COMP390 Evolving a Sorting Algorithm with SNGP

Robin Lockyer

Student ID: 201148882
Primary Supervisor: Dr. David Jackson
Secondary Supervisor: Dr. Valentina Tamma

2018/2019

Abstract

Genetic programming 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. 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.

Single node genetic programming is a variation on GP invented by Dr Jackson 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 compare the effectiveness of the two methods in evolving a sorting algorithm.

Contents

1 Introduction

This project is was done for my project supervisor Dr David Jackson. The aim of this project is to attempt to evolve a sorting algorithm using node genetic programming (SNGP) and, if successful, compare the effectiveness of evolving sorting algorithms using standard genetic programming (GP) to evolving sorts with SNGP.

The purpose of GP is to automate the creation of algorithms and programmes. This is done by applying a genetic algorithm to a population of random programmes so that successive generations of programmes improve at the desired characteristics until a functional programme is created. The standard approach to GP requires evaluating hundreds of programmes per generation over potentially thousands of generations and as such GP can take up a large amount of processing time. Several variations of GP have been created that try to reduce the amount of processing, including Linear Genetic Programming and Parallel Distributed GP [5].

SNGP is one such variation devised by Dr Jackson in *A New, Node-Focused Model for Genetic Programming* [1]. This variation makes use of a form of dynamic programming to re-use results of previously evaluated programmes. It has been shown that SNGP tends to perform better than standard GP in terms of processing time, solution rate, and solution size [1]. SNGP has reduced efficiency when dealing with problems with side-effects because this prevents re-use of evaluations, although it still performs better than GP at some problems with side effects [2].

A sorting algorithm reads and manipulates an array of integers as it executes, and so must make use of side effects. Regular genetic programming has been shown to be capable of evolving a working sorting algorithm [3, 4]. This makes evolving a sort a good problem to evaluate SNGP on as comparisons can be made with previous research.

2 Background

2.1 Standard Genetic Programming

Genetic programming allows the user to automate finding solutions to problems without the need to know much about the solutions themselves. This is done by a stochastic process where successive generations of programmes are altered to create the next generation, with the goal of each generation being an improvement on the last until a desired result is found.

In standard the standard form of GP, described in *A Field Guide to Genetic programming* [5], programmes are encoded as a tree of primitive functions and terminals. Each function has a number of child nodes equal to it's arity, and each individual child represents a single operand of the parent function. Nodes with no children represent terminal functions or constants which require no input. Figure 2.1 shows an example of a programme encoded as a tree.

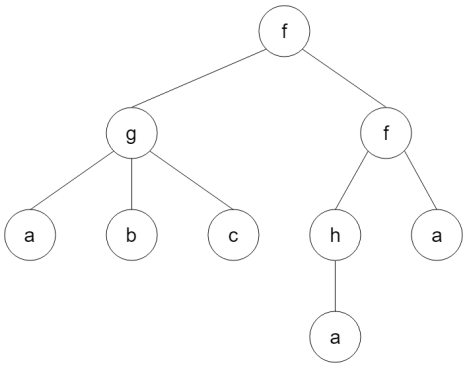

Figure 1: This tree encodes the programme f(g(a, b, c), f(h(a), a)), where f, g, and h are functions and a, b, and c are terminals.

An initial population of random programme trees is generated according to certain parameters that can vary depending on implementation. Each member of the population is executed in turn. The results of these executions are evaluated and each member of the population is given a fitness score so that better programmes have higher scores.

Members of the population are then selected to produce offspring programs based on their fitness. Simply selecting the best individuals to reproduce can lead to low diversity in the resulting population, so a probabilistic is often used to prevent a single programme's offspring from dominating subsequent generations. Common methods for doing this include fitness proportionate selection, where the probability of selection is proportional to the fitness of the individual relative to the fitness of the whole population, and tournament selection, where a small subset of programmes are chosen with equal probability and the fittest of the subset is selected for reproduction.

Genetic operators are then applied to the chosen programmes to create a new generation. The operator selected to create each member is chosen with a pre selected probability. The three most commonly used operators, which are also the three operators Koza used to successfully evolve a sort [3], are:

- **Reproduction:**

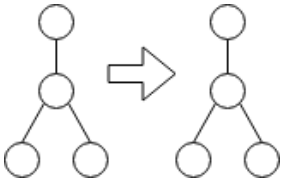

Figure 2: Example of reproduction operator

The selected programme is copied into the new generation without modi-

fication. Reproduction tends to have a low selection probability to ensure the following generation si sufficiently different.

- **Crossover:**

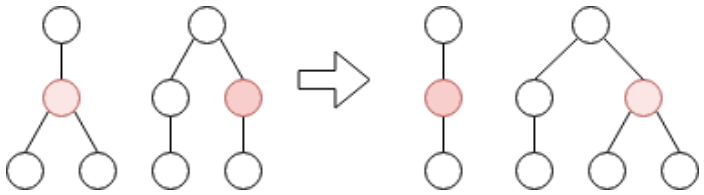

Figure 3: Example of crossover operation. The highlighted nodes are the crossover points

A random node is selected as the crossover point in each of the two chosen programmes and the subtrees rooted at the selected nodes are swapped.

- **Mutation:**

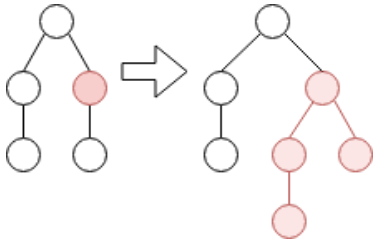

Figure 4: Example of mutation operator. The highlighted node is replaced by the highlighted subtree.

A random node is selected in the chosen programme. A new, random subtree is generated to replace the subtree rooted at the selected node.

The process of creating new generation is continued until a certain condition is reached, usually after a certain number of generations or a programme exceeds a fitness specified by the user.

2.2 Single Node Genetic Programming

Single Node Genetic Programming is a variant of GP that uses a form of dynamic programming to speed up the evaluation of GP programmes.

The population is organised a single directed acyclic graph. This allows the nodes to be organised into topological ordering such that any given node only uses nodes later in the ordering as operands. The nodes are stored in an array in this order, with all the terminals occupying the lowest values, as shown in 2.2

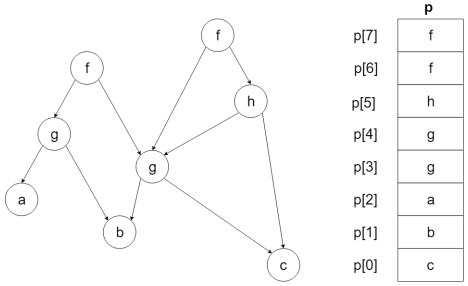

Figure 5: This SNGP graph is stored in topological order in an array. Terminals a,b, and c occupy the lowest positions of the array.

2.3 Prior Work on Evolving a Sort

3 Data Required

4 Design

4.1 GP Implementation

The GP implementation was based example code given by Dr Jackson and also on an implementation of the TinyGP system found in the book *A Field Guide to Genetic Programming* [5]. It was written in C as the example code given to me was also in C. The exact parameters are the same as described in Koza's work [4].

Each programme is stored as a string of primitive functions and terminals. The string is in prefix notation, and as all primitives have fixed arity no bracketing is needed. Executing a programme is done by a recursive interpreter

The each member of the population is initialised by using the grow method. A maximum tree depth is specified and a random primitive is selected as the root. Random primitives are selected as the children for primitives that have been previously selected. The primitives can be either functions or terminals, unless selecting a function would cause the tree to exceed the maximum depth. The initial maximum depth is chosen to be 6 to match Koza's work. The pseudocode for the algorithm is shown in

---

Input: maxTreeDepth

Three genetic operators are used: crossover, reproduction, and mutation.

Algorithm 1: Genetic Programming Algorithm

```
initialisePopulation()
evaluatePopulation(0)
for generation ← 1 to NUM_GENERATIONS do
    for j ← to POPULATION_SIZE do
        if randInt(1, 10) <= 8 then
            newPopulation[j] ← crossover(selectProg(), selectProg())
        else
            newPopulation[j] ← reproduce(selectProg())
        end
    end
    foreach programme ∈ newPopulation do
        if randInt(1, 10) = 1 then
            mutate(programme)
        end
    end
    population ← newPopulation
    evaluatePopulation(generation)
end
Output: best(population)
```

4.2 SNGP Implementation

4.3 Fitness Functions

While many different fitness functions were used, they all counted the number of inversions in a test array before and after executing a programme. To efficiently count inversions I used a modified merge sort algorithms, detailed in algorithm 2. The algorithm is $O(n \log n)$, and merging is done using two preallocated buffers to avoid unnecessary memory allocations while the GP algorithm is running.

4.4 Test Data

Each programme generated during a GP run is executed on a set of test arrays. This data comes from pre-generated file containing a series of random arrays loaded into the programme at the beginning of execution. This has the advantage of allowing for reproduction of results, easier testing of the programme, and prevents random number generation from increasing the execution time.

The arrays are organised into fixed size groupings. Each programme in the same GP generation is executed on the same group of arrays. This was done so that each programme was subjected to the same conditions in each generation, while still preventing over fitting to a fixed set of data. This method also allows each group to be customized to meet certain requirements (e.g. contains a sorted array, a reversed array, or a very short array), but this was found to be unnecessary.

Algorithm 2: Algorithm that sorts array and counts number of inversions

```
Function countInversions (arr):
    if length(arr) <= 1 then
        inversions ← 0
    else
        sizeA ← ⌊length(arr)/2⌋
        sizeB ← length(arr) - sizeA
        A ← [First sizeA elements of arr]
        B ← [Last sizeB elements of arr]
        inversions ← countInversions(A) + countInversions(B)
        C ← Empty Array
        while length(A) > 0 and length(B) > 0 do
            if B[1] < A[1] then
                Append B[1] to C
                inversions ← inversions + length(A)
                Remove first element of B
            else
                Append A[1] to C
                Remove first element of A
            end
        end
        if length(A) = 0 then
            Append remaining elements of B to C
        else if length(B) = 0 then
            Append remaining elements of A to C
        end
        arr ← C
    end
    Output: inversions
```

The file is a standard text file where the first two values on each line are the length and number of inversions of the array, followed by the array itself. Each value is separated by a space. A blank line indicates the end of the current group of arrays.

5 Realisation

6 Results

7 Evaluation

8 Learning Points

9 Professional Issues

10 Bibliography

References

[1] Jackson, D. A new, node-focused model for genetic programming. In *Genetic Programming* (2012), A. Moraglio, S. Silva, K. Krawiec, P. Machado, and C. Cotta, Eds., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 49–60.

[2] Jackson, D. Single node genetic programming on problems with side effects. In *Parallel Problem Solving from Nature - PPSN XII* (2012), C. A. C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, Eds., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 327–336.

[3] Kinnear, K. E. Evolving a sort: Lessons in genetic programming. In *in Proceedings of the 1993 International Conference on Neural Networks* (1993), IEEE Press, pp. 881–888.

[4] Kinnear, K. E. Generality and difficulty in genetic programming: Evolving a sort. In *ICGA* (1993).

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

11 Appendices