation is at solving the problem, so I will have to have many runs to become certain of my results.

A risk of using the C programming language is that is does not easily allow for design methodologies such as object orientation, which can make difficult to structure a programme in an intuitive and maintainable way.

4. Bibliography

[1] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[2] K. E. Kinnear, “Evolving a Sort : Lessons in Genetic Programming,” 1993.

[3] D. Jackson, “A New, Node-Focused Model for Genetic Programming,” in *Genetic Programming*, 2012, pp. 49–60.

[4] D. Jackson, “Single Node Genetic Programming on Problems with Side Effects,” in *Parallel Problem Solving from Nature - PPSN XII*, 2012, pp. 327–336.

[5] K. E. Kinnear Jr., “Generality and Difficulty in Genetic Programming: Evolving a Sort,” in *Proceedings of the 5th International Conference on Genetic Algorithms*, San Francisco, CA, USA, 1993, pp. 287–294.

COMP390 Honours Year Computer Science Project

Robin Lockyer

201148882

Evolving a Sorting Algorithm with SNGP

Design

1. Summary of Proposal

Genetic programming (GP) is a technique for creating programmes not by writing them by hand, but instead by creating a population of random programmes and modifying them using an evolutionary algorithm [1]. The desired result is that after several generations a programme that performs well at a given task is generated. GP has previously been used to successfully evolve sorting algorithms [2].

Single node genetic programming (SNGP) is a variation on GP invented by Dr Jackson [3] which structures the population of programmes in a manner that allows the use of dynamic programming when computing the result of the programmes in an effort to more efficiently generate a working solution.

This project aims to determine if SNGP is capable of evolving a sorting algorithm and, if so, how well suited SNGP is at evolving such algorithms when compared to standard GP.

To implement this project, I will be using the C programming language. This is because previous work done on SNGP has also been implemented in C and using the same language will make it easier to compare my results. In addition, Dr Jackson has allowed me to use some of his C code from his work on SNGP as a basis for my own.

2. Project Design

2.1. Description of Standard GP

The standard method of genetic programming is described here for comparison with SNGP. While I may create a standard GP implementation if I am successful with SNGP, I will not go into exact technical detail as the main focus of the project is the SNGP implementation.

2.1.1 Preparation

The first step in GP is to define the problem to be solved and the operations the algorithm is to use. This is done by the deciding on the following:

Measure of fitness
This is how the algorithm decides which members of the population perform better at the given task than others. The exact criteria for fitness and whether a higher or lower fitness is preferable is specific to the problem.

Set of functions

These are the primitive operations that the GP algorithm will combine to create the population. Elements of this set have arity strictly greater than 0.

Set of terminals
These are a collection of the input variables to the problem, constants, and nullary functions. The elements of the terminal set can be considered to be functions of arity 0.

It is worth noting that GP often works best when the functions and terminals are of the same type [ref: Field guide]. This is because any function or terminal can become an operand for any other function, so type consistency ensures all generated programmes are at least valid. An alternative is modifying mutation operations to prevent them from creating invalid programmes.

Stopping Condition
This will be true when the algorithm should terminate. This could be when enough individuals of sufficient fitness are generated, after a certain number of generations or some other problem specific condition.

Population Initialisation
The method to create the initial population. There are several methods this, but "ramped half-and-half" is a common method that produces good results [ref: Koza] that I will detail later on in section 2.1.3.

Selection
The method by which individuals are chosen for genetic operations. Simply selecting the fittest individuals to reproduce can lead to low diversity in the resulting population [ref: field guide]. A method to prevent this is to randomly select a subset of the population with probability based on the individual's fitness, pick the fittest individual from that subset, and repeat until enough members of population have been chosen for reproduction. This method introduces some randomness that prevents a single strategy from dominating while still preferring candidates that perform well.

Genetic Operations
These are the random operations that alter and combine individuals in the population to create a new generation. There are many possible genetic operations, some of which I will describe in section 2.1.4.

2.1.2 Algorithm

There is no one specific algorithm for genetic programming. A general approach is described here based on the description of genetic programming found in [ref: field guide]. In later sections I will describe potential specific implementations

The pseudocode for a general genetic programming algorithm is as follows:

```
Initialise population of random programmes made of
elements of Function and Terminal sets

DO:

      FOR EACH individual IN population:

            execute(individual)

            evaluateFitness(individual)


      Select small subset of population based on each
      individual's fitness

      Create new population by performing genetic
      operations at random to selected subset of population


UNTIL stopCondition() == FALSE

OUTPUT best individual created
```

2.1.3 Population Initialisation

One of the simplest and most effective methods for creating a random initial population is the ramped half-and-half method [1].It is a combination of two other methods, "full" and "grow".

In the full method a tree depth is specified. A random function from the function set is chosen as the root and then functions are selected at random as children for already selected functions. If the children of a function would be at the specified depth, a random terminal is chosen as opposed to a random function. This results in trees where all leaf nodes are at maximum depth. Note that the resulting trees may have a different shape (i.e. have a different number of nodes) if the function set contains functions of different arities.

Examples of trees generated by full with depth 2 from the set of binary functions {f,g,h} and set of terminals {a,b,c} are shown in Figure 1.

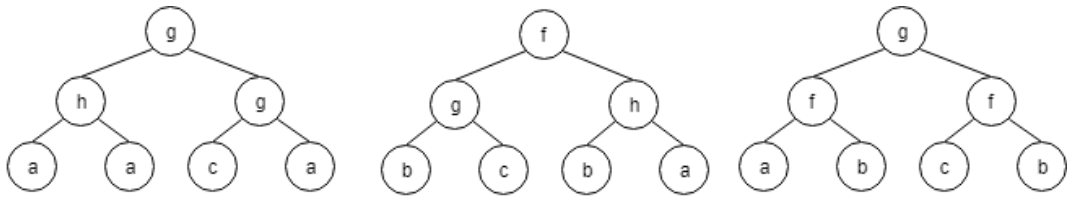


Figure 1: Examples of Trees Created by the "Full" Method

In the grow method a maximum tree depth is specified. It differs from grow in that instead of selecting from just the function set, the grow method may choose an element from the terminal set at any point at random. Again, if the children of a function would be at the maximum depth a terminal *must* be chosen. This allows for more variety in number of nodes in the trees than full, but the resulting trees tend to be smaller.

Examples of trees generated by grow with depth 2 from the set of binary functions {f,g,h} and set of terminals {a,b,c} are shown in Figure 2.

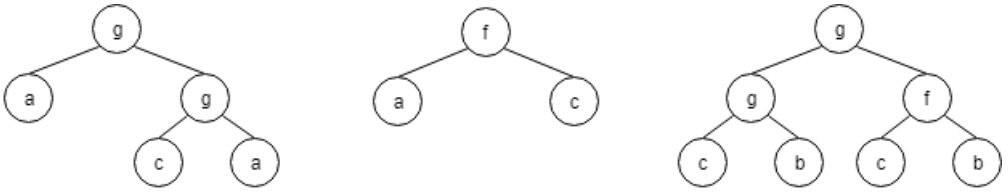


Figure 2: Examples of Trees Created by the "Grow" Method

Ramped half-and-half generates creates half the population with grow and the other half with full, using a range of values for the depth. This ensures that the initial population is sufficiently varied.

2.1.4 Genetic Operations

There are many possible genetic operators that can be used in GP. I will describe here some of the most common ones.

Reproduction

An individual is copied over to the new population with no changes made

Mutation

A copy of an individual is made. A random node in the tree is selected. The node and its subtree are deleted and replaced by a new random subtree. This new tree is added to the new population.

Point Mutation

A copy of an individual is made. A random node in the tree is selected. The node is replaced by a different node of the same arity taken from the function or terminal set. This new tree is added to the new population.

Crossover

Copies of two individuals (X and Y) are made. A random node is selected in each tree. The subtree rooted at the node in X is replaced by the subtree rooted at the chosen node in Y. The modified X tree is added to the new population and the unused tree subsets are discarded. The nodes are selected probabilistically, often with a higher change of selecting an internal node [ref: field guide]. This increases the chance larger subtrees are exchanged instead of individual leaves.

2.2. Description of SNGP

Single Node Genetic Programming was first suggested by Dr David Jackson, and my description of it is based off of his work [3, 4]. SNGP is described here without any specific problem domain in mind.

2.2.1 Overview

SNGP structures the population as an interlinked series of primitive functions and terminals. At each node a complex function can be computed by applying its function to the result of the complex functions represented by its children.
Since the SNGP population is interlinked, any genetic operation applied will change the complex function of multiple nodes simultaneously. This means that the genetic operations are effectively applied to the population as a whole.
After the application of a genetic operation the complex function at each node is evaluated and the whole population is given a fitness. A hill climbing approach is taken, where if the resulting fitness is equal or higher than the population's previous fitness then the algorithm continues using the new population. Otherwise the population is reverted back to its previous state.
At a certain point a problem specific condition is met, the algorithm stops, and the complex function with the best fitness is then output.

2.2.2 Preparation

Preparation for SNGP is similar to preparation for GP in deciding on the terminal set, function set, and the measure of fitness, but there is only one genetic operation and one method of initialisation.
The choices that have to be made before starting SNGP are:

Measure of fitness
This is how the algorithm decides which members of the population perform better at the given task than others. The exact criteria for fitness and whether a higher or lower fitness is preferable is specific to the problem.

Set of functions
These are the primitive operations that the SNGP algorithm will combine to create the population. Elements of this set have arity strictly greater than 0.

Set of terminals
These are a collection of the input variables to the problem, constants, and nullary functions. The elements of the terminal set can be considered to be functions of arity 0. T is the size of the terminal set.

Like GP, SNGP benefits from the function set and terminal set being type consistent.

Stopping Condition

This will be true when the algorithm should terminate. This could be when enough individuals of sufficient fitness are generated, after a certain number of generations or some other problem specific condition.

Selection

In SNGP selection works differently to GP. Instead of selecting a few individuals to perform genetic operations on, a single genetic operation is applied to the population as a whole. The resulting population is then evaluated as a whole and a hill climbing approach is used to decide if this change is preferable.

The new population will either be considered equal or fitter to the old, and kept, or worse at which point the population is restored to its state before the operator was applied.

The choice here is deciding how to evaluate the population as a whole. Dr Jackson's work describes two possible methods: SNGP/A and SNGP/B [ref: Dave2]. SNGP/A considers the fitness of the population to be the average fitness across all individuals. SNGP/B the fitness of the population to be the best fitness of any individual in the population.

2.2.3 Population Initialisation

Each member of the population in SNGP is comprised of the following:

- **Primitive**: The primitive function this node represents. Either an element from the set of functions or the set of terminals.
- **Fitness**: A number representing how useful at the given task this individual is. Whether a higher or lower value for fitness is preferred is determined by the problem
- **Operands**: An array containing the members of the population that are used as operands for the primitive this individual represents. Its size equal to the arity of this node's primitive.
- **Predecessors**: An array containing the nodes that use this individual as an operand. For terminal nodes this is left empty as it will not be used.
- **Outputs**: An array containing the outputs created when this node is evaluated.

The population is stored in an array of fixed size S > T.

For nodes with an index less than T, their primitive will be initialised to a member of the terminal set that is not already in use by node. The result is that the first T elements contain every terminal with no repetitions.

For the remaining nodes with index greater than or equal to T their primitive is initialised to a random element of the function set. The elements of each individual's operands array are set to random node with a lower index. When a node is selected to be an operand and is not a terminal, its predecessors array is updated to reflect this.

The result is that the nodes are structured as a directed acyclic graph. The array they are stored in is a topological ordering of the nodes, with members of the terminal set occupying the lowest positions. At each node the subgraph rooted there is the programme associated with that node.

An example of an initialised population with S = 8 is shown in graph form in Figure 3, in array form in Figure 4, and the programmes represented within the population are written out explicitly in Figure 5. The terminal set is {a,b,c}, and {f,g,h} is the set of binary functions.

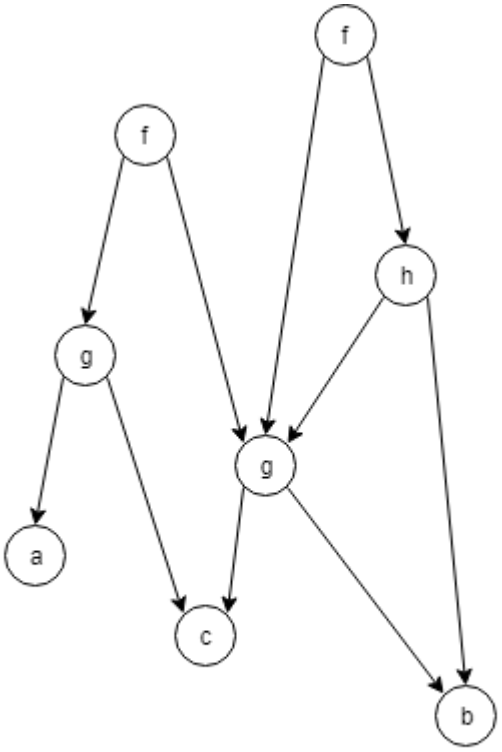


Figure 3: An Example SNGP Population in Graph Form

Index	0	1	2	3	4	5	6	7
Primitive	b	c	a	g	g	h	f	f
Operands	-	-	-	1,0	2,1	3,0	4,3	3,5
Predecessors	-	-	-	7,6,5	6	7	-	-

Figure 4: An Example SNGP Population in Array Form

Index	Programme
0	b
1	c
2	a
3	g(c,b)
4	g(a,c)
5	h(g(c,b),b)
6	f(g(a,c)g(c,b))
7	f(g(c,b), h(g(c,b),b))

Figure 5: Explicit Representation of Programmes in Figure 4

The members of this initial population can then be evaluated from lowest index to highest index, each node storing the result of their evaluation in their outputs array. This allows for a form of dynamic programming. When evaluating a node its operands must also be evaluated. Instead of re-evaluating a node multiple times the pre-calculated values in the outputs array can be used instead, reducing the amount of calculation needed to evaluate the population.

2.2.4 Genetic Operations

There is only a single genetic operation in SNGP, successor mutate, abbreviated to “smut”.

In smut a random non terminal node is selected. One of the nodes operands is changes to another random node with a lower index in the population array. The relevant predecessor arrays are altered to reflect this change. The effect of the smut operation is shown in Figure 6.

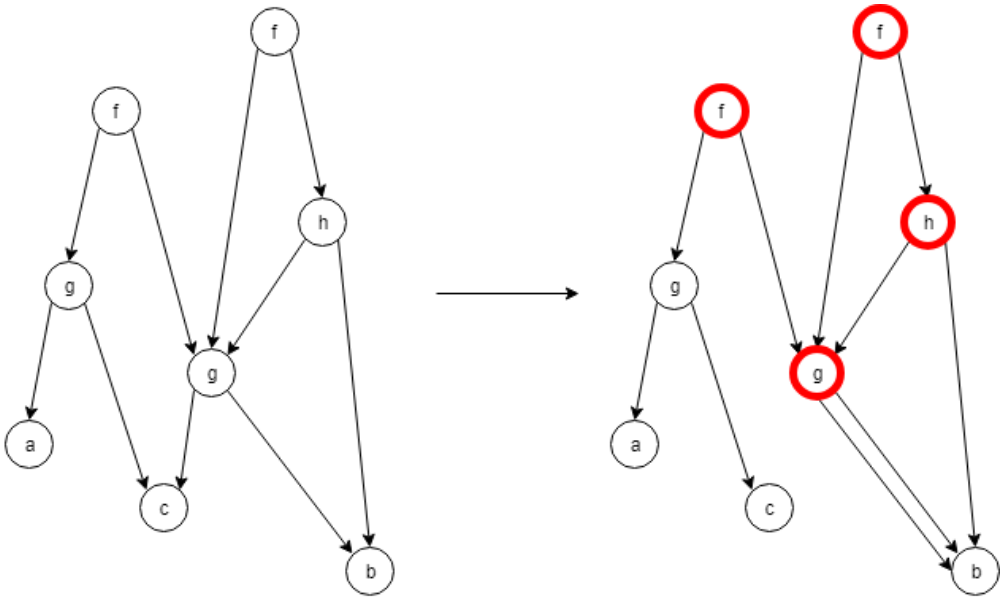


Figure 6: The Effect of the Smut Operation on an SNGP Population

The smut operation require the node to be re-evaluated, along with all the other nodes that make use of it. Only the nodes with a higher index will need to be updated, and the exact nodes can be determined by the predecessor arrays. In Figure 6 only the four highlighted nodes need to be evaluated again.

Note that the h node may be evaluated twice if its predecessor is evaluated first. We can prevent this by making sure that when re-evaluating nodes they are re-evaluated in ascending order of index. As the operands of a node must have a lower index than the node itself, the operands are always evaluated before their predecessors. In the case of Figure 6 the highlighted nodes are evaluated in the order g, h, f, f.

2.2.5 Pseudocode

This section contains the pseudocode for algorithms that are common to all implementations of SNGP. See Section 2.3 for algorithms relating to this specific project.

Initialisation of the Population

```
LET T be the size of terminalSet
LET p be an array of Nodes of size S > T
FOR i=0 to i=S-1:
      LET n = p [i]
      IF i < T:
            n.primitive = terminalSet[i]
      ELSE:
            n.primitive = functionSet.randomElement()
      n.operands = array of same length as the arity of n.primitive
      FOR EACH operand IN n.operands:
            LET r be a random integer in range -1<r<i
            operand = p[r]
            IF p[r].function has arity > 0
                  p[r].predecessors.append(n)
```

Successor Mutate Operation

```
LET p be a properly initialised array containing the population
DO:
      LET oldP be a copy of p
      Apply smut operation to p
      Evaluate fitness of p
      IF fitness of p < fitness of old p:
            p = oldP
UNTIL stopCondition() == FALSE
OUTPUT best individual created
```

The SNGP Algorithm

```
LET p be an array containing the population of size S
LET i be a random integer in range -1<i<S
LET n = p[i]
LET a be the arity of n.primitive
LET r be a random integer in range -1<r<a
LET t be a random integer in range -1<t<i
IF n.operands[r].function has arity > 0:
      n.operands[r].predecessors.remove(n)
n.operands[r] = p[t]
IF p[t].function has arity > 0:
      p[t].predecessors.append(n)
```

2.2.6 Data Structures

This section described the data structures I will use for my implementation of SNGP. Since I will be using the C programming language they will mainly take the form of structs. These structures are based off those used in Dr Jacksons SNGP code.

Function and Terminal Sets

The function and terminal sets will be combined into a single enumeration. An array of structures will act as a table to provide information on arity, which is how the functions and terminals will be differentiated. The will be structured as followed:

```
enum Primitive {p1,p2,…,pn-1,pn}
```

TableEntry
Primitive primitive Int arity

```
TableEntry[] arityTable = {{p1,0},{p2,0},…,{pn-1,j},{pn,k}}
```

The Population

The population will be an array of Node structures. The Node will be the following:

Node
Primitive primitive Int fitness Int oldFitness Int[] operands Int[] predecessors

This structure follows the structure outlined in Section 2.2.3, with one additional variable to store old fitness values so it can be restored if the genetic operator is reversed due to it decreasing overall population fitness.

2.3 Evolving a Sorting Algorithm

Previous work by Kinnear has shown that conventional GP is capable of evolving a sorting algorithm [ref: Kinnear]. SNGP perform better when dealing with purely functional nodes (i.e. nodes with no side effects such as in the symbolic regression problem) [4], but it has been shown to still perform well in such problems in comparison to GP [4]. Evolving a sorting algorithm is not only a problem that makes use of side effects, but is also difficult problem for GP to solve [5]. As such it is unknown whether SNGP is a suitable method for evolving a sorting algorithm.

This section will detail the potential methods I can potentially use to make SNGP evolve a sorting algorithm.

2.3.1 Evaluation of Fitness

The fitness of a sorting algorithm is the measure of how well the algorithm sorts sequences of numbers into non decreasing order. This requires the ability to measure the change in sortedness (or unsortedness) of the sequence before and after the algorithm is executed. The there are several methods to calculate the sortedness of a sequence, including:

- $correctPos(x)$
  Counting the number of elements that are in the correct position. This method is quite poor as it does not result in enough difference between unsorted sequences as the maximum value is the length of the sequence, n, so there are only n possible values that can be assigned.

- $adjPairs(x)$
  Counting the number of adjacent pairs that are in the correct order. Again, this has a low number of possible values, in this case n-1.

- $sortedness(x)$

  Counting the number of pairs that are in the correct order. This gives a much larger possible value, n(n-1)/2, which makes differentiating between unsorted lists much easier. However, this means that the average sortedness of a sequence is proportional to the square of its length. This means that sortedness of a fully sorted sequence is much higher for longer sequences and makes it difficult to normalise the results from different length arrays.

- $unsortedness(x)$.
  This function counts the number of pairs in x that are in the incorrect order. This has the same benefits of the previous methods but has the added benefit that fully sorted sequences all have a value of 0 no matter their length. This makes it easier to compare sorted sequences, but the results are still not normalised for different length sequences. There is also an efficient algorithm to calculate this based on merge sort that is O(n log(n)).

- $\frac{unsortedness(x)}{n}$
  This has maximum value (n-1)/2, which makes it easier to compare sequences of

different lengths while still maintaining the benefits of the previous method. However, longer unsorted sequences are still penalised over shorter sequences.

- $\frac{1}{1+unsortedness(x)}$
  This is the basis of the equation used by Kinnear when evolving sorting algorithms with GP [5]. With this equation all values are in the interval (0,1] no matter what length the sequence is. A sorted sequence always has the value of 1, while increasingly unsorted sequences approach 0.

The change in sortedness can then be calculated either by dividing the old sortedness by the new or by subtracting the old from the new. It may also be useful to give penalties to algorithms that decrease the sortedness of the sequence as they are doing the exact opposite of what is required. The best method for calculating sortedness for SNGP is unknown and will have to be discovered experimentally.

It currently seems likely that the most effective way to calculate sortedness will require the calculation of the number of pairs in the sequence that are not in the correct order, also known as inversions. This can be calculated efficiently using a modified merge sort algorithm that returns the number of inversions and the sorted list, shown here:

```
LET x be an array of numbers of length n

DEFINE unsortedness(x):

      IF n <= 1:

            RETURN (0,x)

      ELSE:

            //Split x into two arrays of length n/2

            LET A = [x[0], …, x[n/2-1]]

            LET B = [x[n/2], …, x[n-1]]


            //find the number of inversions of and sort the
            //subarrays

            (I_A, A) = unsortedness(A)

            (I_B, B) = unsortedness(B)

            LET inversions = I_A + I_B


            //Sort the two arrays together while counting the
            //inversions

            LET C be an empty array

            WHILE length(A) > 0 AND length(B) > 0:

                  LET a be the first element of A

                  LET b be the first element of B

                  IF b < a:

                        C.append(b)

                        inversions += length(A)

                        B.remove(b)

                  ELSE:

                        C.append(a)

                        A.remove(a)

            //When one of A or B is empty, add the non empty
            //array to the end of C

            IF length(A) == 0:

                  C = C concatenated with B

            ELSE:

                  C = C concatenated with A

            RETURN (inversions, C)
```

Since the domain of a sorting algorithm is infinite, it is impossible to test all inputs. I will use multiple randomly generated sequence with a variety of lengths to ensure the algorithms are properly tested. To avoid overfitting I will change the test sequences after every mutation operator, i.e. one test set for every generation. I may modify the set of test sequences for each generation to ensure that they meet certain qualities such as always containing a fully sorted sequence, a completely reversed sequence, or sequences of certain lengths.

Kinnear found that using the size of the evolved programmes as part of the fitness calculation increases the chance of evolving a general sort [5], so I may decide to include that in the fitness calculation.

Normalisation of each generation's fitness is not necessary in SNGP as the methods of selection (SNGP/A or SNGP/B) do not take into account by how much a population has improved, but just if it has improved at all. It may be the case that some degree of normalisation improves the rate at which SNGP finds solutions, but I will not use normalisation techniques unless it becomes clear they are necessary or I have spare time to compare the effectiveness of methods.

I am unsure whether SNGP/A or SNGP/B will have better results in calculating the fitness of the overall population. I will have to decide on which to use through experimentation.

2.3.2 Functions and Terminals

Selection of the function and terminal sets is of great importance. I will use the sets based on those used by Kinnear when evolving sorts with GP[6] as a starting point. The sets are:

Set of Terminals

- index: an integer referring to a position in the given sequence that is manipulated by other primitives.
- length: a constant integer holding the length of the sequence being sorted

Set of Functions

- iterate(start, end, function): Sets *index* to the value of *start* and increments *index* by one until *index* equals *end* or *index* equals *length* -1. Each iteration will execute *function* once if *index* is a valid position in the sequence. This primitive will return the minimum of *end* and *len*. It may be useful to restrict the number of times this primitive can iterate to speed up evaluation.
- swap(x,y): Swaps the elements of the sequence at positions *x* and *y*. Returns *x*.
- smallest(x,y): Compares the values of the sequence at *x* and *y*. Returns the index of the smaller.
- largest(x,y): Compares the values of the sequence at *x* and *y*. Returns the index of the larger.
- subtract(x,y): Returns the value of $x - y$
- increment(x): Returns the value of $x+1$
- decrement(x): Returns the value of $x-1$

For all functions, if the arguments are invalid and would case an error they will return 0. This allows for programmes that have useless but harmless structures without crashing upon execution.

It is difficult to predict what alterations to these sets will be required for SNGP to successfully evolve a sort, but I will discuss some of my thoughts on potential primitive.

Kinnear showed that many programmes made use of expressions that always evaluate to 0 [6], so perhaps an addition constant, 0, should be added to the terminal set so that SNGP doesn't have to create such expressions. Kinnear has also shown that, while not necessary to evolve a sort, programmes that make use of conditional statements such as "if(condition, function)" tend to be smaller than those that don't [5]. This makes conditional statements a prime candidate for reducing the number of nodes needed for a sort to be evolved.

It may be possible to evolve other kinds of sort, for example variants based on merge sort, by giving using primitives that access data structures. The programme may use a stack by using push(x) and pop() primitives. The results of this approach are less certain than the purely iterative approach, and as such I will only use alternative data structures as a last resort or if I have sufficient time to compare results.

3.2.3 Other SNGP Parameters and Comparison of Results

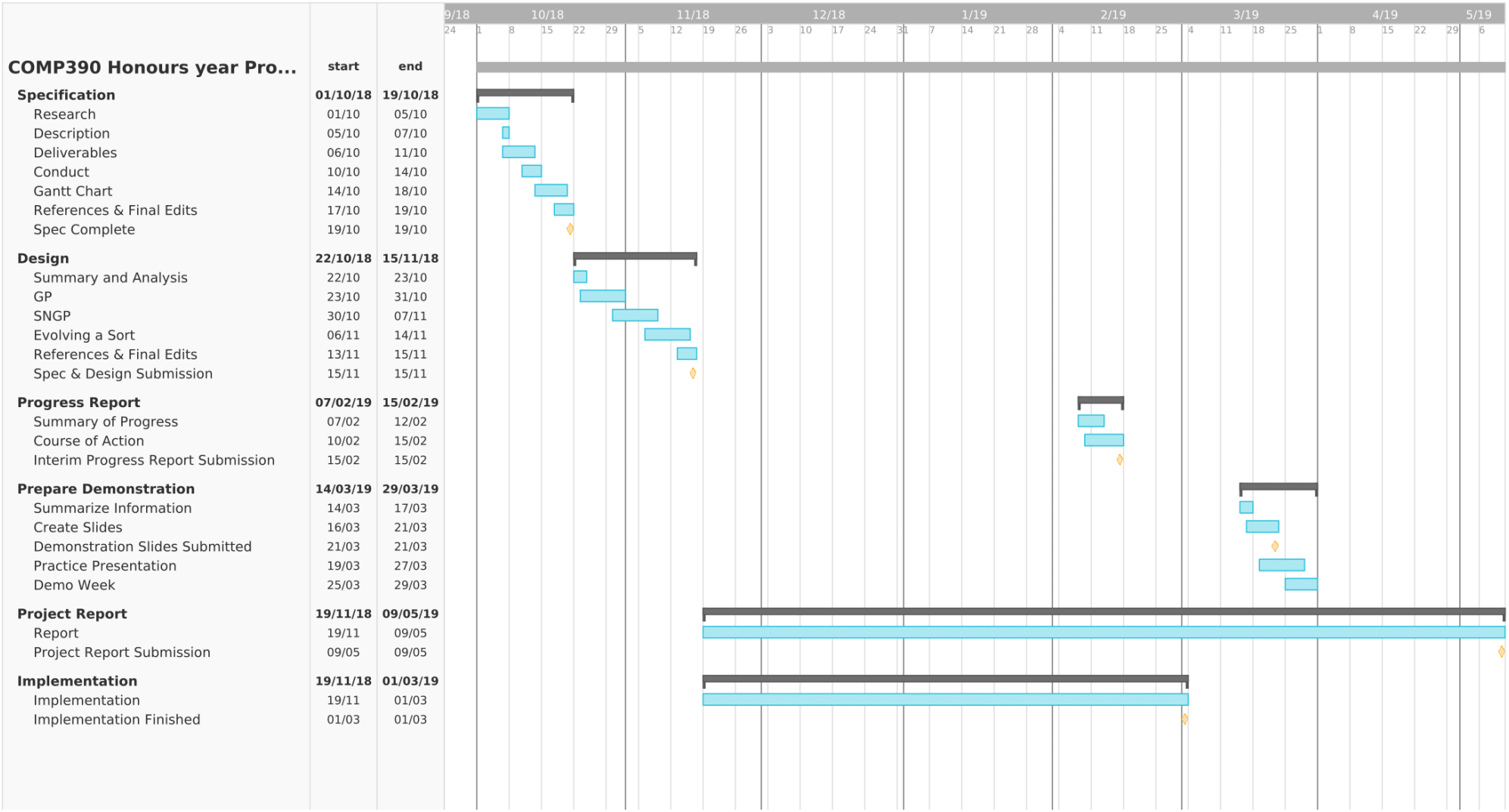
It is important that I select parameters that give a reasonable chance of a successful run of SNGP. This will require some experimentation to find the best, but initially I will use similar parameters to what Dr Jackson has shown to work for other problems with side effect in [4]. These parameters are a population size of 50 and the length of run (i.e. limit on smut applications) is 25,000.

The criteria that I will use for comparing the effect of different parameters are as follows:

- The rate at which runs produce a solution
- The average number of evaluations per solution
- The time taken to complete a constant number of runs
- The average size of generated solutions
- The maximum size of all generated solutions
- The minimum size of all generated solutions

3. Review Against Plan

The plan has been updated as follows, and as of 15/11/2018 everything is going to schedule.



COMP390 Honours year Pro...	start	end
Specification	**01/10/18**	**19/10/18**
Research	01/10	05/10
Description	05/10	07/10
Deliverables	06/10	11/10
Conduct	10/10	14/10
Gantt Chart	14/10	18/10
References & Final Edits	17/10	19/10
Spec Complete	19/10	19/10
Design	**22/10/18**	**15/11/18**
Summary and Analysis	22/10	23/10
GP	23/10	31/10
SNGP	30/10	07/11
Evolving a Sort	06/11	14/11
References & Final Edits	13/11	15/11
Spec & Design Submission	15/11	15/11
Progress Report	**07/02/19**	**15/02/19**
Summary of Progress	07/02	12/02
Course of Action	10/02	15/02
Interim Progress Report Submission	15/02	15/02
Prepare Demonstration	**14/03/19**	**29/03/19**
Summarize Information	14/03	17/03
Create Slides	16/03	21/03
Demonstration Slides Submitted	21/03	21/03
Practice Presentation	19/03	27/03
Demo Week	25/03	29/03
Project Report	**19/11/18**	**09/05/19**
Report	19/11	09/05
Project Report Submission	09/05	09/05
Implementation	**19/11/18**	**01/03/19**
Implementation	19/11	01/03
Implementation Finished	01/03	01/03

4. Bibliography

[1] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

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

[3] D. Jackson, "A New, Node-Focused Model for Genetic Programming," in *Genetic Programming*, 2012, pp. 49–60.

[4] D. Jackson, "Single Node Genetic Programming on Problems with Side Effects," in *Parallel Problem Solving from Nature - PPSN XII*, 2012, pp. 327–336.

[5] K. E. Kinnear Jr., "Generality and Difficulty in Genetic Programming: Evolving a Sort," in *Proceedings of the 5th International Conference on Genetic Algorithms*, San Francisco, CA, USA, 1993, pp. 287–294.

[6] K. E. Kinnear, "Evolving a Sort : Lessons in Genetic Programming," 1993.