Evolving a Sorting Algorithm with SNGP

Robin Lockyer

University of Liverpool

April 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

In standard GP, programmes are encoded as a tree of primitive functions and terminals.

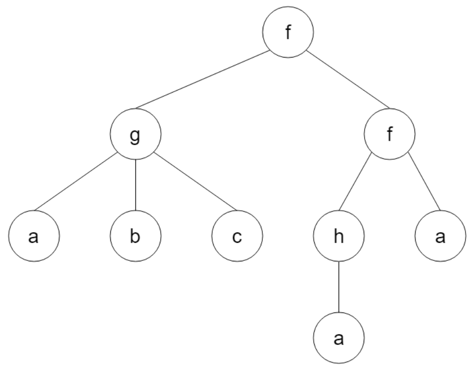

Figure 1: This tree encodes the programme f(g(a, b, c), f(h(a), a)), where f, g, and h are functions and a, b, and c are terminals.

An initial population of random programmes is created

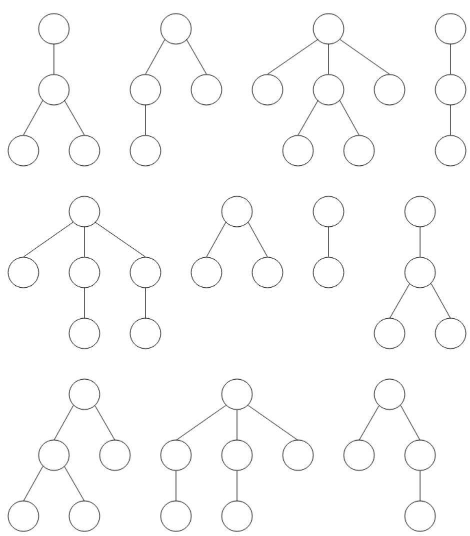

Figure 2: An example GP population

Each member of the population is executed, evaluated, and given a fitness score

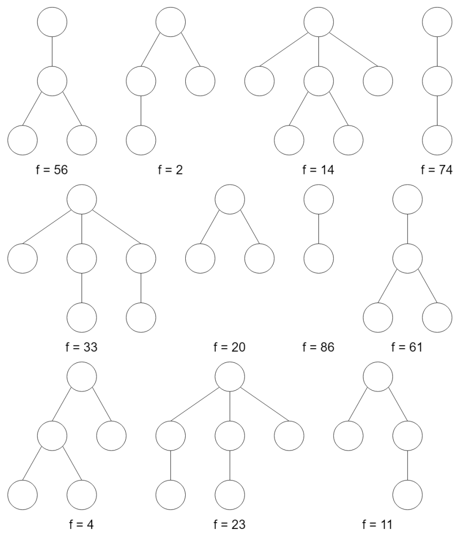

Figure 3: An example GP population with fitness scores

A new population is created by selecting some of the most fit members of the initial population and performing genetic operations on them to create new programmes.

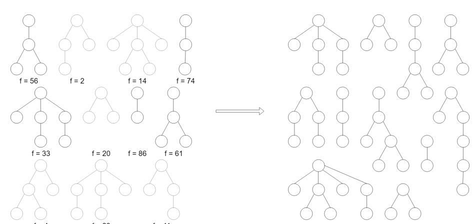

Figure 4: An example GP population with fitness scores

2.2 Single Node Genetic Programming

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())
    foreach programme ∈ newPopulation do
        if randInt(1, 10) = 1 then
            mutate(programme)
    population ← newPopulation
    evaluatePopulation(generation)
Output: best(population)
```

4.2 SNGP Implementation

4.3 Fitness Function

While many different fitness functions were used, they all counted the number of inversions of a test array before and after execution of a programme.

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

```
Function countInversions arr:
    if length(arr) <= 1 then
        Output: 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) > 0andlength(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
        if length(A) = 0 then
            Append remaining elements of B to C
        if length(B) = 0 then
            Append remaining elements of A to C
        arr ← C
        Output: inversions
```

4.4 Test Data

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