COMP390: Evolving a Sorting Algorithm with SNGP

Robin Lockyer

University of Liverpool

03/2019

Project Description

Aims:

- Replicate K. E. Kinnear's work [3] in evolving a sorting algorithm using Genetic Programming

Project Description

Aims:

- Replicate K. E. Kinnear's work [3] in evolving a sorting algorithm using Genetic Programming
- Re-implement Kinnear's work using Single Node Genetic Programming, a variant of GP invented by Dr David Jackson[1]

Project Description

Aims:

- Replicate K. E. Kinnear's work [3] in evolving a sorting algorithm using Genetic Programming
- Re-implement Kinnear's work using Single Node Genetic Programming, a variant of GP invented by Dr David Jackson[1]
- Compare the effectiveness of the two approaches to evolving a sorting algorithm

What Was Achieved?

Successfully replicated Kinnear’s work

What Was Achieved?

Successfully replicated Kinnear's work

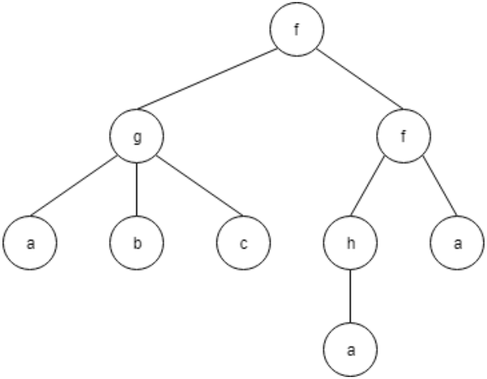
Unable to evolve a sort using SNGP

What is genetic programming?

Genetic programming is applying genetic algorithms to programmes in order to generate a programme that performs well in a given problem domain.

How Does GP Work? - 1

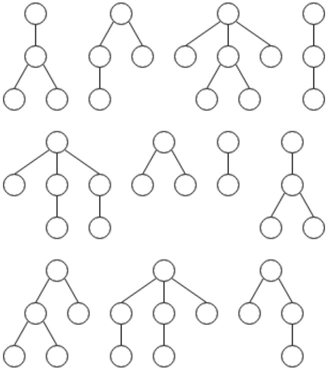
Programmes are encoded as a tree of primitive functions and terminals



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.

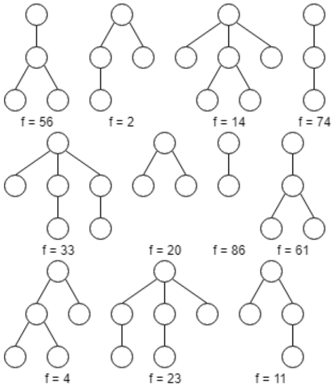
How does GP Work? - 2

An initial population of random programmes is created



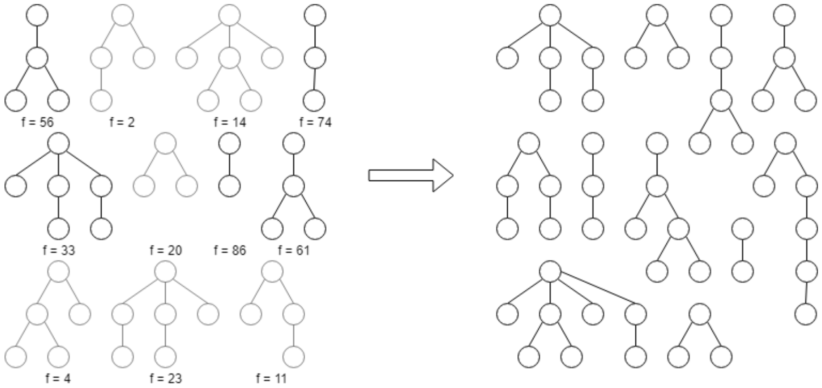
How Does GP Work? - 2

An initial population of random programmes is created



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

How Does GP Work? - 3



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

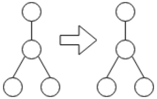
How Does GP Work? - 4

There are three main genetic operators:

How Does GP Work? - 4

There are three main genetic operators:

Reproduction

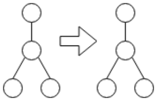


A programme is copied over to the new population without any changes

How Does GP Work? - 4

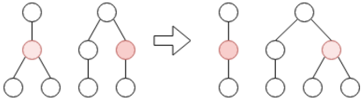
There are three main genetic operators:

Reproduction



A programme is copied over to the new population without any changes

Crossover

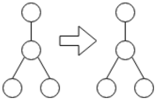


A random node is selected in each of the chosen programmes. The subtrees rooted at the selected nodes are swapped.

How Does GP Work? - 4

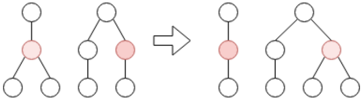
There are three main genetic operators:

Reproduction



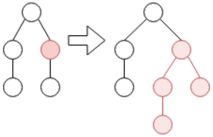
A programme is copied over to the new population without any changes

Crossover



A random node is selected in each of the chosen programmes. The subtrees rooted at the selected nodes are swapped.

Mutation



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

How Does GP Work? - 5

This process is repeated until a programme with high enough fitness is generated

What is SNGP?

What is SNGP?

- SNGP is a variation of GP that organises the whole population into a single interlinked graph

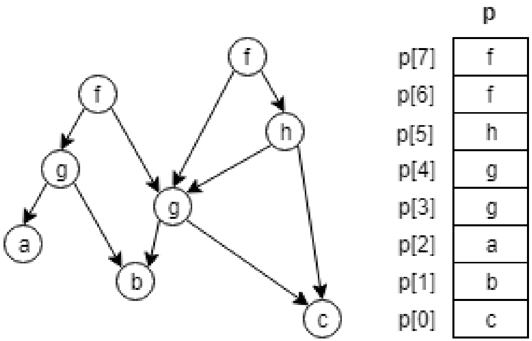
What is SNGP?

- SNGP is a variation of GP that organises the whole population into a single interlinked graph
- The subtree rooted at each node in the graph is considered to be an individual programme

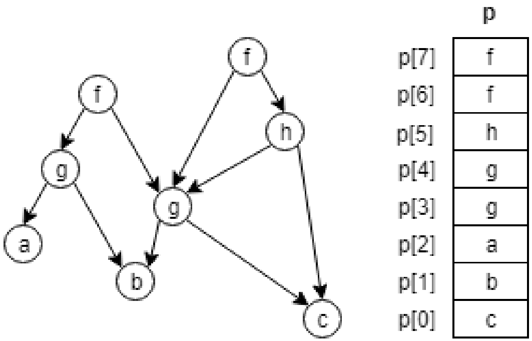
What is SNGP?

- SNGP is a variation of GP that organises the whole population into a single interlinked graph
- The subtree rooted at each node in the graph is considered to be an individual programme
- The graph is structured in such a way that a form of dynamic programming can be used to increase the efficiency of evaluating the population

SNGP Population - 1

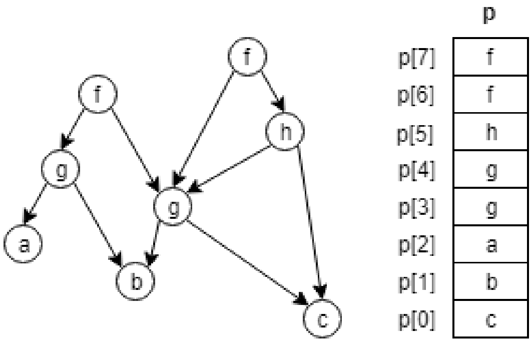


SNGP Population - 1



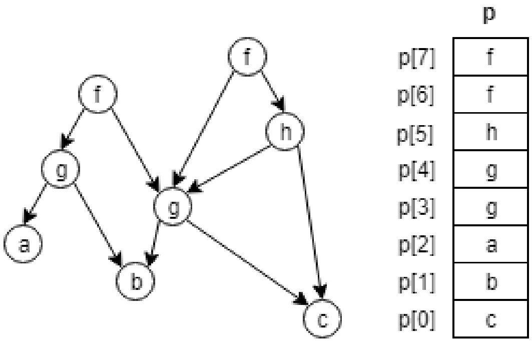
- Graph nodes are stored in an array

SNGP Population - 1



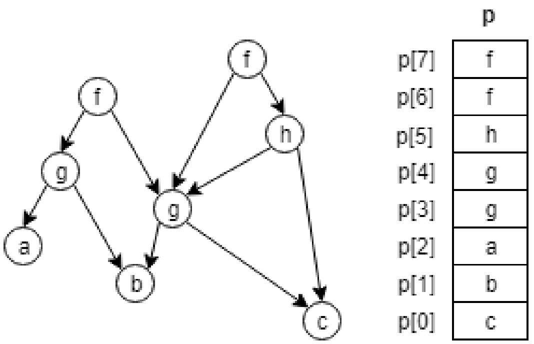
- Graph nodes are stored in an array
- Terminals are stored in the lowest elements

SNGP Population - 1



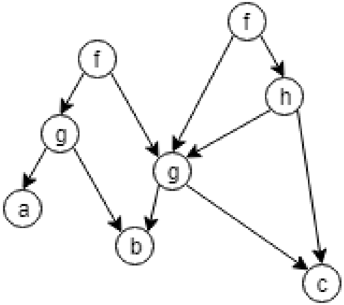
- Graph nodes are stored in an array
- Terminals are stored in the lowest elements
- Remaining elements store a random function

SNGP Population - 1



- Graph nodes are stored in an array
- Terminals are stored in the lowest elements
- Remaining elements store a random function
- Each function's operands are chosen from elements with a smaller index

SNGP Population - 2



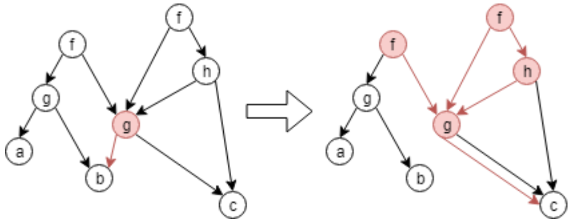
This graph contains the following programmes:

- a
- b
- c
- g(b, c)
- g(a, b)
- h(g(b, c), c)
- f(g(a, b), g(b, c))
- f(g(b, c), h(g(b, c), c))

SNGP Operators

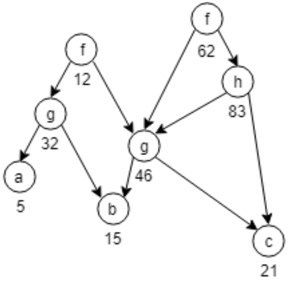
SNGP has only one genetic operator:

Successor Mutate



A random node is chosen and one of its operands is randomly changed. This causes the programmes represented by both the chosen node and all of its predecessors to be altered.

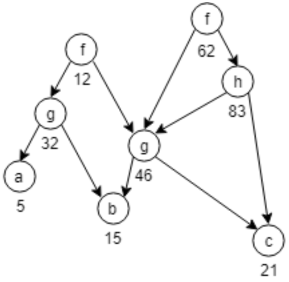
Calculating Fitness - 1



Each individual is executed, evaluated and given a fitness,and then the whole population is given a fitness as a whole.

If a change to the population does not improve this overall fitness, the change is reverted and a different node is selected for the genetic operator.

Calculating Fitness - 2



There are two methods of giving the population a fitness:

- SNGP/A: The fitness of the population is the average of each individual fitness. In the diagram above this fitness is 34.5.
- SNGP/B: The fitness of the population is the fitness of best individual. In the diagram above this fitness is 83.

Benefits

- When using pure functions the graph structure allows for dynamic programming by re-using the results from executing subtrees, making evaluation more efficient.
- A single application of successor mutate can modify many individuals at once
- When re-evaluating after successor mutate, results from the previous generation can be used to speed up the evaluation of the graph.
- SNGP tends to find solutions more often than SNGP [1, 2].
- SNGP allows for a single programme to re-use code, resulting in smaller programmes [1].

Benefits

- When using pure functions the graph structure allows for dynamic programming by re-using the results from executing subtrees, making evaluation more efficient.
- A single application of successor mutate can modify many individuals at once
- When re-evaluating after successor mutate, results from the previous generation can be used to speed up the evaluation of the graph.
- SNGP tends to find solutions more often than SNGP [1, 2].
- SNGP allows for a single programme to re-use code, resulting in smaller programmes [1].

However:

When using functions with side effects SNGP cannot make use of dynamic programming as the result of executing subtrees changes depending on what was executed before.

Reproducing Kinnear's Results

I was successful in reproducing Kinnear's results.

Reproducing Kinnear's Results

I was successful in reproducing Kinnear's results.

Following the method set out in [4] and [3] immediately lead to evolving a sort.

Reproducing Kinnear's Results

I was successful in reproducing Kinnear's results.

Following the method set out in [4] and [3] immediately lead to evolving a sort.

With a population of 1000 Kinnear's method would consistently find a solution within 50 generations.

Implementation Details

Implementation based on example code provided by Dr Jackson from his past experiments with GP, and also on an implementation of the TinyGP system found in the book "A Field Guide to Genetic Programming" [5].

Some changes from Dr Jackson's code has to be made to follow Kinnear's method.

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH
- ITERATE(start, end, function)

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH
- ITERATE(start, end, function)
- SWAP(x, y)

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH
- ITERATE(start, end, function)
- SWAP(x, y)
- SMALLEST(x, y)

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH
- ITERATE(start, end, function)
- SWAP(x, y)
- SMALLEST(x, y)
- LARGEST(x, y)

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH
- ITERATE(start, end, function)
- SWAP(x, y)
- SMALLEST(x, y)
- LARGEST(x, y)
- SUBTRACT(x, y)

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH
- ITERATE(start, end, function)
- SWAP(x, y)
- SMALLEST(x, y)
- LARGEST(x, y)
- SUBTRACT(x, y)
- INCREMENT(x)

Implementing Primitives

The primitives used were the same nine described by Kinnear in [3], although I renamed some to make their function more clear.

- INDEX
- LENGTH
- ITERATE(start, end, function)
- SWAP(x, y)
- SMALLEST(x, y)
- LARGEST(x, y)
- SUBTRACT(x, y)
- INCREMENT(x)
- DECREMENT(x)

Fitness Function

Kinnear's fitness function as described in [4] is as follows:

$$fitness(prog) = \frac{adjusted(prog)}{\sum_{p \in population} adjusted(p)}$$

$$adjusted(prog) = \frac{1}{1 + raw(prog)}$$

$$raw(prog) = praw(prog) - \min_{p \in population} praw(p)$$

$$praw(prog) = \left(\sum_{t=1}^{NumberOfTests} res(t) \right) \cdot of + size(prog) \cdot sf$$

$$res(t) = rdis(t) + pdis(t)$$

$$pdis(t) = max(rdis(t) - idis(t), 0) \cdot 100$$

Where rdis(t) is the remaining number of inversions in a test array t after running a member of the population, idis(t) is the initial number of inversions in a test array t, and of and sf are constant weights used to adjust the resulting fitness.

Bibliography I

[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.

Bibliography II

[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.