

# COMP390: Evolving a Sorting Algorithm with SNGP

Robin Lockyer

University of Liverpool

03/2019

Introduction

Overview of Genetic Programming

Overview of Single Node Genetic Programming

Reproducing Kinnear's Results

Attempting SNGP

Bibliography

# 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:

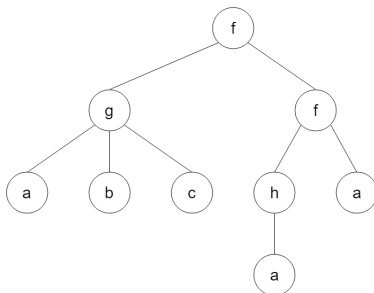
- ▶ Replicate K. E. Kinnear's work [3] in evolving a sorting algorithm using Genetic Programming
- ▶ Re-implement Kinnear's work using Single Node Genetic Programming, a variant of GP invented by Dr David Jackson[1]
- ▶ Compare the effectiveness of the two approaches to evolving a sorting algorithm

# What 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.

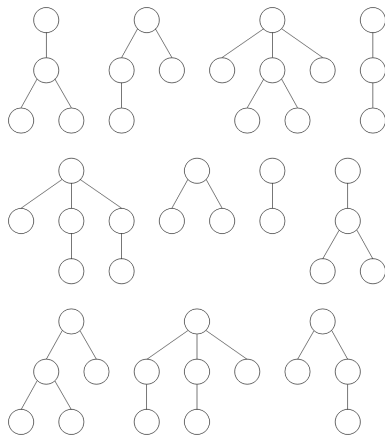
# How Does GP Work? - 1

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.

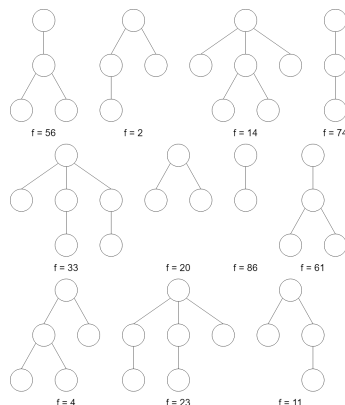
## How Does GP Work? - 2



An initial population of random programmes is created

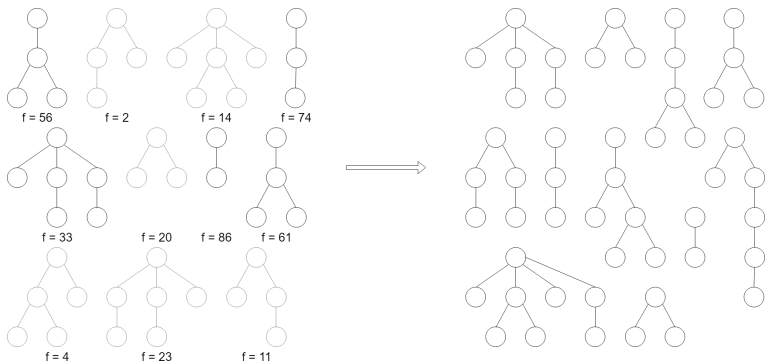


# How Does GP Work? - 3



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

## How Does GP Work? - 4



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

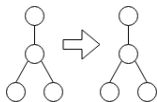
## How Does GP Work? - 5

There are three main genetic operators:

# How Does GP Work? - 5

There are three main genetic operators:

## Reproduction

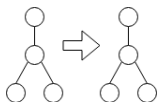


A programme  
is copied over  
to the new  
population  
without any  
changes

# How Does GP Work? - 5

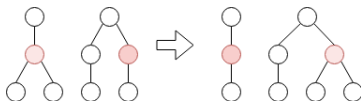
There are three main genetic operators:

## Reproduction



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

## Crossover

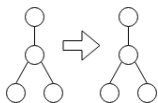


A random node is selected in each of the chosen programmes. The subtrees rooted at the selected nodes are swapped.

# How Does GP Work? - 5

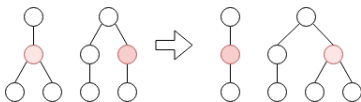
There are three main genetic operators:

## Reproduction



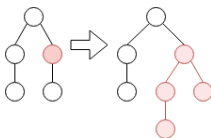
A programme is copied over to the new population without any changes

## 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? - 6

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

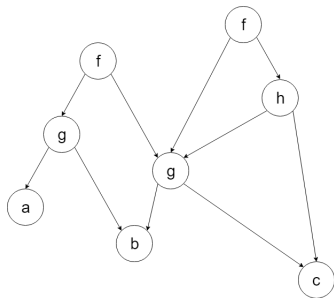
## What is SNGP?

- ▶ 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?

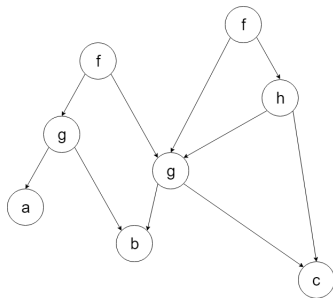
- ▶ SNGP is a variation of GP that organises the whole population into a single interlinked graph
- ▶ The subtree rooted at each node in the graph is considered to be an individual programme
- ▶ The graph is structured in such a way that a form of dynamic programming can be used to increase the efficiency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

# SNGP Population - 1



	<b>p</b>
p[7]	f
p[6]	f
p[5]	h
p[4]	g
p[3]	g
p[2]	a
p[1]	b
p[0]	c

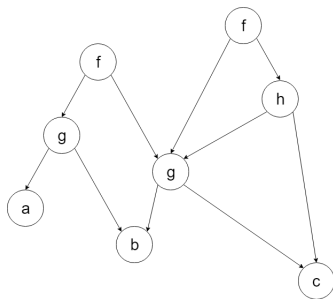
# SNGP Population - 1



	<b>p</b>
p[7]	f
p[6]	f
p[5]	h
p[4]	g
p[3]	g
p[2]	a
p[1]	b
p[0]	c

- Graph nodes are stored in an array

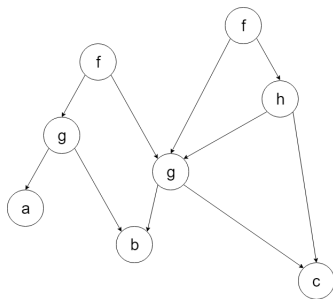
# SNGP Population - 1



	<b>p</b>
p[7]	f
p[6]	f
p[5]	h
p[4]	g
p[3]	g
p[2]	a
p[1]	b
p[0]	c

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

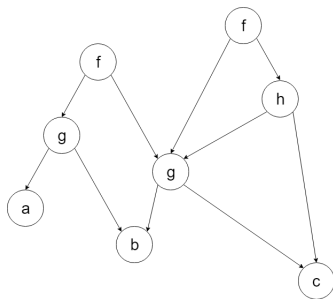
# SNGP Population - 1



	p
p[7]	f
p[6]	f
p[5]	h
p[4]	g
p[3]	g
p[2]	a
p[1]	b
p[0]	c

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

# SNGP Population - 1



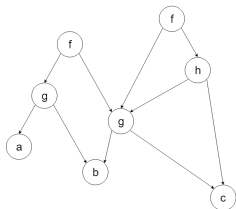
	<b>p</b>
p[7]	f
p[6]	f
p[5]	h
p[4]	g
p[3]	g
p[2]	a
p[1]	b
p[0]	c

- ▶ 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:



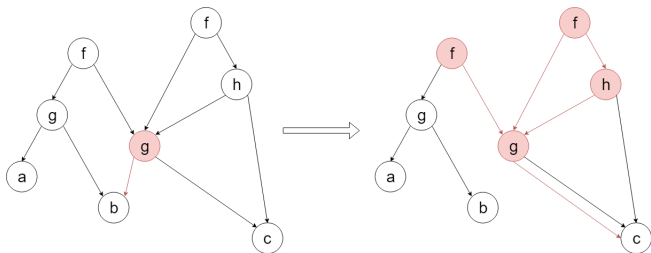
	p
p[7]	f
p[6]	f
p[5]	h
p[4]	g
p[3]	g
p[2]	a
p[1]	b
p[0]	c

- ▶ 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

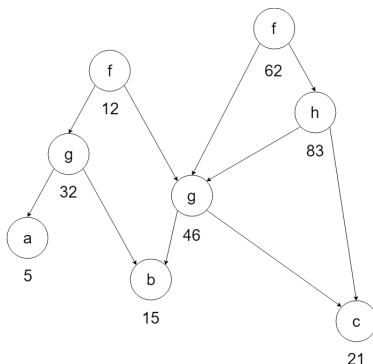
SNGP has only one genetic operator:

## 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.

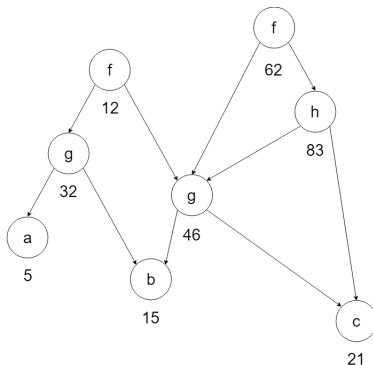
# Calculating Fitness - 1



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.

- ▶ 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.

# Modifications to Dr Jackson's Implementation

Other important modifications to Dr Jackson's Implementation and TinyGP include:

# Modifications to Dr Jackson's Implementation

Other important modifications to Dr Jackson's Implementation and TinyGP include:

- ▶ Used batch GP instead of steady state GP

# Modifications to Dr Jackson's Implementation

Other important modifications to Dr Jackson's Implementation and TinyGP include:

- ▶ Used batch GP instead of steady state GP
- ▶ Used fitness proportional selection over tournament selection

# Modifications to Dr Jackson's Implementation

Other important modifications to Dr Jackson's Implementation and TinyGP include:

- ▶ Used batch GP instead of steady state GP
- ▶ Used fitness proportional selection over tournament selection
- ▶ Modified the *createTree()* function to take an additional integer argument that specified the maximum number of nodes in the generated tree.



# Modifications to Dr Jackson's Implementation

Other important modifications to Dr Jackson's Implementation and TinyGP include:

- ▶ Used batch GP instead of steady state GP
- ▶ Used fitness proportional selection over tournament selection
- ▶ Modified the *createTree()* function to take an additional integer argument that specified the maximum number of nodes in the generated tree.
- ▶ Completed implementation of the mutate operator

# Modifications to Dr Jackson's Implementation

Other important modifications to Dr Jackson's Implementation and TinyGP include:

- ▶ Used batch GP instead of steady state GP
- ▶ Used fitness proportional selection over tournament selection
- ▶ Modified the *createTree()* function to take an additional integer argument that specified the maximum number of nodes in the generated tree.
- ▶ Completed implementation of the mutate operator
- ▶ Added ability to load files containing test data instead of randomly generating arrays at runtime

# Attempting SNGP

My attempt to re-implement Kinnear's work using SNGP was not successful. I was unable to create an implementation that was able to consistently evolve a sorting algorithm.

I believe the main problem to be the fitness of the population reaching a local maxima at which point the successor mutate operation was unable to find a change that would improve the fitness, resulting in stagnation for the remaining generations.

This behaviour occurred no matter the parameters or variations of fitness and primitives.

It is possible the lack of success is due errors in programming, however I believe this to be unlikely for the following reasons:

- ▶ The SNGP implementation reused code from my successful GP implementation

It is possible the lack of success is due errors in programming, however I believe this to be unlikely for the following reasons:

- ▶ The SNGP implementation reused code from my successful GP implementation
- ▶ When the initial population was seeded with a fully working programme it was given the correct fitness score and was correctly identified as a working solution

It is possible the lack of success is due errors in programming, however I believe this to be unlikely for the following reasons:

- ▶ The SNGP implementation reused code from my successful GP implementation
- ▶ When the initial population was seeded with a fully working programme it was given the correct fitness score and was correctly identified as a working solution
- ▶ On each run the overall fitness of the population did increase overtime, until it hit a maxima

It is possible the lack of success is due errors in programming, however I believe this to be unlikely for the following reasons:

- ▶ The SNGP implementation reused code from my successful GP implementation
- ▶ When the initial population was seeded with a fully working programme it was given the correct fitness score and was correctly identified as a working solution
- ▶ On each run the overall fitness of the population did increase overtime, until it hit a maxima
- ▶ When the initial population was seeded with a partially working solution SNGP was occasionally able to use it to create a fully working solution

It is possible the lack of success is due errors in programming, however I believe this to be unlikely for the following reasons:

- ▶ The SNGP implementation reused code from my successful GP implementation
- ▶ When the initial population was seeded with a fully working programme it was given the correct fitness score and was correctly identified as a working solution
- ▶ On each run the overall fitness of the population did increase overtime, until it hit a maxima
- ▶ When the initial population was seeded with a partially working solution SNGP was occasionally able to use it to create a fully working solution
- ▶ On very rare occasions SNGP was able to evolve a sort, typically when the initial population was unusually favourable



# Implementation

This implementation was solely based on example code provided by Dr Jackson.

Added an additional function to calculate size of programmes

The set of primitives were the same as used in the GP implementation.

Used same code as in GP implementation for primitive execution, counting inversions, and loading test data.

# Experiments

Some of the parameters I experimented with include:

- ▶ SNGP/A vs SNGP/B

# Experiments

Some of the parameters I experimented with include:

- ▶ SNGP/A vs SNGP/B
- ▶ Small population over large generations and vice versa

# Experiments

Some of the parameters I experimented with include:

- ▶ SNGP/A vs SNGP/B
- ▶ Small population over large generations and vice versa
- ▶ Experimented with variations of Kinnear's fitness function

# Experiments

Some of the parameters I experimented with include:

- ▶ SNGP/A vs SNGP/B
- ▶ Small population over large generations and vice versa
- ▶ Experimented with variations of Kinnear's fitness function
- ▶ Tried own fitness functions, e.g:

$$fitness(x) = \begin{cases} 1 & \text{if } rdis(t) = 0 \\ -rdis(t) & \text{if } rdis(t) > idis(t) \wedge rdis(t) \neq 0 \\ 0 & \text{if } rdis(t) = idis(t) \wedge rdis(t) \neq 0 \\ 1 - rdis(t)/idis(t) & \text{otherwise} \end{cases}$$

# Experiments

Some of the parameters I experimented with include:

- ▶ SNGP/A vs SNGP/B
- ▶ Small population over large generations and vice versa
- ▶ Experimented with variations of Kinnear's fitness function
- ▶ Tried own fitness functions, e.g:

$$fitness(x) = \begin{cases} 1 & \text{if } rdis(t) = 0 \\ -rdis(t) & \text{if } rdis(t) > idis(t) \wedge rdis(t) \neq 0 \\ 0 & \text{if } rdis(t) = idis(t) \wedge rdis(t) \neq 0 \\ 1 - rdis(t)/idis(t) & \text{otherwise} \end{cases}$$

- ▶ Altered function set to encourage beneficial behaviour

What Could Have Been Done Differently?

# 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.