caRtesian: An Implementation of Cartesian Genetic Programming in R

Final Year Dissertation

Ryan Porteous
BSc (Hons) Computer Science (Artificial Intelligence)

Supervisor: Dr Michael Lones

Second Reader: Dr Katrin Lohan

Declaration of own work

I, Ryan Porteous confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Ryan Porteous

Date: 23/04/18

Abstract

R has become the first-choice language for data scientists. However, it is typically not the first choice for people developing and implementing bio-inspired algorithms. Consequently, it can be hard for data scientists to make use of bio-inspired methods. This project looks at the current availability of bio-inspired algorithms in R and identifies holes in the provision. A discussion of how a package was designed, developed, and ultimately released to fill in one of these holes.

The package is known as “caRtesian” and it implements Cartesian Genetic Programming in R. The package is assessed on its usability since the package is freely available to the public and assessed on its performance. These are also discussed with issues accounted for and solutions suggested.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr Michael Lones for providing his time, feedback and continuous guidance throughout the project. Also, for proposing the topic and supporting my design choices as the project progressed. Finally, for introducing me to the exciting area of Genetic Programming.

I would also like to thank my second reader, Dr Katrin Lohan, also for her feedback during the project.

Finally, I would like to thank my friends and family for providing support through my studies and always encouraging me to push myself further.

Contents

1 Introduction

1.1 Aims and Motivations

Bio-inspired computing is a field which takes inspiration for its algorithms from a variety of nature's systems such as evolution, and the way populations of animals interact with each other in an environment. This collection of algorithms can be applied to various areas and can also be used as a searching method to solve NP-hard problems due to the way the search space can be explored. R is a programming language which has become more popular in recent years as can be seen in the TIOBE Index (TIOBE, 2017). It is among the first-choice of languages for statisticians and data miners, with competition being mainly from Python, but is not the first-choice for those who are interested in implementing bio-inspired algorithms. Due to this, it can be hard for R users to apply bio-inspired methods to problem due to their limited availability.

The primary aims of this project can be defined as:

1. **Investigate the availability of bio-inspired algorithms in R**

   I will begin by identifying the main areas of bio-inspired computing and searching for implementations of these algorithms available through the Comprehensive R Archive Network (CRAN). CRAN is a network which provides a central platform for R users to upload the software packages that they have developed and provide access to them for other users.

2. **Identify implementations to be improved or built upon, and areas where no solution exists**

   From the implementations found in the previous step, I will assess how the solution has been implemented, what the solution provides and if it can be

improved upon. Where no solution exists, this will be identified as an area that can be developed.

3. **To learn the fundamentals of R**

   As mentioned previously R is among the first-choice of languages for data mining which is a field I am interested in. Thus, I aim to learn the fundamentals of the R language and gain practical experience of using them.

4. **Produce an R Package to improve the availability of bio-inspired tools for R**

   A package will be developed to improve upon an existing solution or to provide a solution where no solution exists. I will aim to follow the best practices when creating this package to maintain a high standard of code quality and maintainability.

5. **Produce an implementation of Cartesian Genetic Programming**

   This is the area I have identified as needing improvement and have chosen to produce a package to provide access to this area.

6. **Release the package on CRAN**

   I aim to release the package on CRAN so that the package will be freely available for other R users to make use of.

7. **Evaluate the usability of the package**

   It is essential that the package is easy for users to use with no previous knowledge of Genetic Programming. To provide this, a guide will be distributed with the package and a Usability Study will be carried out on this guide.

8. **Evaluate the functionality of the package and identify areas for improvement**

   Evaluation is an important stage of software development and I plan to incorporate it into this project. I will assess what the created package provides

and what could be added to it. I will also assess the performance of the package and suggest possible improvements.

1.2 Overview of Document

In the document that follows, I will cover the research carried out in order to identify the area of bio-inspired computing that could use improvement. I will then show how I was able to develop and release a package to provide provision to this area, making sure to emphasise what areas of the package work well, and what areas could use improvement. The usability study carried out will then be discussed which highlighted that there are issues with the software guide that I have produced but still allows users with no background in the area, or with R to use it which is an important factor when releasing software. The correctness of the package and its performance will also be discussed. Then a brief section discussing the experiments carried out which highlighted that the set of functions distributed with the package may not be ideal. Finally, a self-reflective section will be covered where I will compare what my initial goals were to what I have achieved, and how I think my skills have developed through the project.

2 Bio-inspired Computing

2.1 Genetic Algorithms

Genetic Algorithms (GAs) are an evolutionary search heuristic which takes inspiration from the process of natural selection (Darwin & Wallace, 1998). The algorithm uses a population of solutions to the given problem where each solution is given a fitness value which defines how suitable this solution is in this domain. The fitness value which can either be maximised or minimised is given from a fitness function which is defined depending on the scenario. This fitness value is used in the selection process which mimics natural selection (Darwin & Wallace, 1998). Each solution has a probability relative to their fitness value of being chosen as a parent. Parent is a term used to refer to a solution from the current generation which will be used in the crossover process to produce a child. A child is a solution that will carry over to the next generation. Crossover is a process, or operator, where two parents are used to generate a child solution. The goal is to combine both parents while removing the negative characteristics of the parents so that the child will have an improved fitness. Another operator which is used is mutation. This randomly alters the child solution and can help to explore the search space quickly (Sivanandam & Deepa, 2008). One implementation of this according to (Moon et al. 2002) is to choose two random values in the solution and to swap them. This process is repeated until a pre-defined number of generations have completed, or a set number of generations have passed with no improvement.

Genetic Algorithms are used for solving optimisation problems which are problems that involve finding the optimal solution in a search space of all possible solution. It can be difficult to find the globally optimal solution due to the search landscape itself having many local maxima, noise or from other constraints according to (Kramer, 2017). The performance of GA's are reduced significantly in problems which have very high

dimensions and where the evaluation of the fitness function becomes very computationally intensive (Kar, 2016).

2.2 Cellular Automata

Although Cellular Automata (CA) were originally outlined by von Neumann and Stanislaw Ulam with the motivation of modelling biological self-reproduction (Wolfram, 1983) they did not gain widespread interest until John Conway's "Game of Life" was revealed in 1970 (Adamatzky, 2010). CA are mathematical models consisting of simple components with local interactions (Navid & Bagheri, 2013) which are made up a lattice consisting of cells. The lattice can be defined as an n-dimensional list of cells where the cells have two states, black or white. To evolve or update the lattice of cells we use discrete time where time 0 is the initial state of the lattice. In each generation, a set of rules is applied to each cell. In a one-dimensional list, the colour of a given cell at each step is dictated by the rules which consider the colour of the cell and it's left and right neighbouring cells on the previous step (Wolfram, 2002). Thus, a simple rule may be defined as if the given cell and all surrounding cells were black in the previous step, then turn the given cell white. In a one-dimensional lattice, the lines of cells can be layered to provide a visual representation of their behaviour over time which is an important characteristic of CA. Of course, there is no reason why this definition cannot be expanded into using more than 2 states for each cell, or defining a cells neighbourhood as all surrounding cells such as in the Moore neighbourhood which applies to two-dimensional automata and is defined as the 8 cells surround a given cell (Adamatzky, 2010). Another common neighbourhood is the von Neumann neighbourhood (Weisstein, 2003) which uses the cells directly above, below, to the left, and to the right of a given cell. The boundaries of the lattice need conditions to handle the problem where a cell's neighbourhood is out of bounds. A common way of handling this problem is to wrap the lattice at the edges.

Cellular automata can be used for the modelling of different processes. One such process is the spread of forest fires (Ghisu et al. 2015). Another is using them to generate random numbers that can be used in encryption (Sarkar, 2000).

2.3 Artificial Neural Networks

Inspired by biological neural networks, Artificial Neural Networks (ANN) are one of the most widely used bio-inspired techniques. McCulloch and Pitts (1943) are credited with the writing of the article which marked the beginning of Neurocomputing (Yadav et al. 2015). In the article they created a computational model for neural networks and showed that any arithmetic or logical function could be computed by a simple neural network. According to (Yadav et al. 2015) an artificial neural network is an information processing system that has performance characteristics also present in biological neural networks. Russell and Norvig (2009) formally define them as collections of nodes, or neurons, connected by directed links. Each link has a continuous weight value which governs the strength and sign of the link. Each node computes the weighted sum of its inputs and then applies an activation function to produce an output value. The activation function works as a threshold which allows a network to represent nonlinear functions. Russell and Norvig (2009) also explain how this node definition can be connected to form a network. There are two main options which are feed-forward networks and recurrent networks. A feed-forward network's connections form a directed acyclic graph as the nodes can only send information forward. Nodes in a recurrent network receive their output values as inputs which allows them to support short-term memory. Figures 2-1 and 2-2 show examples of a feed-forward and recurrent network respectively.

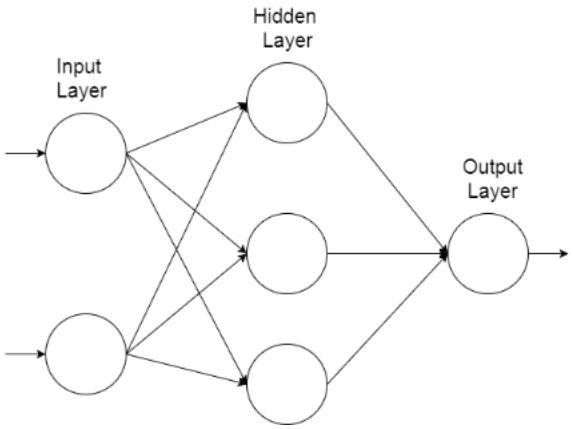


Figure 2-1: An example of a feed-forward neural network topology

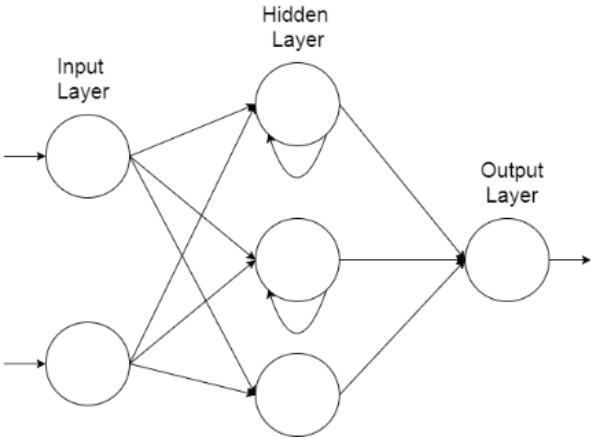


Figure 2-2: An example of a recurrent neural network topology

Due the amount of research applied to ANNs, there are many different topologies or arrangements and can be applied to a variety of problems. They are useful for identifying relationships between variables or recognising patterns within data (Zhang, 2009) and due to this are a common tool used in data mining where they have been applied to both supervised and unsupervised learning problems (Craven & Shavlik, 1997).

2.4 Swarm Intelligence

Swarm Intelligence is an area of algorithms which have gained a lot of popularity due to their versatility and their efficiency in solving nonlinear design problems (Yang & Karamanoglu, 2013). I will cover two of the main swarm intelligence algorithms, namely Ant Colony Optimisation (ACO) and Particle Swarm Optimisation.

2.4.1 Ant Colony Optimisation

This algorithm takes inspiration from real ant colonies. Ants use pheromone to mark paths leading to food to communicate its location indirectly to other ants (Khushaba et al. 2008). Khushaba et al. (2008) continue to explain the behaviour of ants foraging for food. The amount of pheromone deposited depends on the distance to the food source, and the quality and quantity of the food source. The paths that are shorter are visited more on average due to more pheromone existing on the path. After a period, the difference in the amount of pheromone between the path options is large enough so that future ants to come across the paths are likely to follow the path previously

marked and reinforce the option with their own pheromone (Dorigo & Gambardella, 1997). The quality of a solution to a problem can be modelled as the concentration of pheromone on a path according to (Yang & Karamanoglu, 2013). Due to the solution being modelled this way, the algorithm generally produces routes and paths evident from their higher concentrations, thus ant algorithms are well suited to discrete optimisation problems.

The ACO was originally used to solve the Travelling Salesman Problem and was effective at finding good solutions (Khushaba et al. 2008). ACO has been applied as a searching method within feature selection problems namely in areas of face and speech recognition problems (Khushaba et al. 2008). Feature selection is the process of reducing data with high dimensions into a lower dimension while keeping as close to the same amount of information as possible (Khan & Baig, 2015).

2.4.2 Particle Swarm Optimisation

The collective behaviour of different animal species is the basis for Particle Swarm Optimisation (PSO). Behaviours such as fish schooling, insect swarming, and birds flocking are examples of such behaviour (Saka et al. 2013). Many newer algorithms that are based on swarm intelligence have taken inspiration from different areas, but still share connections to components used within PSO, thus it can be said that PSO established the foundational ideas of swarm intelligence-based computation as Yang (2014) describes.

Particle Swarm Optimisation was developed by Eberhart and Kennedy (1995) and they state in this article that it can be used to solve many of the same problems as the previously described area, Genetic Algorithms, but does not suffer from some of the same difficulties. Since it can be used to solve the same types of problems, it is useful to compare the algorithms stating the differences. As mentioned previously, GA use operators known as mutation and crossover, but this is not present in PSO. Instead it uses random real-numbers and allows the particles to communicate with each other (Yang, 2014). Yang (2014) also continues to explain that PSO is easier to implement due

to no encoding or decoding of the solution being used. Eberhart and Kennedy (1995) describe the PSO concept as being like a GA in that a starting population of random candidate solutions is used but differ in that each solution is given a velocity value and is then “flown” through hyperspace. Solutions in PSO are referred to as particles. Each of these particles has memory, which is not a feature in a GA. This stores a value called the pbest which is the coordinates of the best solution found so far in the search space. The gbest is also stored by the particle swarm optimiser which is the best solution found by any of the population of particles. The search space is explored by the particles moving through the space, the moves are decided by referring to the particles own performance so far and the collective performance of the entire swarm (Saka et al. 2013).

Cho et al. (2011) discuss some of the challenges faced by the PSO topology defined by Eberhart and Kennedy (1995). Using gbest helps particles to converge to a solution quickly as they are attracted to move towards the global best solution found by the swarm. This is a problem as often the particles are trapped in a local maximum because not enough of the search space was explored before converging. Another topology which Cho et al. (2011) describe is lbest. In this, particles can only communicate with a select number of other particles allowing for a more thorough exploration to take place, but convergence occurs slower than gbest.

2.5 Genetic Programming

Genetic Programming (GP) is the last area of bio-inspired computing that I will cover and is an area interested in using natural selection to automatically evolve computer programs (Miller, 2011). Koza (1992) describes the structure of a GP algorithm by stating that it starts with an initial population consisting of randomly generated computer programs. These programs consist of functions and terminals defined according to the domain of the problem. Functions can be anything from arithmetic or programming operations to mathematical or programming functions. The collection of allowed functions is called the function set. These functions can branch into other

functions or terminals. Terminals are the variables and constants allowed in the program. The collection of terminals is called the terminal set. Koza (1992) continues by stating that each of these programs are measured according to their fitness value, that is, how well it performs in the given problem. The algorithm performs in generation just like a GA and with each generation with the goal of improving the fitness values of the population each time.

2.5.1 Tree-based Genetic Programming

In tree-based GP, programs are expressed in the form of parse trees or abstract syntax trees. The internal nodes of the tree are elements taken from the defined function set and the leaf nodes are elements taken from the terminal set. For example, a tree built using a function set defined as $\{+, *, -, /\}$ and a terminal set consisting of $\{x, y, 2, 5, 1\}$ may look like the example shown in Figure 2-3 or Figure 2-4.

2.5.2 Initialisation of the Population

There are different styles of initialising the population of random programs and Poli et al. (2008) outline the full and grow methods, as well as a combination of both known as ramped half-and-half.

In the full and grow methods, a user set maximum depth parameter is chosen and the random individuals of the population are generated so that they do not exceed this depth. Poli et al. (2008) define this depth as the number of edges that need to be traversed to reach a specific node from the tree's root node. The full method is appropriately named as it generates full trees, meaning nodes are generated from the function set until the maximum depth is reached, and all the leaves are at the same depth. Each of these leaves may only be a terminal as choosing a function would cause the tree to exceed the maximum depth. The grow method allows for trees with more variation in the shapes and sizes then the full method. It differs by allowing any function or terminal to be selected until the user defined depth is reached.

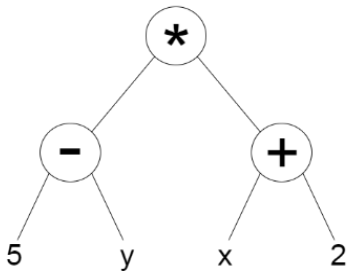


Figure 2-3: A tree built using the full method

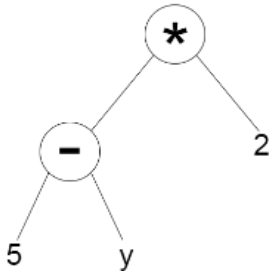


Figure 2-4: A tree built using the grow method

As Poli et al. (2008) state, the full and grow methods do not provide a wide array of tree shapes and sizes. Koza (1992) defines a method to combat this problem called ramped half-and-half. The method incorporates both the full and grow methods and is useful because often in GP the size or shape of the ideal solution is not known in advance. A maximum depth is still used but this time a range of depths from two to the maximum is used so that an equal number of trees is produced for each depth. For each value of depth, half of the trees are created using the full method and the other half are created using the grow method. Due to all full trees for a given depth having the same shape and grow trees shapes varying widely from each other, this allows the ramped half-and-half method to create a variety of sizes and shapes.

2.5.3 Operators in GP

I mentioned previously in Section 2.1 that genetic algorithms are based on natural selection and since this is also true for GP, it uses the same steps and operators although they are defined very differently in practice. Namely these operators are selection, crossover and mutation. Selection is defined the same way for GP and uses the same selection method. Poli et al. (2008) describe the other two operators at a high level as follows. In crossover for GP, a child program is created by combining parts of two selected parent programs. Mutation in GP is defined as the creation of a new child program by altering a randomly selected part of a selected parent program. These operators are used to progressively help to improve the fitness of the programs while still allowing the search space to be explored by not applying too much pressure.

2.5.4 Problems with Tree-Based Representation

Tree-based GP is one of the older methods of GP and as such has various problems associated with it. As Poli et al. (2008) state that in a high-performance environment, a tree-based representation can be too inefficient as it requires the storage and handling of many pointers. Another issue with a tree-based representation is that expressions in separate subtrees need to be re-evaluated multiple times wasting time and memory as well as adding complexity to the tree. An example of this is shown in Figure 2-5. Both the left and right subtrees have the expression $5 - y$ so it must be evaluated twice.

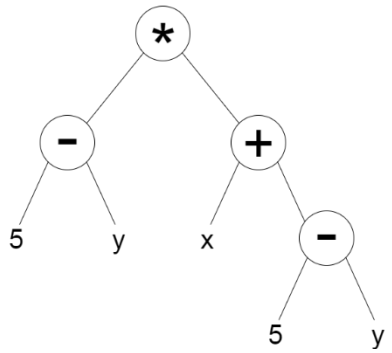


Figure 2-5: A tree with two subtrees containing $5 - y$

2.5.5 Other Types of Genetic Programming

There are other types of GP which aim to improve upon some of the problems faced in a tree-based representation. Some examples of these are Cartesian GP (Miller, 1999), Linear GP and Probabilistic GP along with a variety of others.

3 Availability of Bio-inspired Algorithms in R

In the following section, I will cover the areas of bio-inspired computing which were highlighted in the preceding section and briefly outline a few packages which provide access to each area.

3.1 Genetic Algorithms

Access to the algorithms in this area are covered through the two packages mentioned in Table 3-1.

Package Name	Implemented In	Provides	Source
GA	Entirely in R	Set of tools for implementing GA, Can define own operators Can define own fitness function	(Scrucca, 2013)
genalg	Entirely in R	Implementation of GA for multi-dimensional function optimisation	(Willighagen & Ballings, 2015)

Table 3-1: Packages providing access to genetic algorithms

3.2 Cellular Automata

The package identified in Table 3-2 provides an implementation of one-dimensional cellular automata but as mentioned in Section 2.2, multi-dimensional CA exist and there is no access to these currently within R. This is an area which can be developed.

Package Name	Implemented In	Provides	Source
CellularAutomaton	Entirely in R	one-dimensional cellular automata	(Hughes, 2013)

Table 3-2: Packages providing access to cellular automata

3.3 Artificial Neural Networks

Due to Artificial Neural Networks having a wide range of uses, there are many implementations to cover the different topologies of ANNs. Although there are many packages identified, most of them are built purely in R. These could be improved slightly by implementing the computationally heavy parts in a more efficient language

such as C or Java and interfacing to them from R. This is an area which could be improved.

Package Name	Implemented In	Provides	Source
rnn	Entirely in R	Multi-Layered RNN, Gated Recurrent Unit, LSTM NN	(Quast, 2016)
rsnns	Entirely in R	An R interface to the Stuttgart Neural Network Simulator	(Mergmeir & Benitez, 2012)
neural	Entirely in R	Radial Basis Function and Multi-layer Perceptron with an attached graphical interface	(Fritsch et al., 2016)
nnet	The neural network is implemented in C and an interface to it is provided in R	Feed-forward Neural Networks with a single hidden layer	(Venables & Ripley, 2002)

Table 3-3: Packages providing access to artificial neural networks

3.4 Swarm Intelligence

Table 3-4 shows three packages which implement Particle Swarm Optimisation with "ppso" providing an optionally parallel solution. All three packages are built entirely in R meaning these could be improved slightly by writing code to perform the demanding tasks. This is an area which could be improved.

Package Name	Implemented In	Provides	Source
pso	Entirely in R	Implementation of Particle Swarm Optimisation	(Bendtsen, 2012)
psoptim	Entirely in R	Implementation of Particle Swarm Optimisation	(Ciupke, 2016)
ppso	Entirely in R	Optionally parallelised Particle Swarm Optimisation	(Francke, n.d)

Table 3-4: Packages providing access to areas of swarm intelligence

3.5 Genetic Programming

The package "rgp" provides an implementation of tree-based GP but as previously mentioned in Section 2.5.5, other types of GP exist. This package has also recently been removed from CRAN as the author is no longer actively supporting it. The archived

versions of the software are still available however. This is an area which can be developed.

Package Name	Implemented In	Provides	Source
rgp	R with computationally heavy parts written in C	Implementation of tree-based GP	(Flasch et al., 2014)

Table 3-5: Packages providing access to genetic programming

4 Genetic Programming

While I have identified Cellular Automata as an area that could be developed, ultimately, I have chosen to build upon Genetic Programming. The reasoning behind this choice is that access to the area is limited and I also find this area more interesting that CA. Also, GP has more practical uses than CA has within the field of data science. As mentioned in Section 2.5.5, other areas of GP exist and one of these is Cartesian Genetic Programming (CGP). This is the area I would like to develop a package to provide access to.

4.1 Cartesian Genetic Programming

Cartesian Genetic Programming is a form of graph-based GP. Graph-based GP shares some similarities with tree-based GP in the way that programs are represented but also has some important distinctions. Graphs are like trees but represent the links between nodes using arrows showing directions and allow for cycles which allows for the reuse of previously calculated subgraphs. This solves one of the issues involved with tree-based GP (Miller, 2011). Figure 4-1 and Figure 4-2 show how a graph representation can reuse previously evaluated expressions.

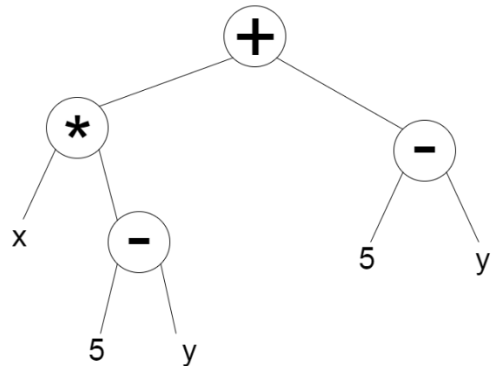


Figure 4-1: A tree with repeated subtrees of 5-y

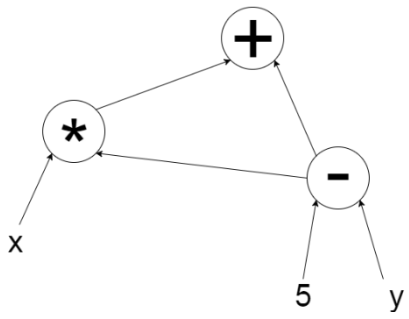


Figure 4-2: A graph reusing the 5-y subgraph

One way to represent a program in CGP is to use a list of integers referred to as a genome. A genome in CGP has a set size which stops this method of GP suffering from bloat which is a common problem in other methods (Turner & Miller, 2017). This set size is a maximum length of the genome and phenotypes of any size less than this can be produced allowing varied sizes of phenotypes. As Turner and Miller (2017) describe, a genome is composed of function genes, connection genes and output genes. Function genes contain integers which are corresponding entries in a function look-up-table. This look-up-table is similar to the function set described previously except that each function is mapped to an integer. Connection genes specify where the inputs for the function come from, and output genes specify where the output of the genome comes from. The structure of a genome for the graph in Figure 4-2 can be seen in Figure 4-3. The first value in each group is the function gene and the others are connection genes. The connection genes can either specify the inputs to the program or other groups. This can be seen in group 1 where "x" is an input to the program and "0" is group 0. The last group containing a single value is an output gene which specifies that the output of the genome comes from group 2. The function table for this genome can be specified as subtract (0), multiply (1), add (2).

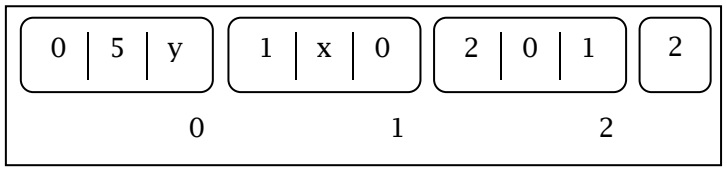


Figure 4-3: A genome representing the graph from Figure 4-2.

Another way of representing programs in CGP which is more visual is shown in Figure 4-4. In this representation the grid imposes a restriction on the maximum size of the program, but this grid still needs to be large enough to still allow for expressiveness and evolvability. The squares in each column are not allowed to be connected to other nodes in the same column as Miller and Smith (2006) describe. Miller and Smith (2006) also describe an additional parameter on the representation that defines how many columns away a column can connect to, this is known as level-back. The squares on the right show the input values to the program. The squares in the two-dimensional grid

show a random selection of functions from a function set. The square on the right marked as outputs has the same meaning as it does in the representation show in Figure 4-3, it marks which parts of the program the output result can be taken from. The arrows mark the direction and destination of the values being passed. The grid contains many functions which were unused in this execution which are represented by the greyed-out squares. During evolution subgraphs can also become greyed-out through mutation and become active at any point. Overestimating the size of the grid allows for more subgraphs to fade in and out of being used. This has been shown to be very beneficial in the evolutionary search involved in CGP (Miller & Smith, 2006).

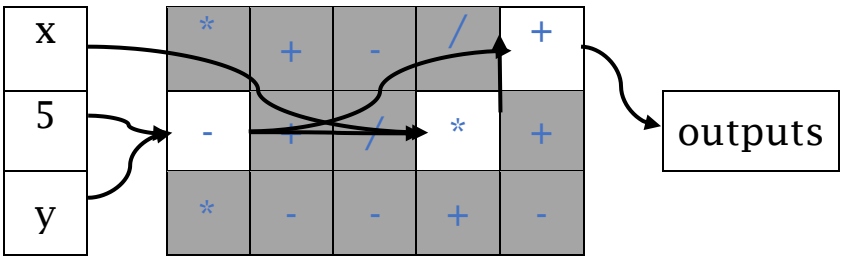


Figure 4-4: A two-dimensional grid representing the graph from Figure 4-2.

5 Implementations of CGP in Other Languages

5.1 JCGP

JCGP is an implementation of CGP in Java created by Eduardo Pedroni (2013). It implements the most commonly used evolutionary strategies as well as mutation methods while also allowing the user to define their own. It also allows the user to select whether the function nodes contain mathematical operators used for Symbolic Regression or whether they contain logic gates for a digital circuit representation. Again, the user can define their own set of function nodes by either selecting a subset of the functions provided by JCGP or to provide a completely different set. The key feature of JCGP is that it has an attached Graphical User Interface (GUI) to allow the user to view the chromosomes as they evolve. The ability to manually change the connections or function of each node in the chromosome is also supported. An interesting feature of JCGP is that it supports the ability to step through the generations of the evolutionary process which makes it clear to see how the chromosome changes over time. Another feature is that it allows the user to save the parameters they have set to a file which can then be loaded and used again later. Files containing the structure of the chromosomes can also be loaded but they architecture of the chromosome must be the same as the parameters in the program.

5.2 CGP-Library

CGP-Library is a cross-platform implementation of CGP written in C (Turner, 2014). It implements classic CGP as well as Recurrent CGP and the ability to use them with Artificial Neural Networks. It provides a C source and header file to be compiled along with the user's own source code allowing the user to use the library. This library also features the ability to view the structure of a chromosome by using the open source Graphvis (Graphviz.org, n.d) software which displays the structure of the chromosome

in a graph format. User defined functions, selection operators and reproduction operators can also be added easily.

5.3 CGP for ECJ

ECJ is an evolutionary computation system written in Java which supports a variety of evolutionary computation techniques. Although its origins are in Genetic Programming, it only provides Tree-Based GP. ECJ also supports plugins and there is a plugin developed to provide Cartesian Genetic Programming called CGP for ECJ (Oranchak, 2009). It supports real-valued and integer representations of the genomes. The plugin also includes a collection of sample problems and associated scripts and parameters files to run them. The plugin also includes three symbolic regression problems which can be run using either the integer representation or the real-valued representation. Classification problems are also included where each problem uses the real-valued representation. The classification problems are configured by default to split the dataset into a test set of 30% of instances and a training set of 70% of instance but this parameter can be altered.

6 R Package Creation and Development Tools

The availability and broad range of packages in R has been one of the main reasons for the success of R as a language. R is distributed with many standard or base packages which make up the R source code. There are more benefits to an R package than being an easy way to structure and share functions and datasets, packages can also be dynamically loaded and unloaded at runtime meaning they only occupy memory while being used.

There are different versions of packages. The diagram shown in Figure 6-1 shows the differences between the types of packages.

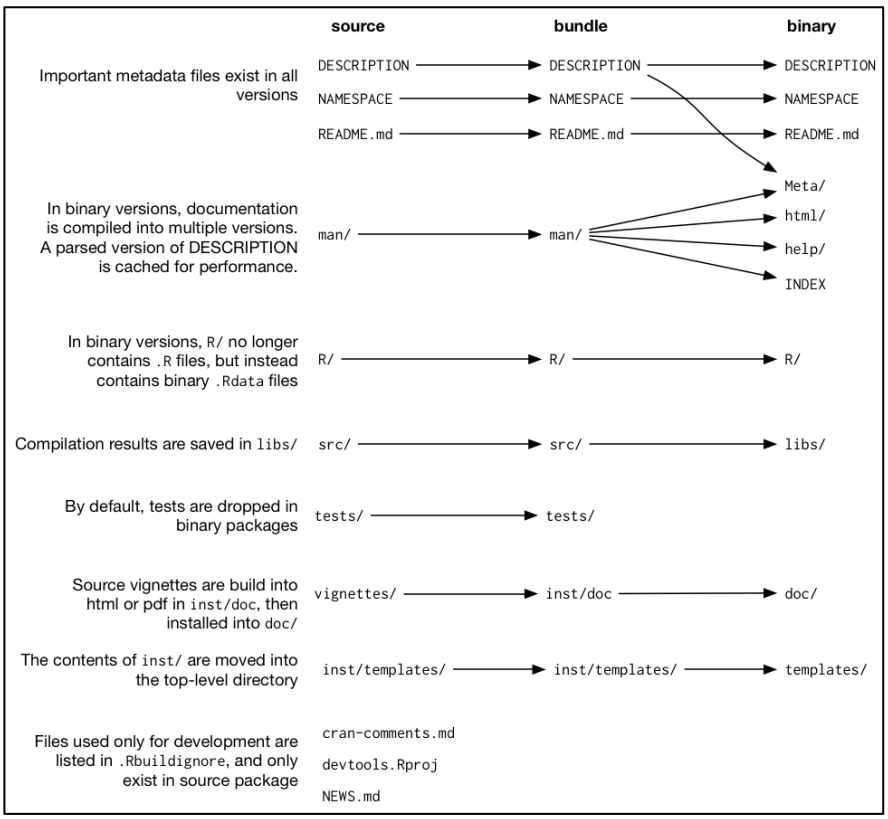


Figure 6-1: The differences between versions of R packages. Taken from R Packages (Wickham, 2015)

To create a package in RStudio there is a convenient menu to allow you to do so. After doing this the created package will contain the three minimal components. The first is a /R directory to contain code written in R. The next is a DESCRIPTION file which describes the function of the package and contains the details of who created the package and what the dependencies are. This is the information shown on the CRAN page for a certain package. The last is the NAMESPACE file which is vital for releasing the package on CRAN as it helps to encapsulate the package and reduce the conflicts it will have with the naming conventions of other packages. It does this by specifying which functions are available from outside of the package which are indicated with the export tag.

6.1 Packages and Software to aid Development

I have used various packages during development which reduced the learning curve of creating R packages, ultimately saving time throughout the project. These packages were also useful in helping to create a pipeline for writing R code, updating any associated documentation, and running tests to ensure the software still works as expected.

6.1.1 roxygen2

The package "`roxygen2`" (Wickham et al., 2017) provides a way of automatically generating documentation for a package. When used it adds a basic skeleton of keywords to the comment section above a function. Figure 6-2 shows a simple example of what happens when pressing the command from within R studio "Insert Roxygen Skeleton. After filling in the skeleton, entering the command "`roxygen2::roxygenise()`" in the console will create documentation files for each function. Updating the documentation is as easy as running the command again as it recognises which parts have changed. This also updates the `NAMESPACE` file saving time and reducing the chance of errors from writing the `NAMESPACE` file manually.

```r
#' Title
#'
#' @param x
#' @param y
#'
#' @return
#' @export
#'
#' @examples
mult <- function(x, y) {
  return(x * y)
}
```

Figure 6-2: Example of code with a roxygen2 skeleton

6.1.2 testthat

The R package "`testthat`" (Wickham, 2016) aims to make testing of software as painless as possible. It provides functions that make it easy to describe what a function is expected to do and can run tests automatically as code or tests are changed. It also displays test progress visually and shows which expectations passed or failed. Using this package helped to quickly identify any areas where behaviour has changed from what was expected, and also helped to catch a lot of cases I did not originally consider when writing various functions.

6.1.3 devtools

The aim of "`devtools`" (Wickham & Chang, 2017) is to make package development easier by providing functions to simplify many tasks. It provides convenient functions for installing and building packages, as well as incorporating documentation and testing into it by utilising the packages I have previously mentioned. Using this package, it is easy to create a simple workflow for when changes are made to code so that documentation is always updated, and unit tests are performed. "`Devtools`" also provides various functions for helping to release R packages onto CRAN as this process has significant overhead.

6.1.4 lintr

I will be using the "lintr" (Hester, 2017) package which checks that code written is compliant with a specific style and highlights any differences. The style I set this package to follow was Google's R style guide. This helped to increase the quality and readability of the code.

6.1.5 RStudio

RStudio (RStudio, 2016) is a free open-source integrated development environment (IDE) for the R programming language. It includes an R console, a code editor, tools for debugging, and provides an interface displaying which functions and variables are stored within the current environment. It also provides the features that can be expected from an IDE such as code prediction and direct support to version control facilities.

6.1.6 ggplot2

This R package "`ggplot2`" (Wickham & Chang, 2016) is a plotting framework for R which builds upon the base graphics that R provides but abstracts the more awkward parts. It is very simple to get started using this package as all that is required is specifying which data to use for each axis and the package will handle the rest. This is the package that I used to allow users to view the results as a graph.

6.1.7 plotly and shiny

Creating an interactive visualisation of the results of my program was something I assumed would have to wait until a further revision until I found the packages "`plotly`" (Sievert et al., 2017) and "`shiny`" (Chang et al., 2017). "`plotly`" provides a way to easily turn plots created in "`ggplot2`" into interactive web visualisations. These visualisations then were displayed in a web browser using "shiny".

6.2 R Workflow

These packages helped to develop two workflows to follow when I was writing R code which helped to automate tasks and keep different areas of my package consistent with one another.

6.2.1 Documentation Changes

The first workflow was used whenever a new function was defined, and I had added documentation for it inside a "`roxygen`" skeleton, or changes were made to existing documentation. The workflow consisted of:

1. Editing an R source file
2. Using "`devtools::load_all()`" which saves any changes made to source files and reloads the updated versions into the environment.
3. Using "`devtools::document()`" which uses "`roxygen2`" to generate any new documentation files and update any existing files with changes. This also keeps the `NAMESPACE` file up to date with the documentation.

This workflow ensured that any changes made were reflected in the packages documentation.

6.2.2 Functionality or Test Changes

The second workflow was utilised when a new function was defined, or an existing function was changed; or whenever new tests were added, or existing tests were changed. This workflow consisted of:

1. Editing an R source file
2. Using "`devtools::load_all()`" which saves any changes made to source files and reloads the updated versions into the environment.
3. Using "`devtools::test()`" which makes use of the "`testthat`" package to run all tests. Each test is run, and the outcome of each test is shown.
4. Repeat this process until all tests pass.

7 Design

7.1 Requirements Analysis

As I did not have previous experience of using Genetic Programming it was not initially clear what functionality the package I produced should have. To gain a better understanding of the important features of a GP package I investigated the implementations of CGP discussed in Section5, as well as an existing R package known as “rgp” which has been briefly mentioned in Section 3.5.

This project has now been released to the public but was originally planned to be a personal project so the requirements that follow have changed from when they were originally outlined. I originally aimed for the package to provide a broad range of customisation and functionality but when it came to development, it was decided that some of the requirements were either no longer practical or their priority has changed. Nevertheless, I will include the original requirements and discuss any changes made to these in Section 11.1.

The project is aimed towards users from a technical background, but it is still essential that the project is easy to use as they may not have knowledge in the field of Genetic Programming but still wish to use the software. Any prior knowledge should prove advantageous in their ability to pick up the package. The software will be provided with documentation describing the purpose and how to use each component as well as a guide for getting started in using the software.

The requirements of the package can be split into three main categories:

1. Allow users to perform Cartesian Genetic Programming on a set of data containing the desired results.
2. Allow users to change the parameters of the program and define their own where necessary.
3. Observe results from the program through textual outputs and graphs.

The following requirements will be assessed using the MoSCoW method which is used to assign priorities to requirements. The four categories are Must have, Should have, Could have, and Won't have. Each requirement will be given a Requirement ID to allow them to be easily referred to through the remainder of this document. The first number refers to the main category identified above, and the second number is the unique number of the requirement within the category.

7.1.1 Functional Requirements

Requirement ID	Description	Priority
FR1-1	The package shall implement Cartesian Genetic Programming	Must
FR1-2	The package shall allow users to run the evolutionary process on a population of chromosomes	Must
FR1-3	The package shall allow users to step through each generation in the evolutionary process	Could
FR1-4	The package shall allow users to pause and resume the evolutionary process	Could
FR1-5	The package shall allow users to load a file containing the desired results	Must
FR1-6	The package shall measure the fitness of a solution compared to the desired result	Must
FR1-7	The package shall be able to construct classifier models for a given dataset	Could
FR1-8	The package shall be able to build regression models through evolution.	Must
FR1-9	The package shall allow existing R data structures to be passed into the program	Should
FR2-1	The package shall provide a symbolic regression function set consisting of mathematical operations	Must
FR2-2	The package shall provide a logical function set consisting of logic gates	Should
FR2-3	The package shall provide the $(\mu + \lambda)$ and Tournament Selection operators to be used	Must
FR2-4	The package shall provide mutation methods to be used	Must
FR2-5	The package shall allow users to choose a function set from the included choices	Should
FR2-6	The package shall allow users to choose a selection operator from the included choices	Should
FR2-7	The package shall allow users to choose a mutation method from the included choices	Should
FR2-8	The package shall allow users to define their own function set	Should
FR2-9	The package shall allow users to select a subset of the functions from a function set	Could
FR2-10	The package shall allow users to define their own selection operators	Could
FR2-11	The package shall allow users to define their own mutation methods	Could
FR2-12	The package shall allow users to change the parameters of the program such as: • Number of columns in chromosome	Must

	• Number of rows in chromosome • Number of generations to run • The levels-back parameter • The population size • Random number seed	
FR2-13	The package shall provide multiple functions to be used for calculating the fitness of a solution	Should
FR3-1	The package shall display the results in a textual format in the R console	Must
FR3-2	The package shall create an output file containing the results	Should
FR3-3	The package shall display the results in a plotted graph	Could
FR3-4	The package shall output an R data structure containing the results through its return value to be used elsewhere	Could
FR3-5	The package shall allow users to load previous experiments to view the results as a graph	Could

Table 7-1: Functional Requirements of the package

7.1.2 Non-functional Requirements

The following non-functional requirements are arranged into categories of:

1. Performance
2. Visualisation
3. Usability
4. Robustness

The requirements are given a Requirement ID in the same manner as the Functional Requirements which is used to categorise them. The first number refers to the category of non-functional requirement identified above, and the second number is the unique number of the requirement within the category.

Requirement ID	Description	Priority
NFR1-1	The packages computationally intensive parts shall will be written in C, C++ or Java to improve performance	Should
NFR1-2	Only active nodes shall be processed when decoding the chromosome to improve performance	Should
NFR1-3	The package shall stop execution when a solution has been found	Must
NFR2-1	The created graphs shall be visually clear and understandable to the user	Must
NFR2-2	The textual format shall be understandable and only contain necessary information	Must
NFR3-1	The package shall provide convenient methods of abstracting away from the lower level functionality	Should
NFR4-1	The package shall minimise errors that cause it to fail	Must

Table 7-2: Non-functional requirements of the package

7.2 Program Structure

In previous project of mine, I created a Genetic Algorithm to solve the Travelling Salesman Problem and I feel the way I structured this project was effective. The structure is simple, but I feel it helps to understand which parts of the program are related. Although object-oriented programming is possible in R, the style I have used is more of a functional style and thus does not enforce the structure created. The package I have produced is a larger scale project than my previous project, so the structure had to be extended slightly. Each file in the package contains related functions. As covered in Section 10.2 when I discuss the correctness of the package, the unit tests are grouped into tests that correspond to each file. The file "`main.R`" contains the function used to run the program so I chose to give it this name as it contains the main function of the program. Another example is "`population.R`" which contains the functions which are required to create the initial population. As previously mentioned, the style of programming I have used does not enforce this structure.

7.3 Structure of a Solution

Instead of using the integer-based representation shown previously in Section 4-3, I chose to use the more intuitive grid-based representation shown previously in Section 4-4. This was so that it is simpler to visualise the resulting graph since a grid can be drawn and filled in with the actual values used in the program.

7.3.1 The Grid Representation

The grid is composed of different types of nodes which are input nodes, function nodes, and output nodes. Input nodes are the nodes which are loaded with values to be propagated through the structure of function nodes. Function nodes are nodes which apply a selected function to its input values and store the result within them. Function nodes may have one or two inputs depending on the chosen function. Output nodes are nodes which simply select which node of the solution is used to give the overall result of the solution. Each of these nodes is given a unique identifier "`chromoID`" so that any

node can be uniquely identified, and it is easy to tell which type of node it is from this ID. An example structure is shown below in Figure 7-3. In this example, there are two input nodes, a 2x3 grid of function nodes, and a single output node. Example functions are also shown in the function node structure to show that function nodes with different functions take a different number of inputs. The inputs for a node are shown using arrows. The structure created is a feed-forward graph.

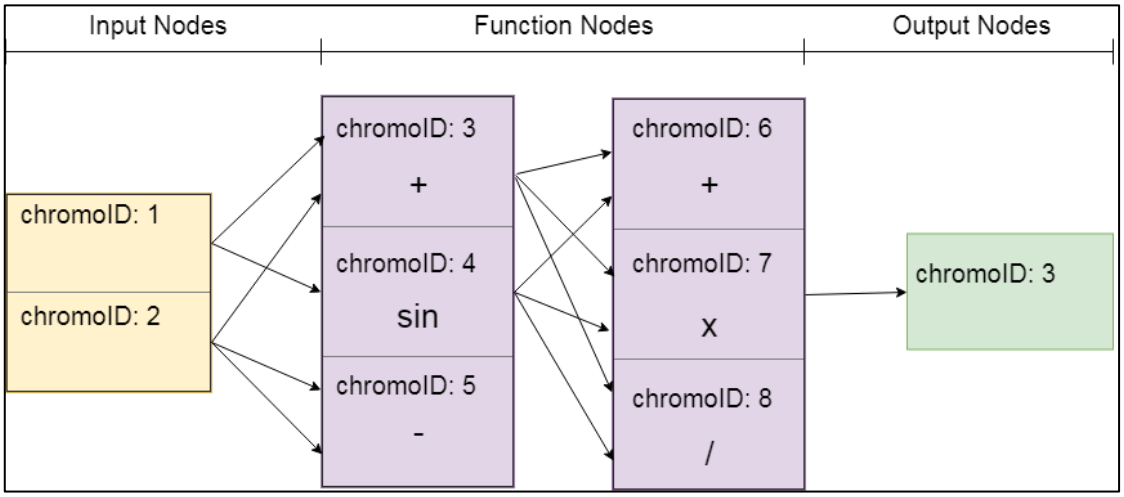


Figure 7-3: The structure of a solution using the grid representation

7.3.2 Structure of Types of Nodes

As each node has different components, each must be created differently from the others. The components of each is mentioned above in Section 7.3.1 but these components must be formalised into how they will be stored in R. The first and simplest component was the "`chromoID`", this was simply stored as an integer. The next was the "`value`" that the node could take, this was stored as a numeric so that real numbers can be used. Both components so far are common to all types of nodes. The diagram in Figure 7-3 can be misleading as although the graph is a feed-forward graph, each node does not store where its value is sent to, it's the reverse, each node stores where its value comes from. This is the next component known as "`inputs`" and is present in function nodes and output nodes. This was represented as a vector of integers where each integer is the "`chromoID`" of the node which is used as input. The final component which is only present in function nodes is the "`funcID`". This is simply the unique identifier for the function from the function set. The function set as discussed in

Section 3 and 4 is the list of functions that the program can use. The structure of each list can be seen in Figure 7-4. With each node having common components it makes sense in an object-oriented style to have a node superclass and have each of these inherit from it but as previously stated, I have not used an object-oriented style.

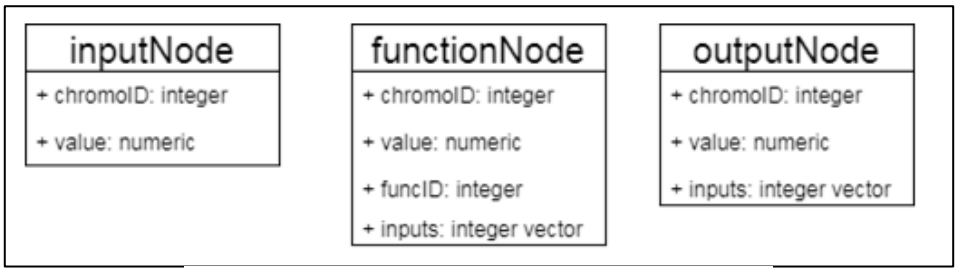


Figure 7-4: Structure of each type of node

7.3.3 Choosing the Correct Data Structure

The next step was to choose a data structure to store each type of node. I wanted the representation used to store solutions to reflect the representation shown above so that it was easy to translate from a visual display to the actual representation used inside the program. I feel this ability would greatly benefit others and myself when trying to decipher what the solution is doing, and how it is structured. Choosing the correct data structures to use is an important part of any software project and it can have detrimental effects if an incorrect choice is made.

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Figure 7-5: R Data Structures (taken from Hadley Wickham's Advanced R)

R has a variety of data structures available to the developer. These are shown above in Figure 7-5 which is taken from the book, Advanced R (Wickham, 2014). The data structure I opted to use were data frames. This was for several reasons. The first being that as previously mentioned I wanted the data structure to be easily translatable to the

visual representation and this required a two-dimensional data frame. This narrowed my choice down to matrices, arrays or data frames. The next reason was that the data to be stored into the structure was the data for each node. This means that the homogenous data structures could not be used since the data to be stored inside each node also was not homogenous. This left only one choice which was data frames. The way this approach was implemented is discussed in Section 8.1.

7.4 Designing the Programs Flow

To design the flow of the program, I wrote down what processes are repeated in an Evolutionary Algorithm and then wrote these into basic pseudocode which is shown in Figure 7-6. The steps shown here are the absolute minimum structure of an Evolutionary Algorithm and help to understand where the evolutionary steps happen. While there are extra steps that will be included in the final version of the main algorithm, this is the structure I based the flow of the program from.

```
main function {

  create initial population

  calculate fitness of initial population

  while maximum generations is not reached and a solution has not
        been found

    store the best solution found so far

    perform selection and create the new population

    calculate the fitness of the new population

    check if a solution has been found

  end while

  store the best solution found

  return the best solution found
}
```

Figure 7-6: Pseudocode showing structure of main function

7.5 Designing a Decoding Algorithm

Decoding is the process of getting an output value from a solution with given inputs. One method to do this is to start at the output node of a solution and to recursively work down through the solution and calculate the value at each step back up the solution. This is the method I plan to use as this allows only the nodes which are connected to the output node to be used. An algorithm for this is shown in Figure 7-7. The algorithm repeatedly calls itself on each of the nodes it takes as input until it finds an "`inputNode`" which is a node that has the value to be propagated through the solution. This value is then passed through the solution as each function node applies its function to it before finally producing an output value.

```
decode(node) {

  if this node is an inputNode

    return this nodes value

  else

    Store the inputs to this node

    firstValue = decode the value of the first input

    if there are two inputs to the node

      secondValue = decode the value of the second input

      result = apply this nodes function to firstValue and
               secondValue

    else

      result = apply this nodes function to firstValue

    end if

  end if

  return result
}
```

Figure 7-7: Pseudocode for a decoding algorithm

7.6 Designing how to Calculate Fitness

The fitness of a solution is the measure of how suitable the solution is to a given problem so calculating the fitness of a solution is an essential element of this package as highlighted by the functional requirement, FR1-6. I aim to distribute multiple fitness functions with the package to allow the user to choose the function which best applies to their scenario as mentioned in FR2-13. To allow for this, it is essential that the method I use to calculate fitness must be customisable. The pseudocode shown in Figure 7-8 is the structure for a function which calculates the fitness of a given solution. To calculate the fitness, it compares what result was given by decoding the solution and compares this to the expected value provided by the dataset passed as a parameter. This is repeated for every entry in the dataset which gives the total error for this given solution. The "`fitnessFunction`" parameter is the fitness function that the user has chosen to use to calculate fitness.

```
calculateFitness (solution, dataset, fitnessFunction) {

  create a list to store results

  for each entry in dataset

    load the solution's inputs with the entry from dataset

    decode this solution to get an output value

    store this output value in the results list

  end for

  create a list so store the results and the expected
  output from the dataset

  get a fitness value by passing this list into
  the fitnessFunction

  return the fitness value
}
```

Figure 7-8: Pseudocode for an algorithm for calculating fitness of a solution

The fitness functions I plan to provide are functions which calculate the Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). Both functions will accept a list which contains the predicted value and the actual value where the predicted value is the value given by decoding, and the actual value is the value from the dataset, and both functions will return the error calculated.

There will also be a convenience function provided for calculating the fitness of the entire population. This function will simply call the "`calculateFitness`" described above and store the result into the solution.

8 Implementation

In this section I will cover the important parts of the package I have created known as caRtesian. Where there is a corresponding design section, I will link back to this and discuss how this design was converted into an implementation.

8.1 Structure of a Solution

The design of a solution is covered in Section 7.3 as well as what information is required to be stored in the various types of nodes. This section will cover how this was implemented in R. The original idea was to use a data frame to represent the grid of nodes and store a node into each entry so that all nodes could be represented under one structure, but this solution proved to be more difficult to work with than it was worth. Instead, I still opted to use data frames but to use a different data frame for each node type, this meant there were three data frames in total. The different components were each given their own column in the respective data frame and each row in the data frame corresponded to a node. This meant that a solution was no longer represented in a two-dimensional manner as I had planned. The structure of each data frame is shown below in Figure 8-1. The data frames represent the solution shown in Figure 7-3.

Input Nodes

	chromoID	value
1	1	NA
2	2	NA

Function Nodes

	chromoID	value	funcID	inputs
1	4	NA	1	c(1, 2)
2	5	NA	4	1
3	6	NA	2	c(2, 2)
4	7	NA	1	c(3, 4)
5	8	NA	3	c(3, 4)
6	9	NA	8	c(3, 4)

Output Nodes

	chromoID	value	inputs
1	10	NA	7

Figure 8-1: Data frames containing the structure used in Figure 7-3

It is easy to see how the input nodes and the output nodes map to the diagram in Figure 7-3, but it is not so easy to understand the structure of the function nodes. This data frame could represent a 2x3 grid, or a 3x2 grid and although this results in the same

number of nodes, the resulting shape is very different. This is exactly what I aimed to avoid but a sacrifice was required to be made in either the difficulty of working with the nodes stored directly into a data frame entry, or with the translation of the data structure used to a visual representation. Once each of these data frames have been created they are combined into a list where each element is a data frame and they are given appropriate names making them easy to refer to. The list is shown below in Figure 8-2. This list is named "solution" and is then stored into a larger list which contains the entire population of solutions.

```
solution        list[3]         List of length 3
  inputNodes    list[3 x 2]     A data.frame with 3 rows and 2 columns
  functionNodes list[6 x 4]     A data.frame with 6 rows and 4 columns
  outputNodes   list [1 x 3]    A data.frame with 1 rows and 3 columns
```

Figure 8-2: Showing the structure of the "solution" list

8.2 Function Sets

The function set is the set of functions that a node can use to apply to its input values. There are various requirements that relate to the functionality surrounding function sets which are FR2-1, FR2-5, FR2-8 and FR2-9. These collectively describe the amount of customisation that should be allowed within the package concerning the function set. Since the users of the package are expected to have experience of using the R language, a solution to provide these requirements has been implemented. Different functions have been provided in the package which return different selections of mathematical functions to use. Each of these different selections can then be combined with another, or a subset can be taken easily by using simple vector manipulation. A helper function "constructFuncSet" has also been provided to help with the case where the user would like to construct their own function set entirely instead of using the provided sets.

The definition of each function contained in a set are stored within a vector composed of the functions name and its arity. Each of these definitions is then combined into a singular vector which is finally stored into a data frame. To use one of these functions stored, a function built into R known as "do.call" is used where the first parameter is

the name of the function and the second parameter is a list of the arguments to the function to be called. An example showing how R's arithmetic functions are stored and can be called is shown below in Figure 8-3.

```
#Singular Vector of Function Definitions
functionDefs <- c(c("+", 2),
                  c("-", 2),
                  c("*", 2),
                  c("/", 2))

#Data Frame of Function Definitions
  funcName arity
1        +     2
2        -     2
3        *     2
4        /     2

#Returns the result of 1 + 2
do.call("+", list(1, 2))
```

Figure 8-3: How function sets are represented and can be used

An admittedly better approach to storing the functions contained in a function set is to store the function directly and store the number of arguments to the function as the arity. I discuss this approach and the issues associated with it in Section 9.2.

8.3 Creating the Initial Population

As discussed in Section 8.1 created solutions are stored into a larger list. This larger list is known as the population. I will discuss how valid solutions were created by initialising each solution with values.

8.3.1 Creating the Input Nodes

As mentioned previously there are three types of nodes: input nodes, function nodes, and output nodes. The first type of nodes that I created were input nodes. These are created using the "`generateInputs`" function. The function takes "`inputSize`" as a parameter which is the number of input fields present in the dataset used with the program. The program also requires random constants to be available to use as input to

function nodes so this "inputSize" variable is increased by one to allow for the constant to be stored here. This solution only allows for one constant to be used through the whole solution. Issues relating to this approach are discussed in Section 9.4. As discussed in Section 8.1 data frames are used to represent each type of node so the next step was to create a data frame to store the input nodes. The data frame created was the length of "inputSize" and each column was initialised with values. The "chromoID" of each row was initialised from one to "inputSize". This resulted in each input node having a unique identifier as required. The next step was to initialise the "value" field where each value was set to "NA" since there were no values to store yet and then convert this to a numeric type so that the appropriate space is allocated in memory to hold the information that will be stored here eventually. Allocating the correct amount of space initially saves computation time later as dynamically growing structures in R are slow. Then the row to store the random constant was given the constant value that it will use for the remainder of this solutions life time. Lastly, the data frame containing the input nodes was returned.

8.3.2 Creating the Function Nodes

Next the function nodes were created. These were created using the "generateFunctionNodes" function. This function takes several parameters which are: the starting "chromoID" to use, the number of rows and columns to be used in the function node structure, the levels back which is a new parameter and will be discussed in this section, and finally the function set containing the functions each node can use. In the same manner as with the input nodes, the appropriate data frame is created first with the correct types in each column before information was loaded into the data frame. This is achieved by the "createFunctionNodesStructure" found in Appendix A.

The next part of this function caused me some issues and I have since found a more elegant solution to this. Both the issues and the new solution are discussed in Section 9.3. The problem was how to restrict where a function node could take its input values from which is what the levels back parameter is used for. The levels back parameter

specifies how many columns back a given function node can take values from. Every function node regardless of the levels back can take an input node as input. This was made more difficult due to the nodes being stored into a data structure which did not truly represent the two-dimensional structure I had hoped for. The way I implemented this was to create a matrix which initially contains the "`chromoID`" of each node in the input node structure created previously. The number of rows in the matrix is equal to the number of levels back add one. A row represents a column within the function node data frame that is valid to be used as input, and there is one extra row to store the input nodes since these can also be used as inputs. The number of columns was initially set to the number of input nodes, but this caused issues when the there were more rows required in the function node structure than there were in the input nodes. These issues are also discussed in Section 9.3. The solution currently used takes the maximum value of either the number of input nodes, or the number of rows required. However, this too brings its own associated problem. The first time the valid input matrix is created, the matrix is filled with the "`chromoID`" of each input node. A row of the matrix is then replaced after one column of function nodes has been created which the intended functionality. However, there are still multiple rows of input nodes which leads to a bias when using the sampling function to choose an input since there is at least double the chance to pick an input node compared to a function node. An example showing this bias is shown below in Figure 8-4. To combat this a wrapper for the "`sample`" function was implemented which removes any "`NA`" values which are caused when there are more input nodes than function nodes to enter, and then removes any duplicates. This allows for unbiased sampling.

To cover this is more detail , there is a nested loop which iterates over the number of columns required in the function nodes and the other iterates down through the number of rows. The inner loop creates a function node giving each node a unique identifier, assigns it a random function from the function set, and chooses input locations, then stores it into the correct position in the function nodes data frame. The "chromoID" of

the node is added to a vector which is the vector that will be placed into the matrix containing the valid input locations after the inner loop completes.

```
Initial Matrix which                Matrix after one column
contains only input                 of function nodes has
node chromoIDs                      been added. Notice the
                                    double entry in row 1 and 3

     [,1] [,2] [,3]                      [,1] [,2] [,3]
[1,]    1    2    3             [1,]    1    2    3
[2,]    1    2    3             [2,]    4    5    6
[3,]    1    2    3             [3,]    1    2    3
```

Figure 8-4: Matrices showing the bias after function nodes have been added

8.3.3 Creating the Output Nodes

The process of creating the output nodes is much simpler than the function nodes and is very similar to the input nodes. I designed this section of the program with the idea in mind that I would like the program to have multiple output values possible, but this idea was scrapped in later stages due to it being a lot more difficult to work with. The "`chromoID`" and "`value`" fields were created in the same way as with the input nodes. Output nodes also have input values but there are no restrictions as with the function nodes and can take their value from anywhere in the solution so calculating the input value for an output node is as simple as selecting a random "`chromoID`".

8.3.4 Wrapping Up

After creating the three different types of nodes, they are wrapped into a single list and given names making them easier to refer to. They are then stored in a larger list called "`population`" and returned to the main function of the program.

8.4 Decoding

The design shown in Section 7.5 is the design I went forward with when implementing the decoding process of a solution. I managed to produce a recursive function which when given a node will recursively call itself on each of the nodes inputs, then calls

itself again on each of the next nodes inputs. This process continues until nodes from the input nodes data frame are reached at which point the value is sent back up the recursive stack with each function node applying its function, altering the value.

This approach did produce the correct output value and was the method I used going forward through the rest of development. I then discovered a problem in that values were not being stored in the function nodes as they should be. This meant that nodes were being re-evaluated each time they were used which is exactly what Cartesian GP was designed to avoid. This required me to implement a new method of decoding a solution.

The second implementation required extra work in that the nodes that are used to get an output value must be found first so that only the required nodes have their value calculated. This was to save time in the decoding process and to satisfy the non-functional requirement NFR1-2. This work was performed by the "`nodesToProcess`" and "`traverseFunctionNodes`" functions which can be found in Appendix C and D respectively. The new decoding function worked similarly to before except instead of starting at the output node, it would start at the first node returned by "`nodesToProcess`". The algorithm would then calculate the value for this node before moving to the next node used. The package would have produced the same results with either implementation, but the first implementation would have been true CGP, so this implementation was not used when decoding the solution.

8.5 Calculating the Fitness of a Solution

Calculating the fitness of a solution is one of the most important areas of my implementation and is also the most demanding resource wise but this is common in Evolutionary Algorithms. For the most part, the pseudocode shown in Section 7.6 is what I followed when creating this functionality, but some extra steps were required to be added. These extra steps are more specific to my implementation and will not be covered, they are operations such as creating variables to store lists which are used or

creating variables to store values which are nested inside other lists so the command to access them was shorter.

For the most part this function is a wrapper for the decoding function covered in the previous section so most of the issues involved with this function were already encountered when implementing the decoding function. One issue which is yet to be mentioned, and is also related to decoding, is that when a function node chooses to use the square root function, "`sqrt`", the program will output warnings whenever this function is used with a negative number. This is because there is no square root of a negative number. R handles this case by setting the value as "`NaN`" for Not a Number. I originally assumed this would be an issue when trying to sort the fitness of a population so that any solutions with a fitness value of "`NaN`" were punished but the function I used to sort the fitness values contained a hidden parameter which can be used to handle this. The function is shown below in Figure 8-5. The parameter "`na.last`" inside the "`order`" function tells this function to place any entries with a fitness of Not a Number to the end.

```
#' sortPopulation
#'
#' Sorts the population from lowest fitness value to highest
#'
#' @param population the population to be sorted
#'
#' @return the population after sorting
#'
sortPopulation <- function(population) {

  #Extract the fitness values from the population
  fitnessValues <- sapply(population, "[[", "fitness")

  #Get the index ordering that will put the values
  #into ascending order and reorder the population
  return(population[order(fitnessValues,
                          decreasing = FALSE,
                          na.last = TRUE)])
}
```

Figure 8-5: sortPopulation function

8.6 Mutation

There are a number of functional requirements relating to mutation, namely FR2-4, FR2-7 and FR2-11. Not all of these were completed due to the mutation method I have chosen to implement causing a lot of problems. This was most likely due to the lack of a design for this complex feature.

Since there is no associated design section, the algorithm for mutation will be described here. The type of mutation implemented is Point Mutation which in this case mutates fixed points of the solution. It is percentage based and mutates a maximum of 10% of the solution. The parts of a solution which can be mutated are the functions to use within the function nodes, and the inputs for a function node or output node. Each input or function mutated counts as one mutation. Random nodes are selected for mutation and if the chosen node is a function node then either the function used, or its inputs are chosen to be mutated. Otherwise the node chosen must be an output node and its inputs are mutated. After these mutations are applied, the fitness of this solution is reset to "`Inf`" for Infinite. I will continue to discuss the process of mutating the function used or the inputs to a node.

8.6.1 Mutating the Function Used

To mutate the function used, a random function is selected from the function set. This random function can be the same as is already in use, perhaps this an area that deserves more experimentation with so that when mutation is applied, a change is always made. The next step is to check if the arity of the old function used matches the newly chosen function. In the case where they match the newly chosen function is simply stored into the node and the node is returned. The next case is where the old function had a greater arity than the new function, an input must be removed to match this new function, or an error will be thrown when passing two arguments to a function which only expects one parameter. The process of removing in an input is not as simple as it seems. A design choice was made at this stage to randomly choose the input to remove. The alternative could be to always remove the first input, or always to remove the second

input. The final case is the most complex and this when the new function has a greater arity meaning another input must be added. It is not as simple as choosing a random node since the constraints used when initially creating these inputs must also be obeyed. These are that the node can only access other nodes within the levels back range. Once this new input has been selected it must be stored into the function node. Another design decision was made here in that the new input will be the second input in the list. An alternative could be to randomise the ordering of these, but I feel this is too obstructive to the structure of the solution being mutated.

8.6.2 Mutating the Inputs Used

Mutating the inputs is simple for an output node, a random unique identifier is chosen and is stored into the output node. Mutating the inputs for a function node however is more difficult. Again, the constraints such as the levels back range must be obeyed here so the first step is to get a list of valid input nodes for the node being mutated. Next, a random input is generated. Similarly, with the way a function is mutated for a function node, the new input is not guaranteed to be a different value. Again, this is a topic I will need to investigate further. The number of inputs the node currently uses is checked as mutating a one input node is different to mutating a two-input node. For a one input node, the new input can be written directly into the node. For a two-input node the input to be changed must be chosen then the new input can be written into the node.

8.6.3 Issues caused by Mutation

As previously mentioned, due to the lack of design for this functionality, it brought difficulties. The implementation discussed above is the version distributed with the package, but multiple revisions took place before a working version was produced.

The first issue was that I did not consider the case where the function used in a function node was being mutated and the new function chosen had a different arity. This caused errors when trying to decode the solution and the program would crash. One error was that too many parameters were passed to a function when the newly selected function had a lower arity than the previous function. The other was the opposite case; where not

enough parameters were passed to a function because the newly selected function had a higher arity than the previous function. This issue was simple to fix once it was identified since the error reporting that R provided for these errors was intuitive to understand. Example errors are shown below in Figure 8-6. The first error is the case where not enough parameters were passed, the second is where too many parameters were passed.

```
Error in *7 : invalid unary operator

Error in sin(-79, -79) : 2 arguments passed to 'sin' which requires 1
```

Figure 8-6: Errors given by R for invalid function calls

The second issue encountered was more difficult to find and involved stepping through my code line by line during execution using RStudio's debugger. Using this I was checking the values being passed around and I realised where my problem was. The error I was receiving was:

```
Error in sample.int(length(x), size, replace, prob) :
   invalid first argument
```

This is the reason the debugger was required since there was no stack trace and the error occurred in one of R's built in functions, "`sample.int`". The issue was sample was being passed an empty vector and the "`size`" parameter was set as 1 forcing it to return a value but there was nothing that could be returned. This was happening after a function inside a function node was mutated and the new function had a higher arity than before. This problem sounds similar to before, but it was not solved in the same manner. The only function nodes which raised this error were those contained in the first column of the function node structure. This was due again due to the lack of design for this stage and I did not account for function nodes still using the input nodes as inputs. Since the function node was within the first column of function nodes, there were no valid inputs to it.

Once this trickier issue was identified and understood, the fix was as straightforward as allowing the function nodes still to sample from the input nodes data frame. A unit test was added to capture the case where a function node in the first column is being mutated to ensure this error does not appear in future.

The last issue encountered with this functionality was that whenever an input of a function node was to be mutated, all the inputs were changed which is incorrect since I am using a percentage-based mutation method. This means that too many mutations were taking place as well as this being a very disruptive operation. The solution was to generate a new input and then write it into either the first or second input of the function node.

9 Issues Faced During Implementation

The issues that I was confronted with during implementation which have not yet been discussed are covered in this section and where possible a solution to a given issue will also be discussed. I feel it is important to also document these issues as they are problems that were required to be overcome and help to highlight areas of which could use refactoring.

9.1 Storing a Function directly in a List

As mentioned in Section 8.2, a better approach to representing function sets would have been to use lists and to store the function directly. This removes the need to use the "do.call" function since the function can simply be called with arguments passed to it. I attempted to use this approach but since this part of the package was still early in the development process, I had trouble with this. The trouble was how to calculate the arity of the functions to be used. I understood how to calculate the arity of a function that I created, and this could be done through calling the "`formals`" function and then counting the length of the returned values. However, this approach did not work for R's mathematical operators since they part of R's base package. When the "`formals`" function was used on these, "`NULL`" was returned. This is due to the functions of R's base package being known as primitives. Some of R is implemented in C and primitive functions are exactly that, they store the C code that implements the function and are compiled at build time (Cran.r-project.org, 2018). These functions do not have formals and instead have "`args`" so to get the number of arguments to a primitive function, first "`args`" must be applied and then "`formals`". This is the approach I would like to use upon refactoring of this project but will also require changes elsewhere in the program wherever the function sets are used.

One problem with this approach is that the basic arithmetic operators "+", "-", "*", and "/" cannot be stored this way as they raise syntax errors. To store these wrapper functions would have to be implemented that call these operators and the wrapper functions would be stored into the function set.

9.2 Checking Valid Inputs to a Function Node

As mentioned in Section 8.3.2, issues were faced when computing the valid inputs to a given node. A working version was created quite quickly which I assumed to work but it assumed that the number of rows to use in the function node structure would be the same as the number of input nodes which is a terrible assumption to make. This was down to a mistake in the values I chose to test this functionality with. However, the issue did not arise until later in development and the problem was confusing to find since there was no error produced but instead, just unexpected behaviour.

When creating the function nodes, once the most recent column was created it was to replace the no longer valid inputs in the matrix but when there were more nodes created than there were columns in the matrix, it would cause values to be cut off. For example, if there were two input nodes and four function nodes were just generated, then when adding these function nodes to the matrix two values would be lost.

As my R skills have developed through this project and I am more aware of different functions available to an R developer I found a new more elegant way to implement this functionality. This method was to use the "`getValidInputs`" function I have created which can be found in Appendix B. This function takes as an argument the "`chromoID`" of the node to get the valid inputs of, the "`functionNodeRange`" which is simply the "`chromoID`" of each function node, and lastly a list containing the rows, columns and levels back used to create the function nodes. This function puts each of the identifier of each of the function nodes into a matrix similarly to before but this time it is structured in the same format as the function nodes meaning they have the same number of columns and rows. The function which simplifies the issues I was having

before is "`which`". This function returns the indices of an object where an expression is true. I had used this function previously in my program, but I did not know about the optional "`arr.ind`" which instead returns the row and column of where an expression true. This allowed me to find the index of the column containing the given node which then allowed me to find the valid columns through some operations applied on this column index. A subset was then taken from the matrix containing the identifiers of the function nodes by using the resulting valid columns. This produced all the valid input nodes for a given node. This was then converted to a vector and returned.

This implementation is much simpler and easier to debug so the old implementation will be replaced with this in the next version of the software.

9.3 Allowing the use of Multiple Random Constants

A GP library should provide the ability to use random constant values in its solutions so that functions such as $y = x + 5$ can be represented. My implementation does this through adding an extra input node and storing a random integer inside between -10 and 10. This allows the value chosen for this random constant to be retained through the life time of the solution and can be used multiple times.

Later in development I encountered the problem where a function which requires two distinct constants could not be represented. Due to how late this problem was discovered and the design of some earlier parts, the amount of work to rectify this problem was large and would require extensive testing to ensure the behaviour of the program did not change. Despite this, I will continue to discuss my attempts to solve this problem and explain a solution which will be attempted in a future version of the package.

The first solution I tried was to store the function which returns a random value into the input node data frame instead of the random value it returns. This would allow for multiple constants to be used through the program since each time it would be used, a random number would be returned. The issue with here is that this meant the behaviour

of the solution would change each time it was used. For example, if a solution used a random constant then it would call the function which returns a random value. The next time this function is used it would return another random value which may well be the same value, but this is unlikely. More concretely, if the solution represented a function such as $y = x + random\ constant$ and random constant was set as 5 which produced no error meaning it was a perfect solution, the next time it was called this random constant could be a 6 which is no longer the correct solution.

This solution is partly correct, but I must implement some manner of retaining the random constant value returned. One way of doing this is to create a new type of node called a constant node which is structured similarly to an input node in that it only has a unique identifier and a value. Each time a random constant is to be used it will either sample a new value or use on of the already existing constants. Any constant values that are no longer used by any node in the solution should be removed to avoid excess memory being used. Another way to do this which may require less refactoring of code is to add a field to the function node structure which can store a random constant if it chooses to use one. However, I feel the first solution is more sensible.

9.4 Allowing the Selection Method to be Changed

Allowing the selection method used by the package to be changed was not an essential element but was marked with a priority of "Should" as can be seen in functional requirement FR2-6 and was a feature I thought would greatly improve the customisation of the parameters used in evolution.

I implemented two different selection methods, "`muLambdaStrategy`" and "`tournamentSelectionStrategy`" and both are distributed with the package. For users of my package to select the selection method they would like to use; the chosen function must be passed into the "`cgp`" function of the program by storing their choice in the "`selectionMethod`" parameter. This parameter can then be treated as a function and putting parentheses after the parameter like so: "`selectionMethod()`" will call the

function passed into it. The problem arises when passing parameters into this function. Although both selection methods I have provided use the same parameters in the same order, this does not mean that any selection methods implemented by an outside user will also be the same. So, calling this function was problematic.

The solution I have used to this problem only partly solves the issue and I will have to investigate further into how to resolve this problem fully. However, I will still discuss this part solution. I changed the "`selectionMethod`" parameter to expect a list containing the selection method to use along with the parameters to pass to the function. These parameters are expected to be in the same order as the selection methods distributed with the package. There is a function "validSelectionInput" which can be used by the user to check the list being passed in contains the correct structure. This part solution is still too restrictive for the selection methods that can be passed into the program, but it allows the user to change between the methods distributed with the package. It is also possible for the user to define their own selection methods to use and pass them into the program given that they accept the same parameters.

9.5 Division by Zero

A problem was encountered when decoding the solution and propagating the input data through each of the function nodes to get an output value for the solution. To understand the problem an example must first be given. Given a dataset of random inputs called x which are in the from 0 to 10, and an output value y which is the result of the function $y = 10/x$. The solution could correctly identify to choose a random constant value of 10 and divide this by the input value. This could work perfectly for every input except when the input value was zero. Division by zero in most programming languages results in a crash but with R, the value is represented as "`Inf`" for Infinite. This would result in this solution not being selected during the next generation since its fitness is extremely high even though it is a perfect solution. I opted to remove the division function which is admittedly a poor decision on my behalf as this makes the example function given above very difficult to represent.

A solution to this issue is to use a protected division operator which returns a result of one when dividing by zero. This approach is discussed in Cartesian Genetic Programming (Miller, 2011).

9.6 Plotting Average Fitness over Generations

The details shown as the evolutionary process of the program is run display the generation count, the fitness of the best solution so far, and the average fitness of the population. I have provided a function with the package which displays the best solution so far against the generation count as a line graph. I originally wanted to also display the average fitness on this graph, but this was problematic.

Due to fitness values of a solution ranging from zero to Infinite and Not a Number, the points were difficult to plot. Even after removing any points with Infinite or Not a Number values, it was difficult to find a scale to use for the y-axis since the values still contained a very large range. It meant that differences could not been seen in the best solution points at all since these points were very close together compared to the wide range contained in the average fitness. An example of how a plot with both the best fitness and the average fitness plotted over the generations is shown in Figure 9-1. In the example, the blue lines and dots are the average fitness and the red line and dots are the best fitness. It is impossible to see and difference in values between most of these points due to the extreme range of values for the average fitness.

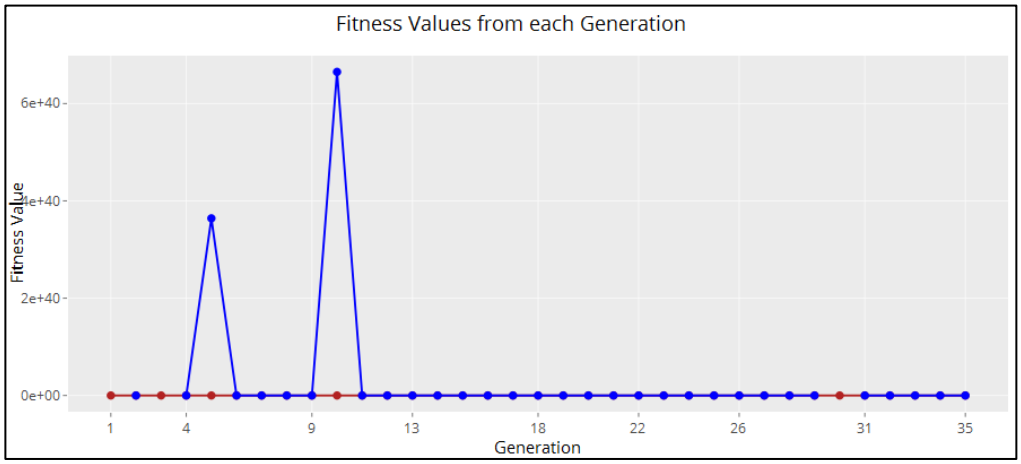


Figure 9-1: A plot of the best fitness and average fitness over generations

I tried to instead plot this using two separate graphs, but the average fitness plot was difficult to read, again due to the large range of values. I reconsidered if this feature was useful before spending any more valuable time on it and the choice was made that this was not a required feature. Creating plots to show results is a nice feature to have and was identified as "Could" in the functional requirement FR3-3 and this requirement has been satisfied by plotting the best fitness. Also, being able to visualise how the average fitness changes over time does not show any interesting details of how the program is operating since it only takes one small change to massively increase this average. Perhaps this could be more useful if the program used a different mutation rate as evolution progresses to help visualise how the program focusses on mutating smaller parts of the solution, so the fitness is closer to the best found so far.

9.7 Releasing Software

As outlined in my 6th aim, I aim to release the software on CRAN so that other R users can freely access the package I have produced. Unfortunately, this was not achieved but the package is still available through GitHub. The addition of a unit test for the main function of the program "`cgp`" is the only obstacle preventing from the package being released. For a package to be released, all unit tests must pass, and this unit test fails when run on CRAN but passes when run locally. This is due to the working directory changing when working on CRAN and trying to access a file that it can no longer file. More research will have done in how to solve this problem before the software can be released. I will still discuss the process of preparing the package for release as this taught me a lot about R packages.

For an R package to be released there are many checks performed on the structure of the package and all of these must pass. The "`devtools`" package also helps with this process through the "`check`" function. One such check is "`checking for dependencies in the R code`" which is checking that any outside functions used by the package are defined in the "`NAMESPACE`" file and the packages containing these functions are added to the "`DESCRIPTION`" file. This is so that when another user installs my package, the

dependencies are also installed making it easier for the user to setup. This check was returning "`no visible global function definition`" since the outside functions could not be found. My first strategy to solve this was to add calls to the "`library`" function in the functions where outside packages are used. However, this did not solve the issue and the "`library`" calls were also flagged to be removed. It is possible to solve this issue by writing the "`NAMESPACE`" and "`DESCRIPTION`" files manually but as discussed in Section 6.2, I was using a workflow which automated the process of documentation. This meant research was required into how to let these automated methods find the imported functions. The solution was to add the "`@importFrom`" tag into the roxygen skeleton for the functions which use any outside functions and specify the name of the package used, and any functions used from it. "`roxygen2`" makes sure that any functions marked with the tag are added to the "`NAMESPACE`" and "`DESCRIPTION`" files. This solved this issue and there no more complaints of global variables or functions being used.

In a roxygen skeleton, it is possible to provide examples on how the function is used and I done this for every function in my package since it is important that users understand how they are used. When running the checks so that the package could be released there is a check "`checking examples`" which runs each of the examples defined to make sure they are valid code. This was crashing on any of the functions which were not exported from the package. This was because an example is not required for a function which is not being exported since users of the package cannot directly call these functions. The only solution was to remove the examples used above functions which were not exported.

10 Evaluation

To evaluate this project, I have performed a usability study that has highlighted some usability issues with the guide that I have produced. This study will be covered in detail. I will also discuss the correctness of my software and cover the process of writing unit tests using the "`testthat`" package. Finally, I will assess the performance of the package by discussing both the speed of the implementation, and its ability to find solutions for one of the datasets distributed with the package.

10.1 Usability Study

A small user study was performed with test subjects using a guide I created containing instructions on how to use the package. The study was explained to test subjects and any questions they had were answered, they were then asked to fill in a consent form. The consent form can be found in Appendix E. Test subjects were asked to complete a survey after working through the guide to give their opinion on how they found the guide. The survey was created using Google Forms. All results from the surveys are contained in Appendix G. The interesting results from the survey will be discussed and any usability issues identified will be addressed. The conclusions of the study will be discussed at the end of the section.

10.1.1 caRtesian – Getting Started (User Guide)

I created a user guide which is to be distributed along with package I have developed. This guide is important as it is the mechanism for teaching users how to use the basic functionality of the package. The guide aims to give a quick overview of Cartesian Genetic Programming and for users to be able to use the package without any previous knowledge of the area. Users who have experience with Evolutionary Algorithms will potentially pick up the commands quicker since they use terminology they will be

familiar with. It is assumed that readers of this guide will have some previous experience using the R language, but it is not a requirement. The guide can be found in Appendix F.

The guide, "caRtesian – Getting Started" contains commands for users to run and an explanation as to what each command accomplishes. Through running each of these commands, the user will be able to set up the required variables, run the main function of the program, and to view the results that it produces. Installation instructions are also included.

As previously mentioned, the usability study I carried out has highlighted some issues with the user guide and these will be improved in a future version.

10.1.2 Background of Test Subjects

Before the participants of the test performed any tasks, I asked them to fill in a pre-test questionnaire. This was to gauge the participants backgrounds, so statements could be made saying that participants from a certain background had more issues with the guide than others. The questions asked the participants experience using R, their experience with Data Mining, their experience with Evolutionary Algorithms, and finally how regularly they program. If the subject said they had experience with Evolutionary Algorithms, further questions were asked on their experience with Genetic Algorithms and Genetic Programming.

I was not able to get any subjects with any reasonable experience in R however two subjects had previously used it. It is important that I conduct another study in future with subjects who have more experience with R as it is important that the package has a similar interface to it as the popular data mining packages. Regarding having previous experience with data mining, just over a third of subjects had previous experience even though it was just for a small project. Similarly, just over a third had previously used Evolutionary Algorithms with one subject using them regularly. Subjects were asked how regularly they program and a range of responses was given here. Most respondents said they program every day, or very often, with others saying they rarely program.

10.1.3 Getting Started Section

The Getting Started section of the guide covers the absolute basics of the package, so it is essential that the subjects were able to complete this section easily. The results gathered from the questions containing in the Getting Started section of the survey are encouraging and show that users were able to get through the guide with ease as shown in Figure 10-1. Only one user responded with a three. This user had no previous experience using R or with Evolutionary Algorithms, so it is possible that the lack of knowledge in Evolutionary Algorithms can make the guide harder to understand.

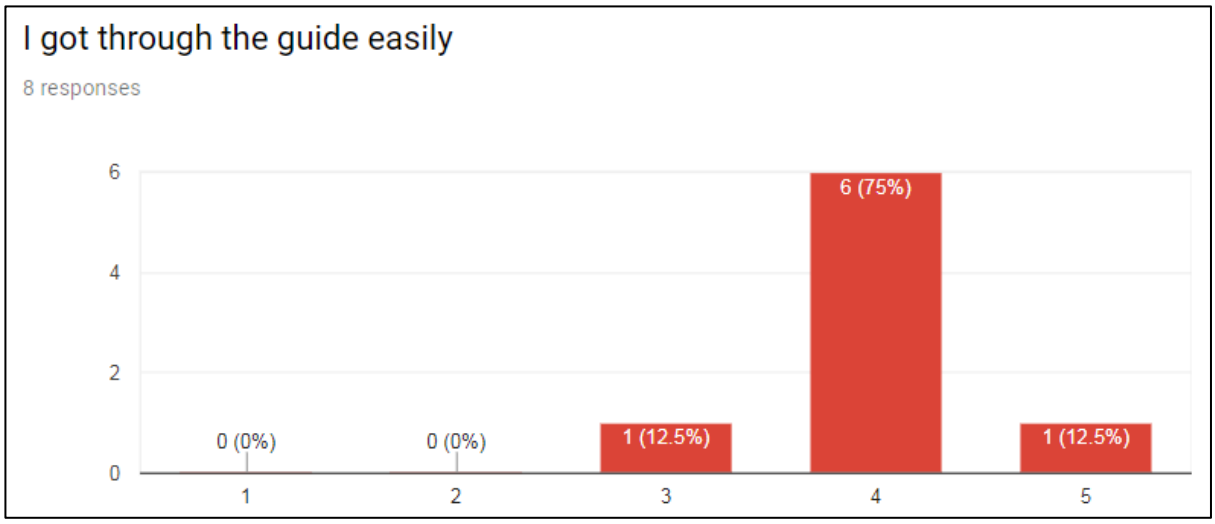


Figure 10-1: Responses from asking subjects if they got the guide easily

The next few questions asked if the subject was able carry out the steps easily and this also produced promising results with one subject responding with three again for these questions. A question with more interesting results asked if the subject felt the commands the subject they had to write were too long which can be found in Figure 10-2. This gave very mixed results with a 25% of respondents saying the commands were too long. The only command that could be considered lengthy in this section of the guide is the command to run the main function of the program. This will be addressed by either shortening the variable names but still allowing them to have self-descriptive names, or by providing more default values so that not as many parameters need to be set.

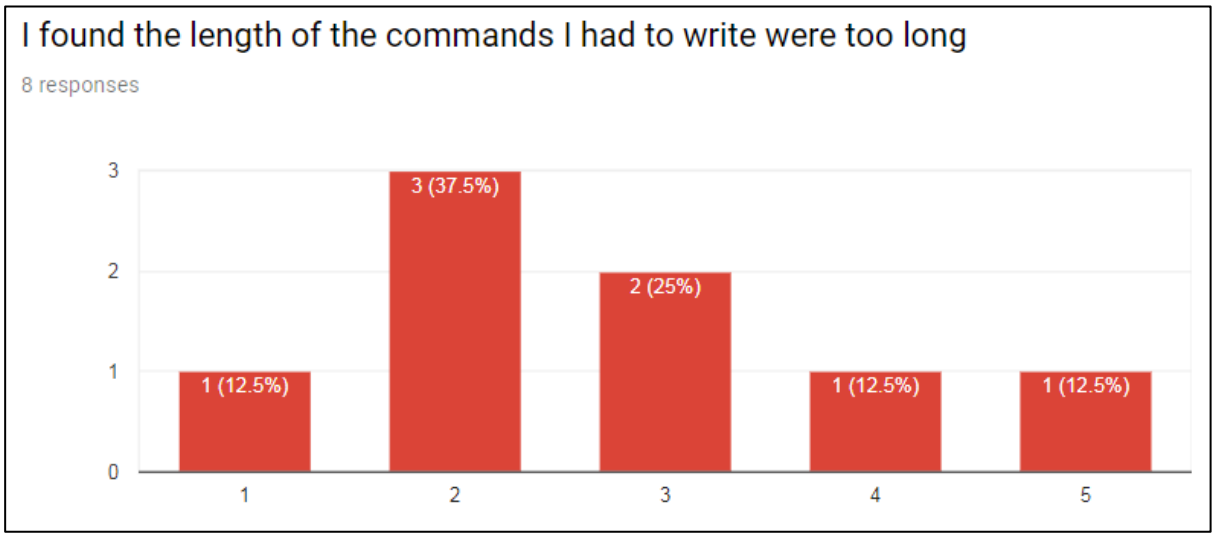


Figure 10-2: Responses from asking subjects if they felt the commands were too long

In the question regarding if the user understood the information displayed while the program was running, two people responded no before going on to say they did not understand both the fitness of the best solution so far and the average fitness of the population. Both users had no previous experience in Evolutionary Algorithms. While the guide I created is not designed to explain the terminology used in Evolutionary Algorithms it seems it would be beneficial for some users if short descriptions were added so they could at least refer to them if they did not understand something. All users said they found the graphs useful and were able to read the results from the graphs accurately. While this is a less important feature, it is still good to see positive feedback for this.

10.1.4 Changing the Parameters Section

This section of the guide was designed for more advanced users of the package, but I decided it was important to evaluate regardless of the user's experience. If the feedback given was positive then most likely the feedback will be positive for more experienced users but again, this would to be evaluated.

The tasks the user is asked to carry out in this section are to change the more advanced parameters of the program such as the fitness function and the selection method. The results from these questions highlighted some major issues with usability. During the tests, I found users were struggling to understand how to change the fitness function. I

explained to them that this is a hidden parameter of the main function. Once this was explained, they understood exactly how to carry out the task. Although this is written in the guide, it seems it would be beneficial to make this clearer perhaps by showing the functions signature or by giving the command directly as I have with the steps in the previous section of the guide. A similar task is then given to change the selection method. The response from this was more positive than the question regarding the fitness function even though this command is more difficult syntactically and much longer in length. The combined responses can be seen below in Figure 10-3. The low score for the fitness function task was most likely due to subjects requiring some advice on how to perform this task, but then understood that changing the selection method was done in a similar way. In hindsight, the order of these questions should have been swapped for each test to give a more balanced result. Nonetheless, the usability issue was identified and will be addressed.

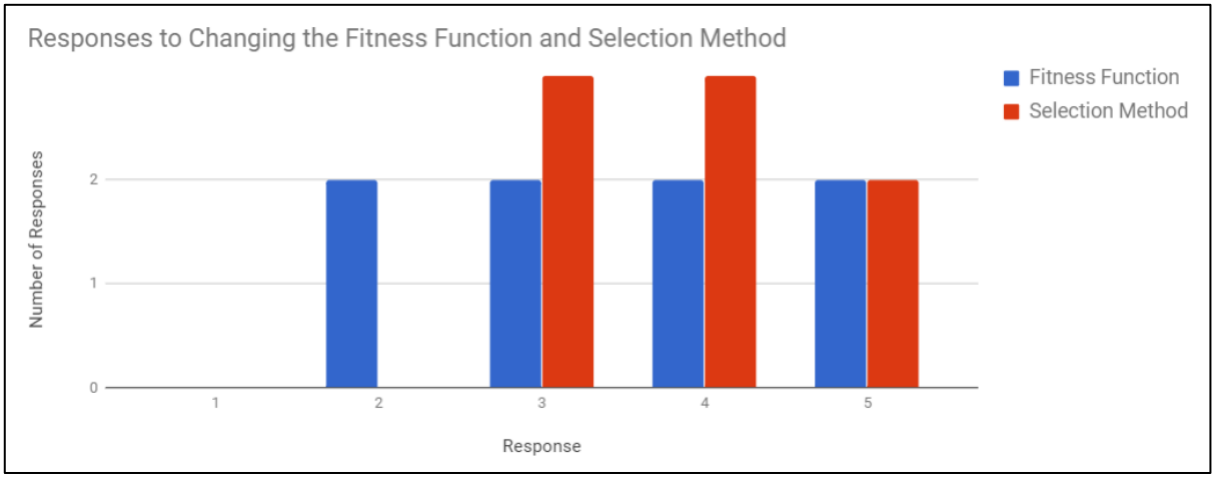


Figure 10-3: Combined responses from asking the users how easy the found changing the Fitness Function and Selection Method

The length of the commands is an area I have been worried subjects would have issues with. Subjects felt that the command to change the fitness function was not too long which means that this command can stay as it currently is. However, some users felt the command to change the selection method was indeed too long. I myself find this command too long so I can understand why users feel this way. This command is something I will assess and look for a solution to in a future release of the guide.

10.1.5 Using the Solution with New Data

This section covers tasks regarding the operations a user can perform to use a solution returned by the program with any future data. The overall response was that subjects were able to complete this section easily but felt that the commands were too long to load the data and to get a result.

10.1.6 Overall Opinion

At the end of the survey, test subjects were asked to answer some questions on their opinion of the software and guide overall. Every user left a response saying they found the software very interesting although they did not necessarily understand what it was doing. Again, this is due to the subjects lacking knowledge of the field of Genetic Programming, but it is not required to know how it works in order to use the software, so this is actually a very good response as it shows the commands encapsulate enough of the complex parts into high level commands that are usable.

Most users left a comment on their opinion of the guide overall. Every comment contained some feedback on the "Changing the Parameters" section with feedback such as "An example could be added to the fitness function section ...", and "It needs to be clearer that changing the fitness function and selection method are parameters to a function". As mentioned previously, these issues will be addressed as it should be straightforward to make these steps clearer by adding as suggested, an example to the fitness function section.

One user left a response saying, "commands could be encapsulated a little bit better". This user felt most of the commands were too long, so the length of the commands will also be addressed and where possible wrapper functions will be implemented to reduce the length of the commands and further abstract what the commands are doing.

10.1.7 Conclusions

Even though the test subjects were not particularly the target audience of the package, the overall response of the guide and software was very positive, but usability issues

were highlighted. Test subjects got through the guide easily, but most felt the length of the commands used were too long, so these will be addressed by providing wrapper functions to reduce the length and abstract further what the commands are doing. The terminology used in the guide was also found to be confusing for users without knowledge of Evolutionary Algorithms but as described in Section 10.3, this guide is not designed to teach the terminology used in this field. However, less jargon will be used in a future version of the guide and definitions of any terms used will also be included. The major issues found by the study were that changing the fitness function and selection method was more confusing than it should have been. This was due to these parameters being hidden parameters and users were not shown this. To address this, the signature of the function will be included in the guide. an example of how the command used in the "Getting Started" section changes to include these hidden parameters will also be included.

10.2 Correctness

I made extensive use of the "`testthat`" package throughout development as running tests was a core part of my workflow. Using this package, I added unit tests for the functions that I created. This was important as it helped to ensure that my program was working in the way I expected. Using unit tests are also a good way of detecting any behavioural changes that occur in my program caused by later changes. This was important when restructuring some existing functions as the tests acted as a safety net catching anything unintended. Unit tests also serve as a form of documentation in an R package and are distributed within the package.

10.2.1 Approach to Unit Testing

Where possible I followed Test-First Programming (Madeyski, 2010) which involves defining tests first that the code about to be written should pass. Using this method, it encouraged me to think more about the structure of the function I was about to write, and I feel that the functions written using this method are more concise and well-structured.

However, as I this was my first project where I have used unit testing, and my first large project in R, I found it difficult at some points to know what a function was expected to do until I had written it. Where this was the case, I would aim to implement as little of the function as required and then write a unit test to capture this behaviour.

10.2.2 Unit Testing using testthat

Unit testing in "`testthat`" aims to make testing as painless as possible (Wickham, 2011). It provides a visual display of which tests have passed or failed so it is straightforward to get an indication of where in your code something is not working as intended.

Test files are R scripts composed of a hierarchical structure. This structure is made up of expectations, tests and contexts. A "`context`" is used to group multiple tests into related functionality. In my package I have grouped tests into a file for each R source file and set the "`context`" as the file name. The next part is the "`test_that`" statement which groups related expectations into one function and further helps with tracking down any failed tests. The last is the expectations or "`expect_that`" statements. An example of this structure can be seen below in Figure 10-4.

```
context("funcSet")

test_that("constructFuncSet returns a data frame of the correct
structure", {

  dummyFunctions <- c(c("add", 2),
                      c("subtract", 2),
                      c("sqrt", 1))

  maxColumns <- 2

  funcSet <- constructFuncSet(dummyFunctions)
  expect_equal(ncol(funcSet), maxColumns)
  expect_equal(nrow(funcSet), length(dummyFunctions) / maxColumns)

  expect_is(funcSet, "data.frame")
  expect_is(funcSet$funcName, "character")
  expect_is(funcSet$arity, "integer")
})
```

Figure 10-4: Example of testthat test structure

The "expect_that" statements I mentioned previously are shown here as "expect_equal" and "expect_is" as this is less verbose than the "expect_that" version. For example, the "expect_is(x, y)" is actually "expect_that(x, is.a(y))". I used the less verbose option as I believe it does not reduce readability of each of the tests and saves time from remembering the syntax of many different expectation statements. These statements as well as the "test_that" statement read as sentences. For example, the "test_that" statement reads as "test that constructFuncSet returns a data frame of the correct structure", and one of the above "expect_is" statement reads as "expect funcSet is a data.frame".

Using the testing workflow I mentioned in Section 6.2.2 the tests for my package produce the output shown in Figure 10-5. The contexts are shown on the right side with the time taken to run each of the tests inside, and the outcome of each test is displayed on the left. In this case all tests have passed so are marked as "OK".

```
> devtools::test()
Loading caRtesian
Testing caRtesian
√ | OK F W S | Context
√ |  8       | dataHandler
√ | 22       | fitness [0.3 s]
√ |  6       | funcSet
√ |  6       | main [1.2 s]
√ | 54       | population [0.6 s]
√ |  6       | selection [0.2 s]
√ | 15       | utility [0.2 s]

== Results =====================================================================
Duration: 2.7 s

OK:       117
Failed:   0
Warnings: 0
Skipped:  0
```

Figure 10-5: Output from using devtools::test() to run the tests in the caRtesian package

For completeness I will also show what is displayed when a test is failed. This is not a real case as I have edited the test in Figure 10-4 to purposely fail but it shows the

informative output of "testthat". This can be seen in Figure 10-6. The context that contains the failed test is marked with a red "x" and an entry is put into the "F" column. This shows that a test has failed. The line number of the failed test, and a description of the failing test and error received is then displayed below the context. This is what allowed me to quickly identify where problems were and quickly address them. The total number of passing tests, failed tests, tests that have raised warnings, and skipped tests is also summarised at the end.

```
> devtools::test()
Loading caRtesian
Testing caRtesian
√ | OK F W S | Context
√ |  8       | dataHandler [0.1 s]
√ | 22       | fitness [0.4 s]
x |  5 1     | funcSet
-------------------------------------------------------------------------
test-funcSet.R:18: failure: constructFuncSet returns a data frame of
the correct structure
funcSet$arity inherits from `integer` not `numeric`.
-------------------------------------------------------------------------
√ |  6       | main [1.2 s]
√ | 54       | population [0.6 s]
√ |  6       | selection [0.2 s]
√ | 15       | utility [0.3 s]

== Results =============================================================
Duration: 3.0 s

OK:       116
Failed:   1
Warnings: 0
Skipped:  0
```

Figure 10-6: Output from using devtools::test() when a test fails in the caRtesian package

10.2.3 Unit Tests highlighting need for Refactoring

When creating the functions of the package which create the initial population and populate it with the correct structure of nodes I found that I was using a lot of parameters to each of the functions. This felt like bad practice and felt like my code was tightly coupled as some of these parameters were unrelated to what the function was intended to do. My original solution to this issue was to use global variables. When R is

evaluating a variable, if it is not defined in the current environment, it will then investigate the global environment. This also felt like bad practice as it was not clear by simply looking at a function what it was intended to do as there were variables being referenced from outside the scope of the function. However, I chose to continue with this approach.

When the time came to implement unit tests it was clear this approach was a bigger problem than having extra parameters to functions. Wherever a global variable was used, I needed to also define this global variable within each context that required it. An example is shown below in Figure 10-7.

```
cgp <- function(-PARAMETERS OMMITTED-) {

  #Double arrow to assign the variable into the global environment
  inputSize <<- 2

  ...


  initPopulation(popsize = 5)
}

initPopulation <- function(popsize) {

  ...

  #Using a variable defined outside of this functions scope
  generateInputs(inputSize)

  ...
}

library(caRtesian)
context("population")

#Must also define this global variable again here
inputSize <<- 2

test_that("initPopulation returns the correct number of solutions", {
```

Figure 10-7: Structure of code and associated test when using global variables

This example looks different from the code distributed in the final version as this was early in the development process. The parameters to the "`cgp`" function have been omitted as they are not important for this example. Also, where appropriate code has been missed out or marked with "…" to shows steps take place before the line shown. The example shows a variable defined globally in the "`cgp`" function called "`inputSize`". It then shows a call to the "`initPopulation`" function but does not pass the "`inputSize`" as a parameter. The "`inputSize`" is used within this function when calling "`generateInputs`" even though it is defined out of this function's scope. So far, this global variable has been assigned once. The global variable must also be initialised before any tests that use this variable can be run since tests are run under a different environment to the package. This means this same variable has been assigned twice and must be assigned in every test file that requires it. In this case it is not clear why "`inputSize`" is being defined in the first place since "`initPopulation`" does not use it as a parameter as previously mentioned.

Without adding unit tests to my software, I may not have removed the use of global variables and thus the quality of my code would have suffered in that readability would be reduced for outside users.

10.3 Performance of Package

10.3.1 Speed of Implementation

To assess the speed of the implementation I have used the "`profvis`" R package (Chang & Luraschi, 2018). This package extends R's built in line profiler by providing an interactive graphical interface for visualising the data gathered during profiling. A line profiler measures the time it takes to run each line of code and can be used to identify areas of the program that take up the majority of the running time. Results are displayed in both a graphical display as shown in Figure 10-8 as well as two textual approaches. The first is to show the source code if it is available and display the time taken for each line to complete on each line. This approach interacts directly with the

graph and vice versa. Highlighting either a section of the graph or the code itself also highlights the corresponding entry in the other. The other approach is to provide a tree-based display of times taken with higher times displayed at the top of the tree. This approach also sums the time a function is used into a total instead of showing every time a function is called like the graph does. I find that the tree approach is easier to use when analysing a specific function, but the graph-based approach is also very useful as it gives a quick overview of the entire function being profiled. In the examples that follow which contain screenshots of the lines of code, the lines with negligible execution time and the memory managed by each line have both been hidden. This is because I am mainly concerned with the speed of this implementation.

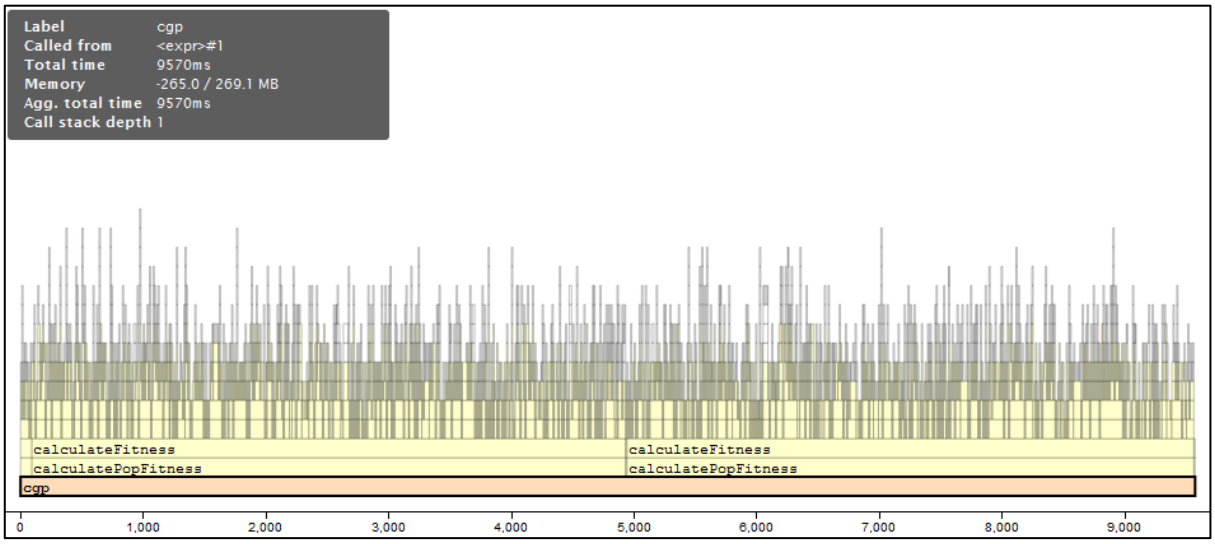


Figure 10-8: Visualisation of profiling of cgp function

The chart shown in Figure 10-8 shows the function calls inside the program against time. At the base is the "`cgp`" function which takes up the entire time. This is due to this being the function that was being profiled. The row above the "`cgp`" function shows the functions which are called from within this function. The small square on the very left is the creation of the initial population which takes a very small amount of time when compared to the amount of time taken to calculate the fitness of the population. Although it cannot be seen in this visualisation, there is a call to the selection method directly between the "`calculatePopFitness`" function calls. This cannot be seen as it takes a miniscule amount of time to run. Finally, the solution is printed after the fitness

of the population is calculated the second time. This also takes a negligible amount of time. From the scale shown at the bottom of the chart, it can be identified that calculating the fitness of the population takes up roughly 98% of the run time of the program. Calculating the fitness of the population is commonly a resource demanding task in Evolutionary Algorithms but a closer look must be taken to identify why this implementation runs so slowly. Since the time taken by "`calculatePopFitness`" is entirely composed of "`calculateFitness`", this is the function which will be looked at.

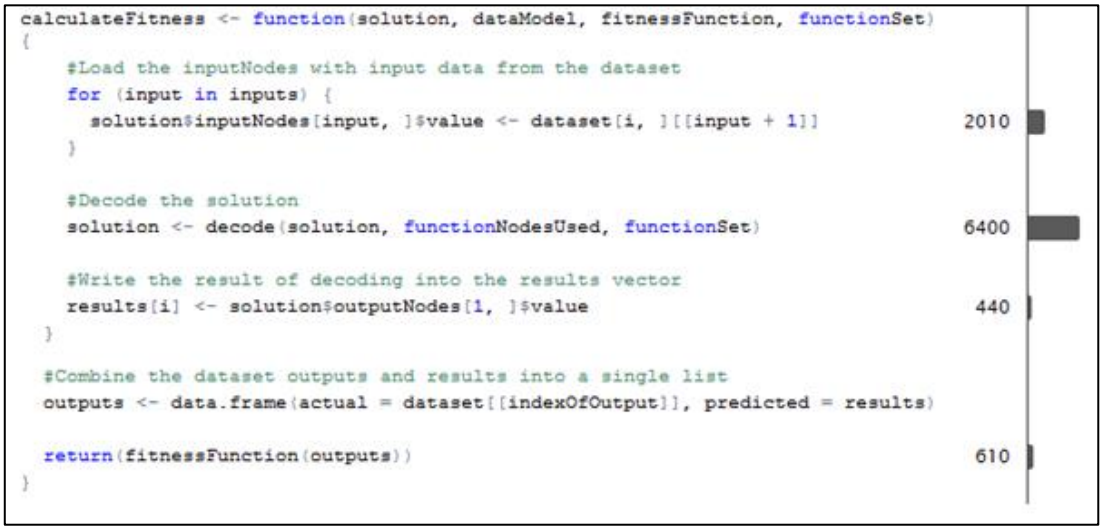


Figure 10-9: Profiling of calculateFitness

Upon further inspection using the graph-based approach it can be seen that there are four time consuming tasks performed by this function. These are loading the solution with data, decoding the solution, writing the results of decoding into a list, and finally calculating the fitness value of the solution. The total time taken for each of these steps is shown in Figure 10-9. The graphics on the right of each line displays the time taken in milliseconds. Since each of these lines repeats for every row in the dataset, the times taken will be drastically different depending on the length of the dataset used. The dataset used in this case was a 1000 row dataset. The profiling shows that the final step of calculating the fitness of the solution is not as time consuming as I originally thought it would be. The more time consuming parts are loading the solution with data, and the decoding of the solution with these inputs to get a value. It is not clear how the step of loading the data could be made more efficient. Perhaps if I removed some of the nesting

used here so that every time the "for" section is run, it loads these values into a temporary variable and this temporary variable could replace the entire "value" column inside the input nodes. This should be less time consuming than the current implementation if only by a small amount. The decoding has the majority of time used which makes sense since this is a demanding task.

Looking into the reasons for decoding using so much computational time, I found interesting results. The "decode" function calls "calculateValueInSolution" which in turn calls "calculateValue" and most of the time used by "calculateValueInSolution" is taken up by storing values into data frames. It is not clear why this is an issue since it is not nested in a for loop as with the "calculateFitness" function. More research will have to be carried out in order to understand why this uses up so much time as the performance of R is not something I have previous experience with. The remainder of the is used by "calculateValue" so this function will now be looked at.

```
calculateValue <- function(node, solution, functionSet) {

  #Get the name of the function to call from the functionSet
  funcToCall <- functionSet[node$funcID, ]$funcName                640

  inputs <- unlist(node$inputs[[1]])                               170

  #Get the value of the first argument of the funcToCall
  firstArgument <- findRow(solution, inputs[1])$value              1090

  #If the function takes two parameters
  if (length(inputs) == 2) {                                        10
    #Get the value of the second argument of the funcToCall
    secondArgument <- findRow(solution, inputs[2])$value            70
  } else {
    #The function takes one parameter
    value <- do.call(funcToCall, list(firstArgument))               60
  }

}
```

Figure 10-10: Profiling of calculateValue

Most of the time used by "calculateValue" is using the "findRow" function which can be seen above in Figure 10-10. This function finds a node within the solution when it is unknown which data frame the node is located in. This function is used for convenience since the operation of finding a node when its location is unknown is a common

operation in my program. There is not much that can be done to avoid the computational time involved with this function. One solution would be to store the ranges of unique identifiers of each type of node in the program and directly access these wherever required instead of using this function.

Overall the speed of the package is very slow, but it is still unclear what makes some of the steps discussed so time consuming. As mentioned, further research must be required out before an answer can be given as to why the package performs so poorly. R is a slow language, but I feel the implementation must be able to run quicker than this. If there is not much improvement after improving the solution still using solely R code, it will be investigated how to embed C code within R scripts as this will greatly increase performance. The reason for investigating the use of R code first is I feel I will learn more about the language and be able to avoid these problems in future even in small projects.

10.3.2 Ability to find Solutions

Experiments were carried out using one of the datasets distributed with the package, namely the dataset contained in "`x_squared_minus_y.csv`". This was to check that the package is able to find solutions to a sample problem. Although this sample problem is significantly simpler than a real world problem, the results gathered are promising in showing that the package does indeed manage to find solutions.

The sae dataset used in each experiment and the maximum generations and levels back parameters remained constant. The structure of the function nodes was changed from a 5x5 grid, to a larger 10x10 grid, and finally to a 5x10 grid. Three experiments were run using each of these parameters. Due to the stochastic nature of the program, the results gathered here are not reproducible and could show a completely different outcome if carried out again. To make a confident conclusion on how these parameters affect the programs ability to find solutions, many more experiments would have to be run, and the results would require to be averaged. Nonetheless my findings will be discussed since this was my first experience of experimenting with the program.

The experiments with each of the parameters all managed to find solutions were perfect meaning the answer the solution computed for every input was correct. Although these solutions were correct, they were not the exact function that the dataset represents which is "`output = x * x + y`". An example of one of the solutions that was found was "`ceiling(((a * a) - b))`". This managed to correctly identify most of the expression but then applies "`ceiling`" to the result which will round the result up. This is not a problem for this case since each of the inputs used in the dataset were integers meaning there will be no rounding done by functions such as "`ceiling`" but if this solution was to be used with new values which were real valued, then the result would be incorrect. This issue will only be picked up on sample datasets since the function to find is known. With all the experiments carried out either a solution was found, or the best solution found would have a fitness of 51.159. Each of the experiments which did not found the correct solution managed to find the "`x * x`" part of the expression but did not find the "- y". It is possible that this would have been found if the experiments were set to run for more generations. The success rate of the program to find solutions was 55% and as said previously the correct result was almost found in the other experiments.

These experiments were also biased in that I knew what the function be found was and the structure of the function nodes was restricted to small sizes mainly due to the reasons discussed relating to the speed of the program. It was difficult to test the program with any larger of a grid than 10x10 since it took a few hours to run each experiment. Once the issues relating to performance are addressed more experiments will be run to fully assess the program I have produced.

One interesting area that the experiments highlighted was that many generations would pass without any improvement at all in the best solution found. This is maybe due to the simplicity of the function to be found. More experiments would have to be carried out with functions of more complex length and structure before concluding this is the reason. Another aspect I investigated was the function set used and two of the functions

that could be used had no effect in this scenario since it was integer based. These functions were "`ceiling`" and "`floor`". I decided to look into a recommended set of functions to include and found that the tree-based Genetic Programming package "rgp" (Flasch et al., 2014) does not use any rounding functions, the same set of functions is also discussed in Cartesian Genetic Programming (Miller, 2011).

11 Reflection

In this section I will reflect on what I have achieved through this project. I will cover what requirements outlined in Section 7.1 were met. I will also evaluate which of the initially aims listed in Section 1.2 have been completed. Finally, I will cover how I feel my skills using R have developed through the project as this was a key part of this project.

11.1 What I Achieved vs Requirements

The requirements outlined in Section 7.1 changed during design. These requirements were initially gathered from what I found to be important features of other implementations of Cartesian GP. I decided that some requirements were marked with an incorrect priority, so the priorities were re-assessed and updated. I also changed the direction I wanted to take the package so some of the requirements were no longer related to what I wanted to produce.

Each requirement is marked with a colour: green for completed, yellow for in progress, and red for incomplete. These can be found in Appendix H. Important requirements will be discussed in the appropriate sections below.

11.1.1 Completed Requirements

Most requirements were completed with some of the lower priority requirements also being completed. This is due to these being easier to implement than I originally thought. One such of example of this is FR1-5 and FR1-9. These were completed in the same manner. Since the main function expects a data frame and a model describing that data frame to be passed into the program as parameters, files can also be passed in by also passing the "`read.csv`" function in. The requirement FR3-4 was also completed

since the priority of this was changed to "Should" from "Could" after I decided that I wanted the package to be usable as a data mining tool.

Some of the requirements relating to the customisation of the package were also completed such as FR2-5, FR2-6, FR2-12 and FR2-13. These have all been achieved through adding parameters to the programs main function.

The requirements relating to the more critical aspects of Cartesian Genetic Programming were also completed. FR1-1 and FR1-2 are the most important requirements of the package since these are requirements stating that the package shall implement CGP and shall run evolution. One of the non-functional requirements, NFR1-2, was also completed. This is a large performance boost to the program even though it remains very slow.

11.1.2 In-Progress Requirements

The most important requirement which is still in progress is FR2-7 which states that the package shall allow users to choose a mutation method from the included choices. This was not completed due to the mutation method provided with the package being difficult to implement and using more time than I assumed this would take. The structure of this functionality of the program is also not designed to be parameterised but can be completed with the same method as is done with the selection method although there are some usability issues regarding this.

FR2-10 is marked as in progress since this could be completed to a much higher standard. While it is possible for users to define their own selection methods, this will be quite difficult and possibly confusing for users. To fully complete this requirement the usability issues involving changing the selection method are required to be solved first, then helper functions will be added guiding the user on how to get their function to the stage that it can be used. A unit testing framework could also be developed for this functionality to further help users understand what is expected by a selection method, but this requirement was marked as "Could" meaning it is a nice feature to have but not essential by any means.

11.1.3 Incomplete Requirements

The requirements which were not completed range over different topics. Two of the requirements, FR1-3 and FR1-4, are features I thought would be interesting for the package to have but I knew these would be difficult to implement and were far from essential, so these were not completed.

FR2-11 is the most important requirement which has not been completed. This was not completed due to being related to other requirements which are marked as in-progress, namely FR2-7 and FR2-10. Issues surrounding these requirements must be solved before this requirement can be addressed.

Finally, NFR1-1 specified that the computationally intensive parts of the package will be written in faster languages such as C or Java. While this would have greatly benefitted the speed of the package, this was not completed due to the project being a personal project. I decided that I would learn more by implementing the entire project directly in R and this ties into one aim 3 which is to learn the fundamentals of R.

11.2 What I Achieved vs Initial Aims

Most of the aims I set out to achieve were gated by previous aims. This meant that there was a clear order that these were required to be completed in.

11.2.1 Completed Aims

The two aims were regarding the availability of bio-inspired algorithms in R and identifying areas which could be improved or those which lacked any freely available implementations. Both aims were completed successfully. Section 3 which covers both of these aims could have been more in depth into exactly what each of the packages provide but this was not necessary in this case since an entire field was poorly represented which was Genetic Programming. I chose to develop a package to provide access to this area.

The third aim was to learn the fundamentals of the R language. This aim is purely opinion based but I would say this has been completed to a high standard. There are

more areas of R which I want to delve into such as object-oriented programming and how to improve the performance of R code, but I consider these more advanced topics and this aim was only to learn the fundamentals.

Aims 4 and 5 are similar with 4 being to produce an R package to improve the availability of bio-inspired algorithms in R and 5 being to implement Cartesian Genetic Programming. Aim 4 could been completed in different ways by providing a package to different areas but in this case was completed in the same manner as the fifth aim. Sections 7, 8 and 9 covers how these aims were completed. There is also a small section in 10.3.2 which discusses how well the package performs at finding solutions with the sample datasets.

The package was evaluated for usability issues by performing a small usability study as outlined in Section 10.1. This ties in to the seventh aim. This study revealed some major usability issues and solutions to these issues have been discussed in Section 10.1.7. The main issue was that some commands asked the test subject to change the values of some hidden parameters, but users did not understand how to do this without guidance.

Evaluating what the package provides and what could be added to it is discussed in the previous section. This is done by comparing the initial requirements to what I have been able to produce. Further extensions are also discussed in Section 12. Assessing the performance of the package has also been assessed in Section 10.3.1 with issues regarding performance discussed and possible improvements also mentioned. These topics which make up the eighth aim and this has also been completed successfully.

11.2.2 Incomplete Aims

The sixth aim is the only aim I have been unable to complete. This was to release the package on CRAN. Although the package has been made public through GitHub, this does not satisfy the criteria. The reasons this was not completed, and the issues related are discussed in Section 9.7

11.2.3 Overall Discussion

Since seven of the eight initial aims were achieved, I conclude this project was a success. Although there are a lot of areas of the package which can be improved or extended, the version I have produced does as is required. The evaluation of the package regarding its ability to find solutions is an area I would like to spend more researching once the package's performance has been greatly improved.

11.3 How R skills have Developed

Through this project I feel my R skills have developed massively. Since the style of R programming I have used in this project is functional, I also feel I have developed the problem-solving skills required in functional programming as they are very different to the object-oriented methods I would normally use. I can say that these have developed due issues that I came across during early development being a lot easier to solve by the end. One such example of this is discussed in Section 8.3.2 and 9.2 where I discuss the issues involved in finding the valid inputs to a given function node. I feel the reason I had so many difficulties solving this issue originally was not because I was new to R, but new to functional programming.

While my skills in R can still be developed in many different areas such as object-oriented R, I feel I now have a solid foundation in the language.

12 Conclusions

This project is the product of continuous research, planning and development over the period of two semesters at university. The primary aim was to produce a package which filled a hole in the provision of bio-inspired algorithm in R. Since this, and most other aims being were completed, I would consider this project a great success.

This project has not only taught me the fundamentals of R but with it being a software development project, I learned a useful skill in unit testing. Automated testing was a completely new topic to me at the start of this project and I dread to think of the difficulty of this project if I did not have this framework in place. Although my approach to testing can be improved, I would also say my first experience with this was a success and I will look into how to also automate testing in other languages.

The project ends with a package known as "`caRtesian`" containing a working implementation of Cartesian Genetic Programming in native R which supports customisation of the parameters used within evolution. The package can be improved vastly in many areas, but it provides a starting point for anyone wishing to now further extend this implementation or to take it into a different direction. I have thoroughly enjoyed producing this package, working through the difficulties involved and I look forward further developing this software.

13 Further Work

The package produced is a working version but there are many areas that can be extended. These areas are but not limited to: the functionality of the package, the usability of the package, and the performance of the package.

13.1 Functionality

While a lot of the planned functionality has been achieved as discussed in Section 11.1, there are still some important areas so be completed. Completing these before finding additional functionality to add is the route I am going to take. To identify additional functionality, more publicly available implementations of CGP will be investigated for features that would be useful if included in the package I have created. Another user study could also be performed with users from a background in Genetic Programming to get feedback on any features they feel are missing.

The crossover operator mentioned in Section 2.1 and 2.5 has not been implemented in this package. The use of crossover may improve the ability of the package to find solutions and to help explore the sample space of solutions better, but further research is required before it can be decided if this is a worthwhile investment.

The recursive version of decoding discussed in Section 8.4 is still used in the package when the user wants to decode a solution with new input values after the program has finished. This was due to this function taking less parameters than the iterative version but again this version will recalculate nodes values which is incorrect. The recursive version will be removed in a future release and the iterative version will be simpler to use.

One feature I think would be useful to add to this package is the ability to visualise the resulting graph. The implementation discussed in Section 5.2 uses GraphViz (Graphviz.org, n.d) to display the graphs it creates. Perhaps bindings could be written to

this software or there may be a package directly available in R since the visualisation tools in R are highly used.

As Cartesian Genetic Programming is a still evolving field, is it likely new advancements will be discovered which will greatly improve the performance of CGP. It would be beneficial for these also to be implemented into this package.

13.2 Usability

As covered in Section 10.1, the package has some usability issues. These issues are addressed, and solutions are discussed but a further study is required to ensure that these issues are indeed solved. Again, as mentioned in section 10.1, the target-audience of the package were not part of the test subjects used to evaluate the package, so it is important that a study is carried out with the appropriate audience. It is likely that more issues will be highlighted here.

13.3 Performance

The performance of the package is the area I would describe as the most important to be improved upon. The package currently takes an extremely long time to run as shown in Section 10.3.1 with the exact areas highlighted which use up the majority of the computational time. The most straightforward way to improve the performance of these areas is to embed C code to perform these tasks. This should see a massive increase in performance. The line profiler used in Section 10.3.1 cannot profile embedded code so a different solution must be used to gauge the performance increase.

The memory efficiency of the package has not been assessed in this study but is also possible using the "`profvis`" package.

14 References

Adamatzky, A. (2010). Game of life Cellular Automata. London: Springer, pp.19-25.

Bendtsen, C. (2012). pso: Particle Swarm Optimization.

Bergmeir, C. and Benitez, J. (2012). Neural Networks in R Using the Stuttgart Neural Network Simulator: RSNNS. Journal of Statistical Software, [online] 46(7), pp.1-26. Available at: http://www.jstatsoft.org/v46/i07/ [Accessed 12 Nov. 2017].

Ciupke, K. (2016). psoptim: Particle Swarm Optimization.

Chang, W., Cheng, J., Allaire, J., Xie, Y. and McPherson, J. (2017). shiny: Web Application Framework for R. [online] cran.r-project.org. Available at: https://cran.r-project.org/web/packages/shiny/index.html [Accessed 01 Apr. 2018]

Cho, H., Kim, D., Olivera, F. and Guikema, S. (2011). Enhanced speciation in particle swarm optimization for multi-modal problems. European Journal of Operational Research, 213(1), pp.15-23.

Cran.r-project.org. (2018). R Internals. [online] Available at: https://cran.r-project.org/doc/manuals/r-release/R-ints.html [Accessed 7 Apr. 2018].

Craven, M. and Shavlik, J. (1997). Using neural networks for data mining. Future Generation Computer Systems, [online] 13(2-3), pp.211-229. Available at: http://www.sciencedirect.com.ezproxy1.hw.ac.uk/science/article/pii/S0167739X97000228 [Accessed 3 Nov. 2017].

Darwin, C. & Wallace, J., 1998. The origin of species / Charles Darwin., Ware: Wordsworth Editions.

Dorigo, M. and Gambardella, L. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary

Computation, [online] 1(1), pp.53- 66. Available at: http://ieeexplore.ieee.org.ezproxy1.hw.ac.uk/document/585892/ [Accessed 21 Oct. 2017].

Eberhart, R. and Kennedy, J. (1995). A new optimizer using particle swarm theory. MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science. [online] Available at: http://ieeexplore.ieee.org.ezproxy1.hw.ac.uk/document/494215/ [Accessed 5 Nov. 2017].

Flasch, O., Mersmann, O., Bartz-Beielstein, T., Stork, J. and Zaefferer, M. (2014). rgp: R genetic programming framework.

Francke, T. (n.d.). ppso - Particle Swarm Optimization and Dynamically Dimensioned Search, with parallel option.

Fritsch, S., Guenther, F., Suling, M. and Mueller, S. (2016). neuralnet: Training of Neural Networks.

Ghisu, T., Arca, B., Pellizzaro, G. and Duce, P. (2015). An Improved Cellular Automata for Wildfire Spread. Procedia Computer Science, [online] 51, pp.2287-2296. Available at: http://www.sciencedirect.com/science/article/pii/S1877050915011965 [Accessed 10 Nov. 2017].

Hester, J. (2017). lintr: A 'Linter' for R Code. [online] Cran.r-project.org. Available at: https://cran.r-project.org/web/packages/lintr/index.html [Accessed 23 Nov. 2017].

Chang, W., Luraschi, J. (2018). profvis: Interactive Visualizations for Profiling R Code. [online] Cran.r-project.org. Available at: https://cran.r-project.org/web/packages/profvis/index.html [Accessed 18 Apr. 2017].

Hughes, J. (2013). CellularAutomaton: One-Dimensional Cellular Automata.

Kar, A. (2016). Bio inspired computing – A review of algorithms and scope of applications. Expert Systems with Applications, [online] 59, pp.20-32. Available at: https://www-sciencedirect-

com.ezproxy1.hw.ac.uk/science/article/pii/S095741741630183X [Accessed 29 Oct. 2017].

Khan, A. and Baig, A. (2015). Multi-objective feature subset selection using mRMR based enhanced ant colony optimization algorithm (mRMR-EACO). Journal of Experimental & Theoretical Artificial Intelligence, [online] 28(6), pp.1061-1073. Available at: http://www.tandfonline.com.ezproxy1.hw.ac.uk/doi/abs/10.1080/0952813X.2015.1056240 [Accessed 4 Nov. 2017].

Khushaba, R., Al-Ani, A., AlSukker, A. and Al-Jumaily, A. (2008). A Combined Ant Colony and Differential Evolution Feature Selection Algorithm. Ant Colony Optimization and Swarm Intelligence, [online] 5217, pp.1-12. Available at: https://link-springer-com.ezproxy1.hw.ac.uk/chapter/10.1007/978-3-540-87527-7_1 [Accessed 4 Nov. 2017].

Koza, J. (1992). Genetic programming: on the programming of computers by means on natural selection. MIT Press.

Kramer, O. (2017). Genetic Algorithm Essentials. Studies in Computational Intelligence, [online] 679, pp.11-18. Available at: https://doi-org.ezproxy1.hw.ac.uk/10.1007/978-3-319-52156-5 [Accessed 29 Oct. 2017].

Madeyski, L. (2010). Test-Driven Development: An Empirical Evaluation of Agile Practice. Springer-Verlag New York Inc.

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics, [online] 5(4), pp.115-133. Available at: https://link.springer.com/article/10.1007/BF02478259 [Accessed 3 Nov. 2017].

Miller, J. (1999). An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming Approach. In: Proceedings of the 1st Genetic and Evolutionary Computation Conference. Morgan Kaufmann, pp.1135-1142.

Miller, J. (2011). Cartesian Genetic Programming. Natural Computing Series. [online] Available at: https://link-springer-com.ezproxy1.hw.ac.uk/book/10.1007%2F978-3-642-17310-3 [Accessed 13 Nov. 2017].

Miller, J. and Smith, S. (2006). Redundancy and computational efficiency in Cartesian genetic programming. IEEE Transactions on Evolutionary Computation, [online] 10(2), pp.167-174. Available at: https://www.researchgate.net/publication/3418872_Redundancy_and_computational_efficiency_in_Cartesian_genetic_programming [Accessed 19 Nov. 2017].

Moon, C., Kim, J., Choi, G. and Seo, Y. (2002). An efficient genetic algorithm for the traveling salesman problem with precedence constraints. European Journal of Operational Research, 140(3), pp.606-617.

Navid, A. and Bagheri, A. (2013). Cellular Learning Automata and Its Applications. [online] InTech. Available at: https://www.intechopen.com/books/emerging-applications-of-cellular-automata/cellular-learning-automata-and-its-applications [Accessed 31 Oct. 2017].

Oranchak, D. (2009). Cartesian Genetic Programming for the Java Evolutionary Computing Toolkit (CGP for ECJ). [online] Oranchak.com. Available at: http://www.oranchak.com/cgp/doc/ [Accessed 24 Nov. 2017].

Pedroni, E. (2013). JCGP · Equal Parts. [online] Equalparts.eu. Available at: https://equalparts.eu/projects/jcgp/ [Accessed 21 Nov. 2017].

Poli, R., Langdon, W., McPhee, N. and Koza, J. (2008). A field guide to genetic programming. [S.l.]: Lulu Press.

Quast, B.A. (2016). rnn: a Recurrent Neural Network in R

RStudio. (2016). RStudio - Open source and enterprise-ready professional software for R. [online] Available at: https://www.rstudio.com/ [Accessed 23 Nov. 2017].

Russell, S. and Norvig, P. (2009). Artificial intelligence: A Modern Approach. 3rd ed. Pearson, pp.749-753.

Saka, M., Doğan, E. and Aydogdu, I. (2013). Analysis of Swarm Intelligence–Based Algorithms for Constrained Optimization. Swarm Intelligence and Bio-Inspired

Computation, [online] pp.25-48. Available at: https://doi-org.ezproxy1.hw.ac.uk/10.1016/B978-0-12-405163-8.00002-8 [Accessed 5 Nov. 2017].

Sarkar, P. (2000). A brief history of cellular automata. ACM Computing Surveys, [online] 32(1), pp.80-107. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.4529&rep=rep1&type=pdf [Accessed 10 Nov. 2017].

Scrucca, L. (2013). GA: A Package for Genetic Algorithms in R. Journal of Statistical Software, [online] 53(4), pp.1-37. Available at: http://jstatsoft.org/v53/i04/ [Accessed 12 Nov. 2017].

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., Despouy, P. and Plotly Technologies Inc. (2017). plotly: Create Interactive Web Graphics via 'plotly.js'. [online] cran.r-project.org. Available at: https://cran.r-project.org/web/packages/plotly/index.html [Accessed 01 Apr. 2018]

Sivanandam, S. and Deepa, S. (2008). Introduction to genetic algorithms. Berlin: Springer, pp.14-39.

Tiobe.com. (2017). R | TIOBE - The Software Quality Company. [online] Available at: https://www.tiobe.com/tiobe-index/r/ [Accessed 27 Oct. 2017].

Turner, A. (2014). About - CGP-Library. [online] Cgplibrary.co.uk. Available at: http://www.cgplibrary.co.uk/files2/About-txt.html [Accessed 21 Nov. 2017].

Turner, A. and Miller, J. (2017). Recurrent Cartesian Genetic Programming of Artificial Neural Networks. Genetic Programming and Evolvable Machines, [online] 18(2), pp.185-212. Available at: https://link.springer.com/article/10.1007%2Fs10710-016-9276-6 [Accessed 13 Nov. 2017].

Venables, W. and Ripley, B. (2002). Modern Applied Statistics with S. 4th ed. New York: Springer.

Wickham, H. (2011). testthat: Get Started with Testing. The R Journal, [online] 3(1). Available at: https://journal.r-project.org/archive/2011/RJ-2011-002/RJ-2011-002.pdf [Accessed 4 Apr. 2018].

Wickham, H. (2014). Advanced R. [ebook] Boca Raton: Chapman & Hall. Available at: http://adv-r.had.co.nz/ [Accessed 10 Apr. 2018]

Wickham, H. (2015). R Packages. Cambridge: O'Reilly & Associates.

Wickham, H. (2016). testthat: Unit Testing for R. [online] cran.r-project.org. Available at: https://cran.r-project.org/web/packages/testthat/index.html [Accessed 23 Nov. 2017].

Wickham, H. and Chang, W. (2017). devtools: Tools to Make Developing R Packages Easier. [online] cran.r-project.org. Available at: https://cran.r-project.org/web/packages/devtools/index.html [Accessed 23 Nov. 2017].

Wickham, H. and Chang, W. (2016). ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics. [online] cran.r-project.org. Available at: https://cran.r-project.org/web/packages/ggplot2/index.html [Accessed 01 Apr. 2018].

Wickham, H., Danenberg, P. and Eugster, M. (2017). roxygen2: In-Line Documentation for R. [online] cran.r-project.org. Available at: https://cran.r-project.org/web/packages/roxygen2/index.html [Accessed 23 Nov. 2017].

Willighagen, E. and Ballings, M. (2015). genalg: R Based Genetic Algorithm.

Wolfram, S., 1983. Statistical mechanics of cellular automata. Reviews of Modern Physics, 55(3), pp.601–644.

Wolfram, S. (2002). A new kind of science. 1st ed. Champaign, IL: Wolfram Media.

Yadav, N., Yadav, A. and Kumar, M. (2015). An introduction to neural network methods for differential equations. Dordrecht: Springer, pp.29-54.

Yang, X. and Karamanoglu, M. (2013). Swarm Intelligence and Bio-Inspired Computation. Swarm Intelligence and Bio-Inspired Computation, pp.3-23.

Yang, X. (2014). Particle Swarm Optimization. Nature-Inspired Optimization Algorithms, [online] pp.99-110. Available at: http://www.sciencedirect.com.ezproxy1.hw.ac.uk/science/article/pii/B9780124167438000075 [Accessed 6 Nov. 2017].

Zhang, G. (2009). Neural Networks For Data Mining. Data Mining and Knowledge Discovery Handbook, [online] pp.419-444. Available at: https://link.springer.com/chapter/10.1007/978-0-387-09823-4_21 [Accessed 3 Nov. 2017].

15 Appendices

15.1 Appendix A – createFunctionNodesStructure

```
#' createFunctionNodesStructure
#'
#' Creates a data frame containing the structure of a function node. The data
#' frame is the length set in the generateFunctionNodes parameters.
#'
#' @param rowsRequired the number of rows to create
#'
#' @return the data frame created
#'
createFunctionNodesStructure <- function(rowsRequired) {

  #Create integer vectors containing rowsRequired NA values
  naColumn <- rep(as.integer(NA), rowsRequired)

  #Create a data frame with the length required to store the function nodes
  functionNodes <- data.frame(chromoID = naColumn,
                              value = as.numeric(naColumn),
                              funcID = naColumn)

  #Add a column to store a vector specifiying the input nodes
  functionNodes$inputs <- vector(mode = "list", length = nrow(functionNodes))

  return(functionNodes)
}
```

15.2 Appendix B – getValidInputs

```
#' getValidInputs
#'
#' Determines the valid range of input chromoIDs for a given chromoID
#'
#' @param chromoID the chromoID to calculate the range of
#' @param functionNodeRange all the chromoIDs contained in functionNodes
#' @param functionNodeStructure the parameters used to create functionNodes
#'
#' @return the valid chromoIDs
#'
getValidInputs <- function(chromoID, functionNodeRange,
functionNodeStructure) {

  #Put the functionNode chromoIDs into a matrix
  functionNodeMatrix <- matrix(functionNodeRange,
                               nrow = functionNodeStructure$rows,
                               ncol = functionNodeStructure$cols)

  #Find the column index containing chromoID
  column <- which(functionNodeMatrix == chromoID, arr.ind = TRUE)[[1, 2]]


  #Get the columns that are in the levelsBack range
  validColumns <- (column - functionNodeStructure$levelsBack):(column - 1)
  validColumns <- validColumns[validColumns >= 1]

  #Extract the chromoIDs from the validColumns
  validChromoIDs <- functionNodeMatrix[, validColumns]

  return(as.vector(validChromoIDs))
}
```

15.3 Appendix C – nodesToProcess

```
#' nodesToProcess
#'
#' Find the functionNodes which are required by the outputNodes.
#'
#' @param solution The solution containing the nodes
#'
#' @return the functionNodes required
#'
nodesToProcess <- function(solution) {

  functionNodes <- solution$functionNodes

  outputID <- solution$outputNodes[1, ]$inputs

  nodesUsed <- vector(mode = "logical", length = nrow(functionNodes))

  nodesUsed <- traverseFunctionNodes(functionNodes, nodesUsed, outputID)

  return(functionNodes[nodesUsed, ])
}
```

15.4 Appendix D – traverseFunctionNodes

```
#' traverseFunctionNodes
#'
#' Traverses through the functionNode structure starting at chromoID
#' and then recursively running on each of the nodes inputs.
#'
#' @param functionNodes the functionNode structure
#' @param nodesUsed a boolean vector signifying if a node was used
#' @param chromoID the chromoID of the starting node
#'
#' @return a boolean vector signifying the nodes used
#'
traverseFunctionNodes <- function(functionNodes, nodesUsed, chromoID) {

  #If the chromoID is now an inputNode
  if (chromoID < functionNodes[1, ]$chromoID) {

    return(nodesUsed)
  } else {

    #Find the index of the node with this chromoID
    index <- which(functionNodes$chromoID == chromoID)

    #Set the corresponding row in nodesUsed to TRUE
    nodesUsed[index] <- TRUE

    #Recursively loop over the inputs of each node used
    inputs <- unlist(functionNodes[index, ]$inputs)
    for (input in inputs) {
      nodesUsed <- traverseFunctionNodes(functionNodes, nodesUsed, input)
    }

    return(nodesUsed)
  }
}
```

15.5 Appendix E – Consent Form

caRtesian – Getting Started

Case Study

Consent to Act as a Subject in an Experimental Study

Principal Investigator: Ryan Porteous

Description: The purpose of this study is to assess the accessibility and usability of the software package I have produced. The package implements a type of Genetic Programming which will be used in this case to solve symbolic regression problems. The tasks you will be asked to carry out involve reading from a User Guide, entering the commands given into an R terminal, and observing the results.

There are minimal risks involved with taking part in this study. All information gathered will remain anonymous throughout the process.

You are free to decline in taking part in this study. Should you decide to participate, you are free to leave the study at any time.

Voluntary Consent: I certify that I have read the preceding statements and that I understand their contents. Any questions I have pertaining to the research have been and will be answered by the Principal Investigator. My signature below means that I have freely agreed to participate in this study, and that I agree to the publication of the results for scientific purposes, and to the distribution of the recordings and transcripts of the sessions for research purposes so long as my identity is not revealed.

Subject Signature _____ Date ___/___/___

Investigator's Certification: I certify that I have explained to the above individual the nature and purpose, the potential benefits, and possible risks associated with participation in this research study, have answered any questions that have been raised, and have witnessed the above signature.

Investigator Signature _____ Date ___/___/___

15.6 Appendix F – caRtesian – Getting Started (User Guide)

caRtesian – Getting Started

Author: Ryan Porteous
rp10@hw.ac.uk

The software package caRtesian provides an implementation of Cartesian Genetic Programming directly in R. The package is to be used in a data mining environment for symbolic regression problems. This guide aims to introduce the package and how to get started using it.

Introduction to Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a variety of Genetic Programming which uses graphs to represent programs instead of the traditionally used trees. It creates a population of solutions to a given problem and progressively improves these solutions through an evolutionary process.

The caRtesian R Package

The package is used to solve symbolic regression problems where a dataset is provided containing rows of input values and the desired output from these values, but the function providing the output value is unknown. The package provides generates candidate solutions for this function and assesses each of them with the desired output values in the dataset. The solutions progressively improve through a process called evolution as the program runs.

Installation

The package is available on GitHub and can be installed by using the devtools package and running the following command in an R session:

```
> devtools::install_github("porteous54/caRtesian")
```

As the package is installed, the dependencies are also installed so installation can take longer than expected if the dependencies are not already installed.

Getting Started

The package must be loaded into the running R session can be done through the command:

```
> library("caRtesian")
```

Loading Variables

The next step is to load the required variables into the Global Environment. The variables that are needed, are a `dataset` containing the input and output values, a `model` specifying the input and output fields of the dataset, and the set of functions that the program can sample from to build a solution called a `functionSet`.

For this example, the `dataset` will be `x_squared_minus_y` which contains two input values `x` and `y`, an output value: `output`. The output value is actually the result of the function $x^2 - y$ but the program does not know this and will try its best to find this function. The dataset can be loaded using the command:

```
> dataset <- x_squared_minus_y
```

The dataset structure and example values can be viewed using the command:

```
> head(dataset)
```

This shows the name of each column and shows the values contained in it. The next step is to define a model which describes this dataset. Since this dataset was created for the purpose of this guide, the `output` depends on both the `x` and `y` variables, but this is not always the case in a symbolic regression problem. The model can be defined through the command:

```
> model <- output ~ x + y
```

Finally, the set of functions must be loaded and in this case a set containing simple arithmetic operations will be used. This can be done through the command:

```
> functionSet <- arithmeticSet()
```

The functions contained in this set can be viewed by writing the name of the variable `functionSet` into the console:

```
> functionSet
  funcName arity
1        +     2
2        -     2
3        *     2
```

Run the Program

To begin running the program and start searching for solutions we use the `cgp` function and give the variables we have defined already as arguments to it as well as adding some others:

```
> set.seed(10)
> result <- cgp(dataset = dataset,
+               model = model,
+               functionSet = functionSet,
+               maxGenerations = 15,
+               rowsFuncNodes = 6,
+               colsFuncNodes = 2,
+               levelsBack = 1,
+               updateFreq = 3)
```

The `set.seed` function is used so that the results will be reproducible. Then the program is run which runs for a maximum of 15 generations but will finish before that if it finds a solution. The rest of the parameters are to define the shape of the graph used inside the program to represent the solution.

As the program runs, details of its progress are displayed in the console with the best solution found printed as a mathematical equation at the end of execution. An example of this can be seen here:

```
Generation: 15 / 15
Fitness of best solution so far: 52.159
Average fitness of population: 704.783

Best solution found as text:
(a * (c + a))
```

View the Results

Once the program is finished, the results are stored in the variable we defined called `result` which contains various information so that the solution found can be used now with future data. The progress information that is printed as the program runs is also stored in the result so that it can be viewed as a plot. The plot can be loaded through the command:

```
> plotGraph(result$plotData)
```

The graph shows the how the solution evolved over time and hovering over each of the points provides the exact value of the fitness and generation. There are tools attached such as zooming and being able to select a subset of points for a closer display of them.

In this case the program was unable to find the exact function that the dataset was formed but there are actually multiple parameters that can be tuned to suit the needs of the user. These parameters are discussed in the next section.

Changing the Parameters

The parameters used in the previous section produced a solution which was not correct but was the best found. It is possible that by tuning these parameters a perfect solution will be able to be found. The parameters set previously: `rowsFuncNodes`, `colsFuncNodes` and `levelsBack` affect the structure that it is possible for the solution to take so it may be the case for a certain dataset, these parameters simply cannot allow the solution to be represented fully.

There are also hidden parameters that have not been seen yet which are `fitnessFunction` and `selectionMethod`. These parameters have default values so that users less experienced with Genetic Programming can still get started in using the package.

Changing the Fitness Function

The `fitnessFunction` parameter specifies the function used to assess a solution against the desired output defined in the dataset. There are two fitness functions distributed with the package which are `mae` (Mean Absolute Error) and `rmse` (Root Mean Squared Error). The default fitness function is `mae`, but this can be changed to `rmse` through changing the `fitnessFunction` parameter to `rmse`. Parameterising this also allows any user of the package to define their own fitness function and also pass it into the program, as long as it accepts the same arguments as `mae` or `rmse` and returns a numeric value.

Changing the Selection Method

The `selectionMethod` parameter expects a list which provides the function to use for selection, and its associated arguments. The program by default uses the `muLambdaStrategy` which accepts three parameters so these must be defined in the list. An example of how `muLambdaStrategy` is used is shown below.

```
selectionMethod = list(func = muLambdaStrategy,
                       args = c(population = NA, 4, NA))
```

The vector `args` follows the same order that the `muLambdaStrategy` function accepts and dummy values are provided here for the `population` and the last parameters, `functionNodeStructure` since these change during execution of the program.

Another selection method called `tournamentSelectionStrategy` is also distributed with the package. To use this selection method instead, the `selectionMethod` must be changed but it still follows the same structure since `tournamentSelectionStrategy` accepts the same parameters.

```
selectionMethod = list(func = tournamentSelectionStrategy,
                       args = c(population = NA, 4, NA))
```

To help with structuring this parameter, there is a function `validSelectionInput` which checks the structure matches what the program expects.

Using the Solution with New Data

The solution is stored in the result of the program, along with the set of functions the program used to sample from when creating the solution. This is so that new data can be entered into the solution and a value can still be given. Data can be loaded into the new solution with the command:

```
> result$bestSolution$inputNodes$value[x] <- value
```

In the above command `value` is the value to be loaded and `x` is the row to load it into. Be careful not to overwrite the last row as this stores a random constant value used by the program.

An example which can be used with the result generated in the Getting Started section is:

```
> result$bestSolution$inputNodes$value[1] <- 5
> result$bestSolution$inputNodes$value[2] <- 10
```

To then get a value using the new loaded data use the command:

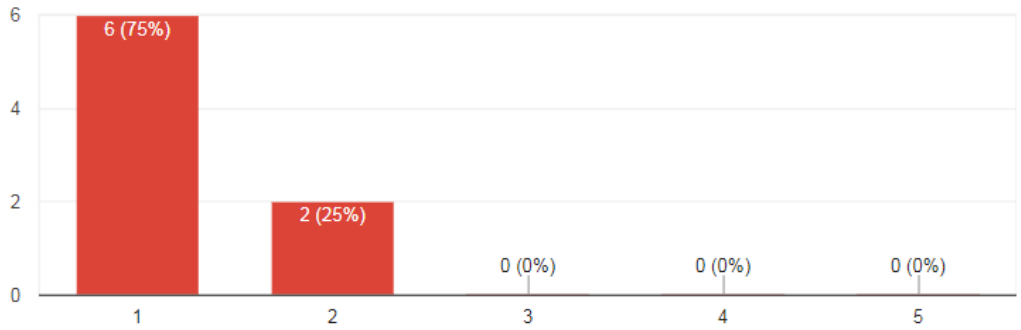
```
> decode2(result$bestSolution, result$functionSet)
```

15.7 Appendix G – Usability Study Results

Pre-Test Questionnaire

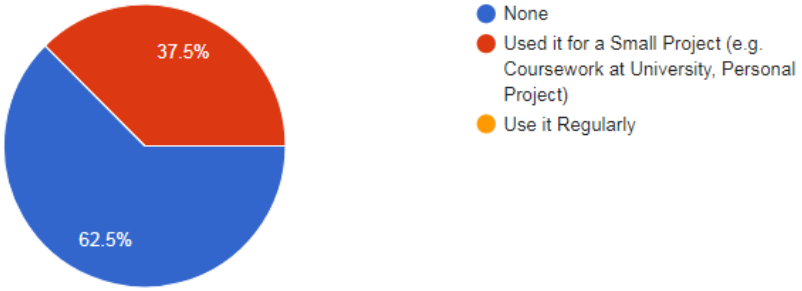
How much experience do you have in using the R language?

8 responses



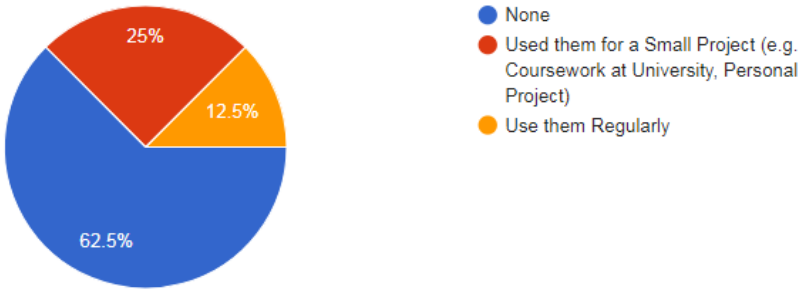
How much experience do you have with Data Mining?

8 responses



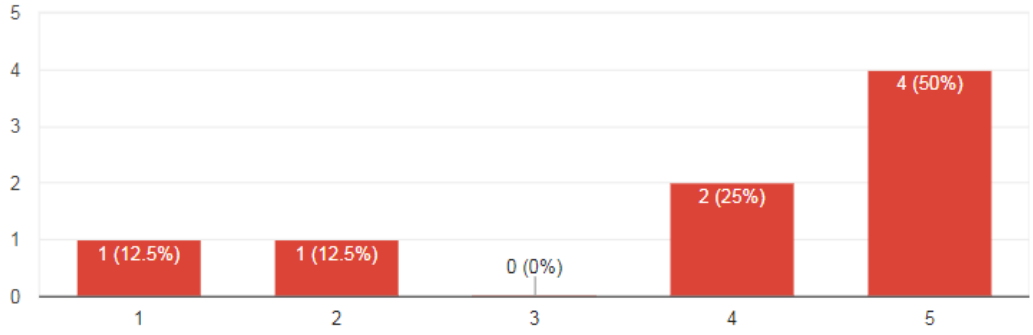
How much experience do you have with Evolutionary Algorithms?

8 responses



How regularly do you program?

8 responses



Evolutionary Algorithms

If you have experience with Genetic Algorithms please provide details of your experience with them.

3 responses

Implemented a Genetic Algorithm to solve the Travelling Salesman Problem as coursework at University
Learned about them at uni and used them in personal project
Wrote small programs. Done coursework at University.

If you have experience with Genetic Programming please provide details of your experience with it.

2 responses

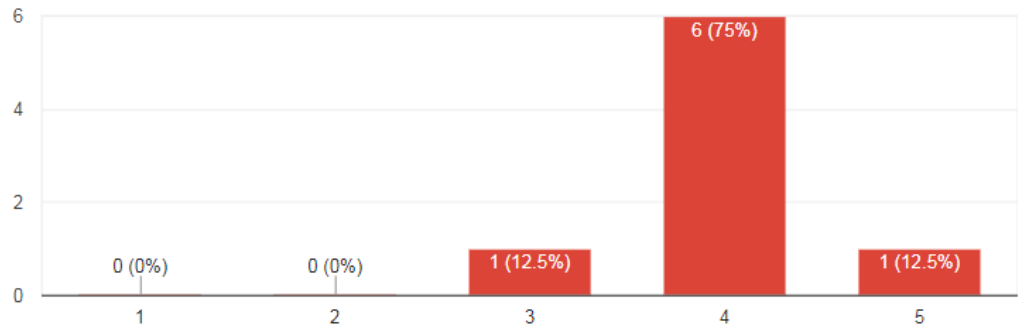
Experimented with a GP library to investigate how different parameters affect results

Wrote fairly large programs. Done coursework at University.

Getting Started

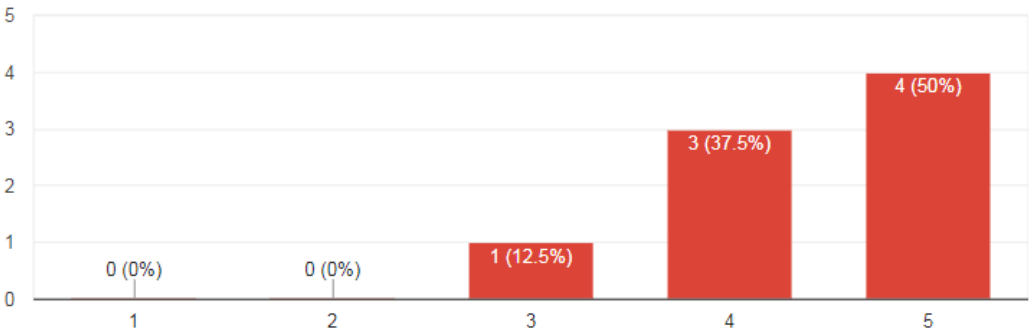
I got through the guide easily

8 responses



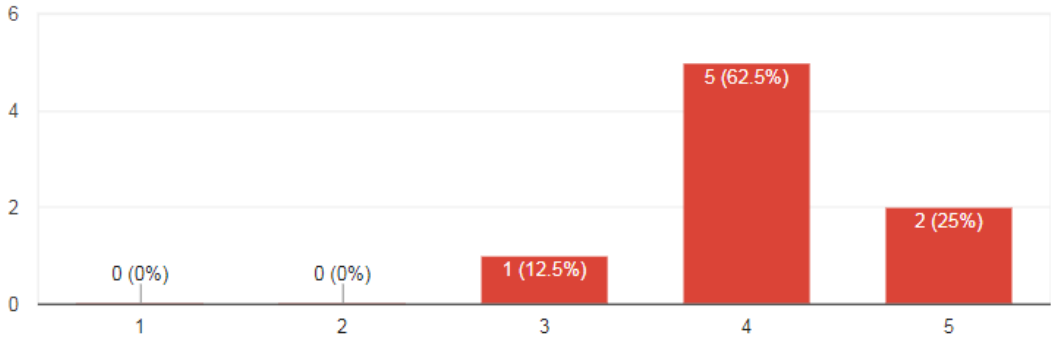
I was able to set up the required variables easily

8 responses



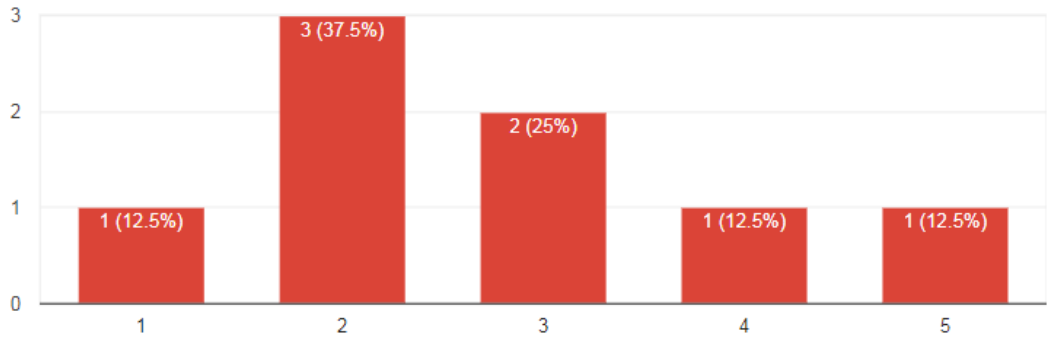
I was able to run the program easily

8 responses



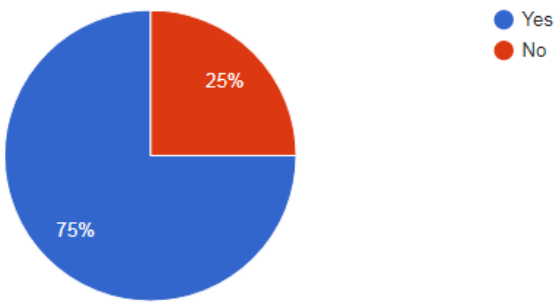
I found the length of the commands I had to write were too long

8 responses



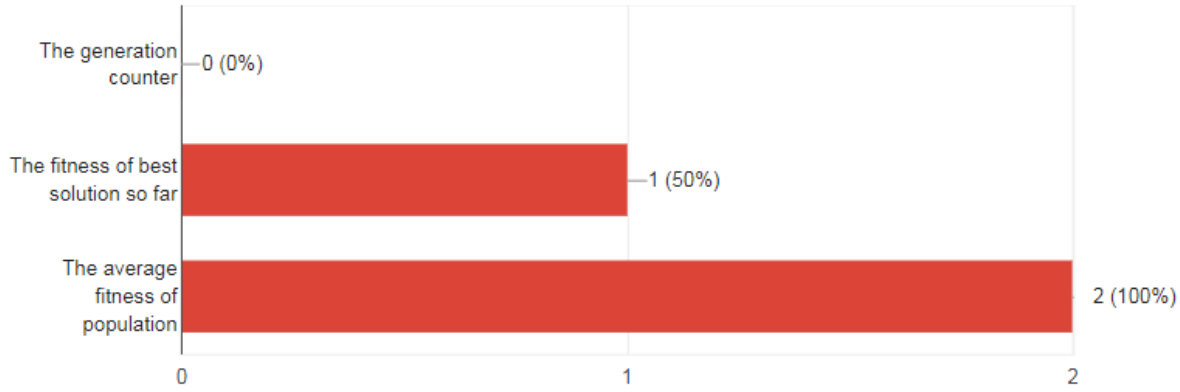
Did you understand what the information displayed while the program was running meant?

8 responses



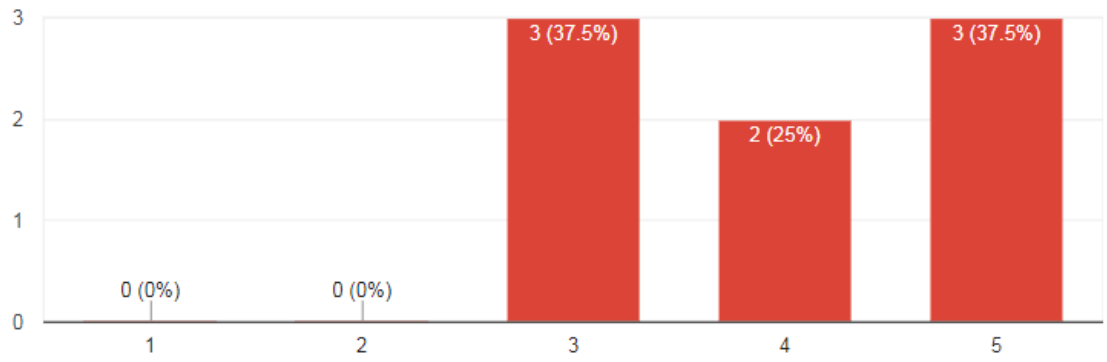
If you answered No to the previous question, what parts did you not understand?

2 responses



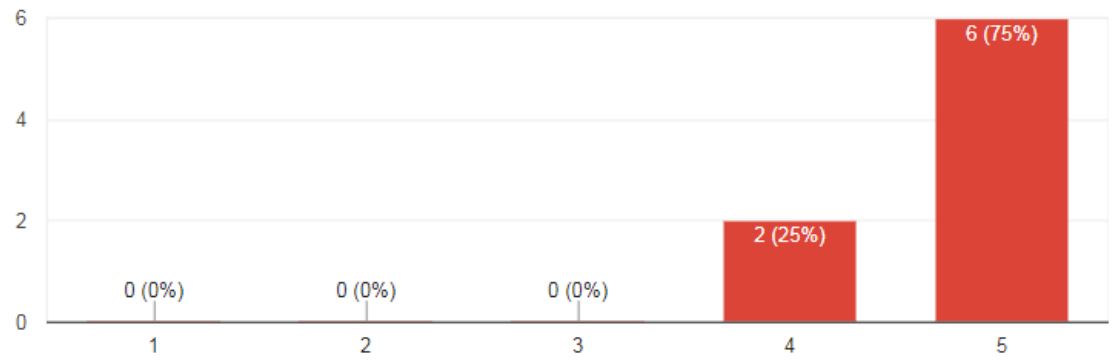
I found the text format of the solution useful

8 responses



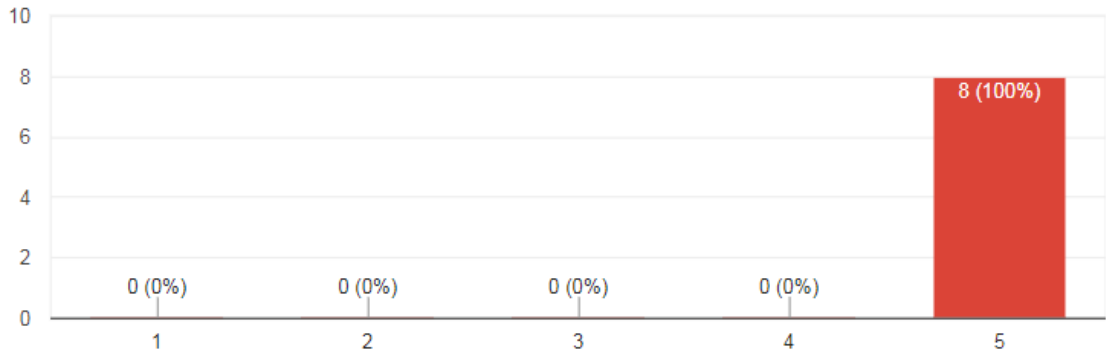
I found being able to view the results as a graph useful

8 responses



I found it easy to read the graph accurately

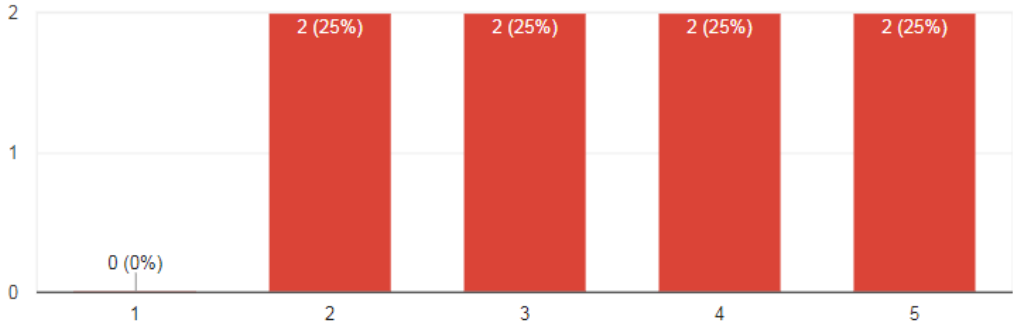
8 responses



Changing the Parameters

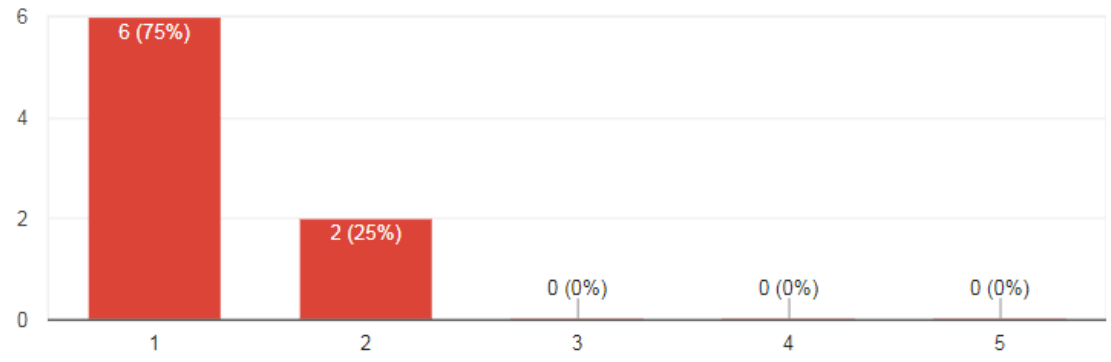
I was able to change the fitness function easily

8 responses



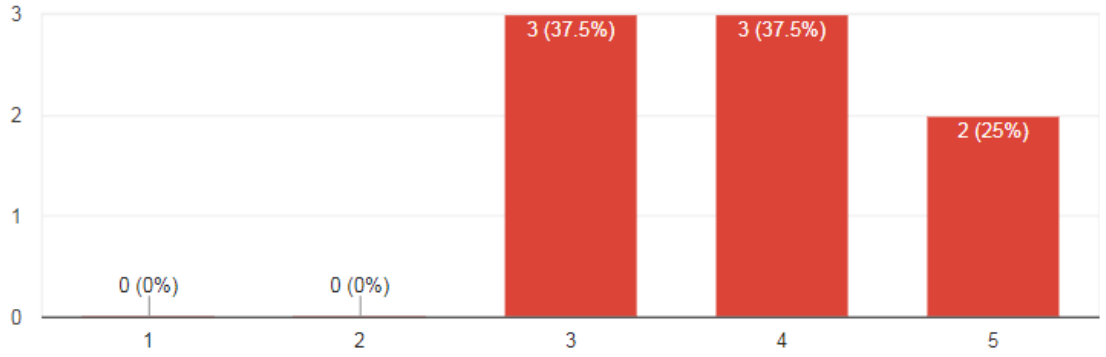
I felt the length of the command to change the fitness function was too long

8 responses



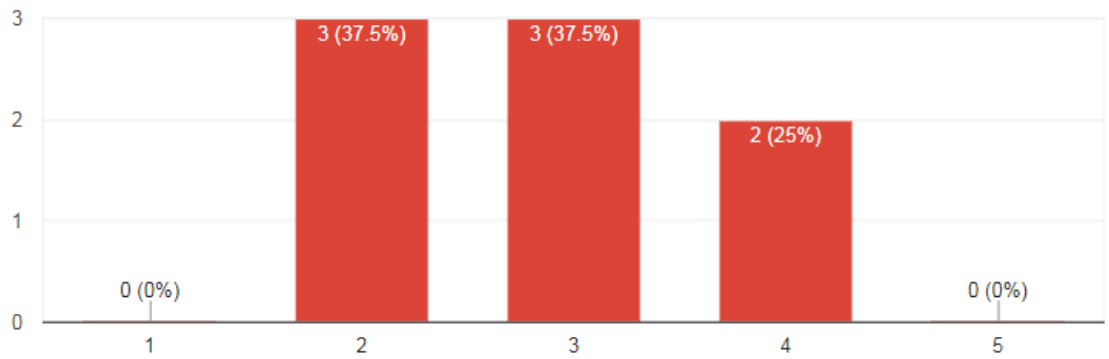
I was able to change the selection method easily

8 responses



I felt the length of this command to change the selection method was too long

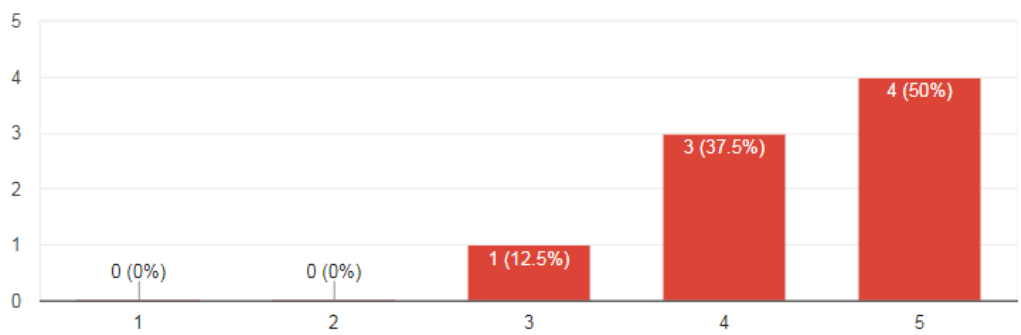
8 responses

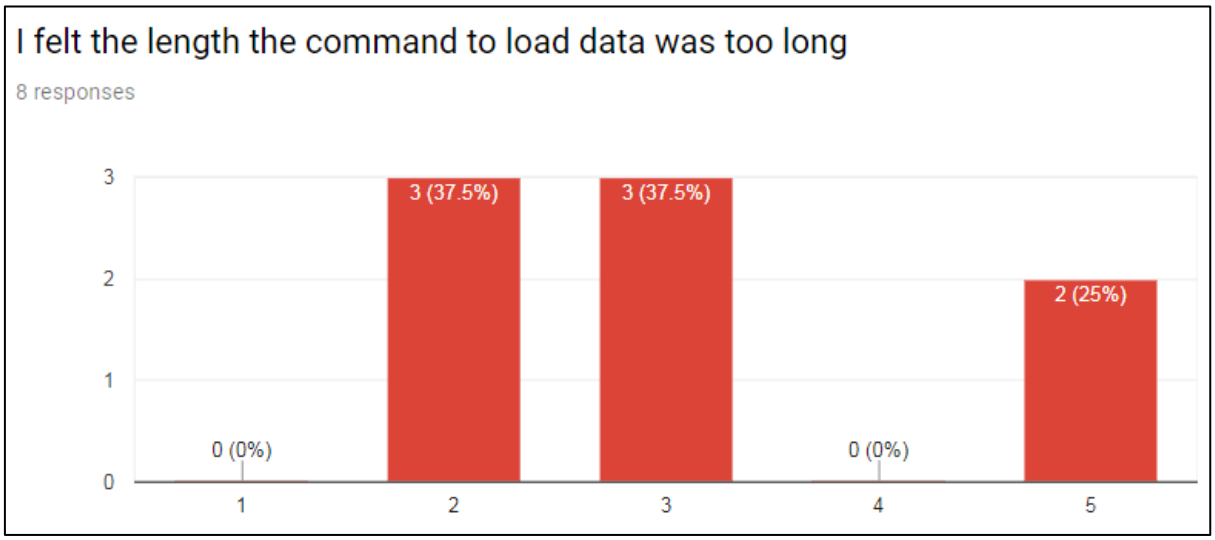


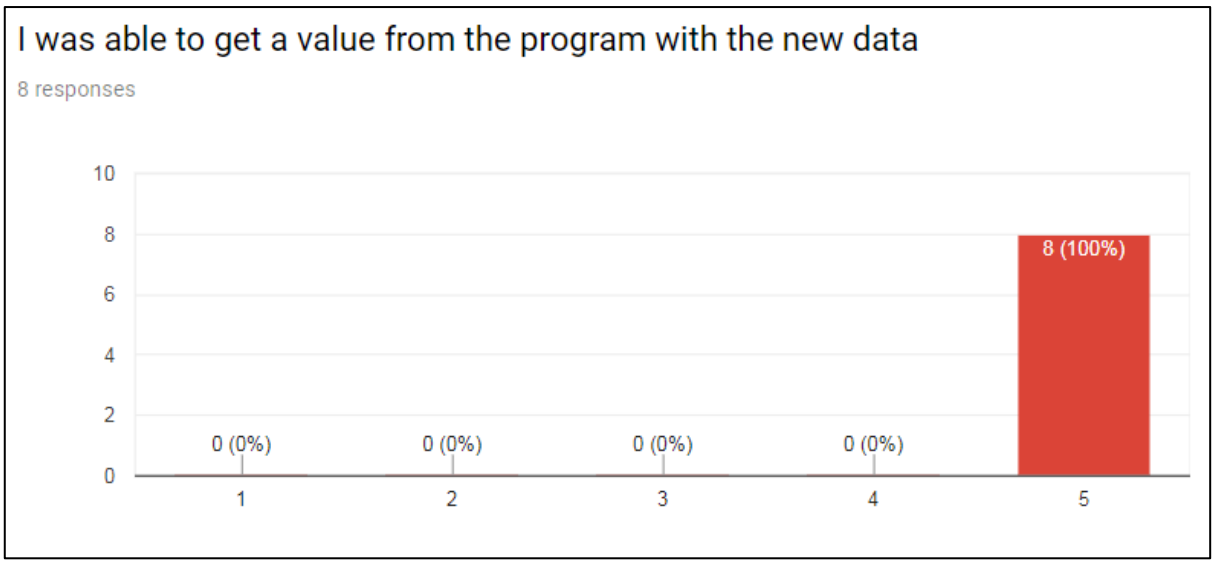
Using the Solution with New Data

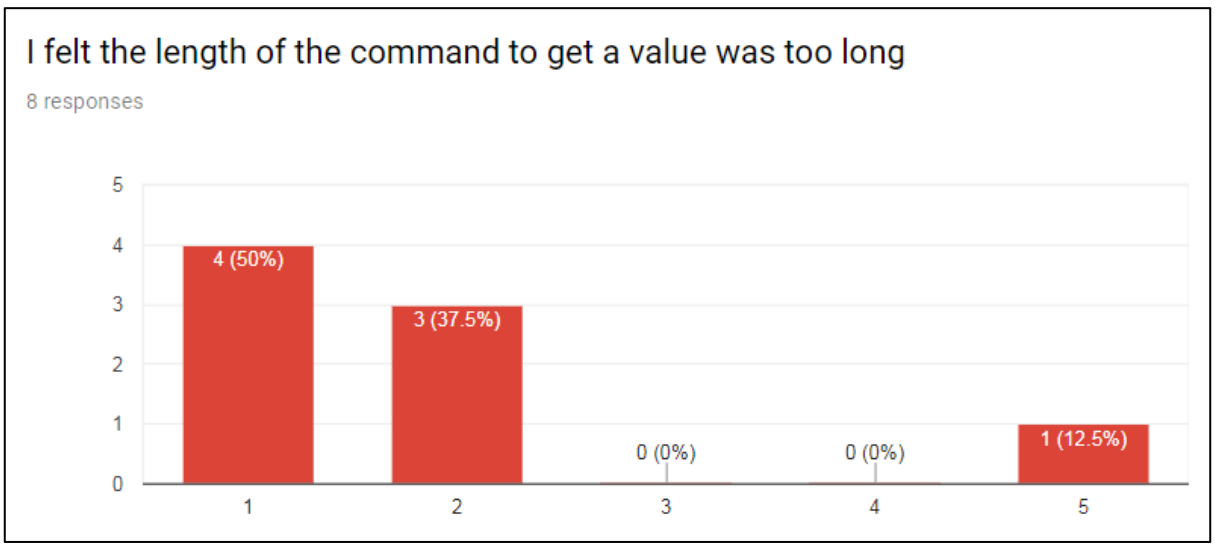
I was able to load new data into the solution easily

8 responses




I felt the length the command to load data was too long
8 responses
3
2
1
0
3 (37.5%)
3 (37.5%)
2 (25%)
0 (0%)
0 (0%)
1
2
3
4
5

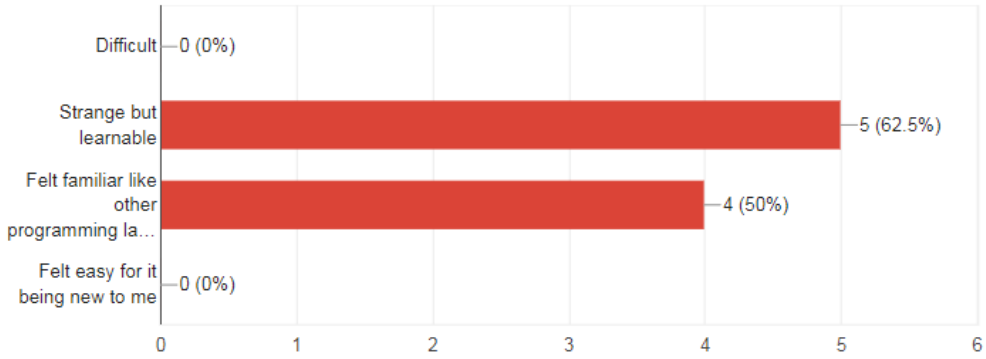

I was able to get a value from the program with the new data
8 responses
10
8
6
4
2
0
8 (100%)
0 (0%)
0 (0%)
0 (0%)
0 (0%)
1
2
3
4
5


I felt the length of the command to get a value was too long
8 responses
5
4
3
2
1
0
4 (50%)
3 (37.5%)
0 (0%)
0 (0%)
1 (12.5%)
1
2
3
4
5

Overall Opinion of the Package

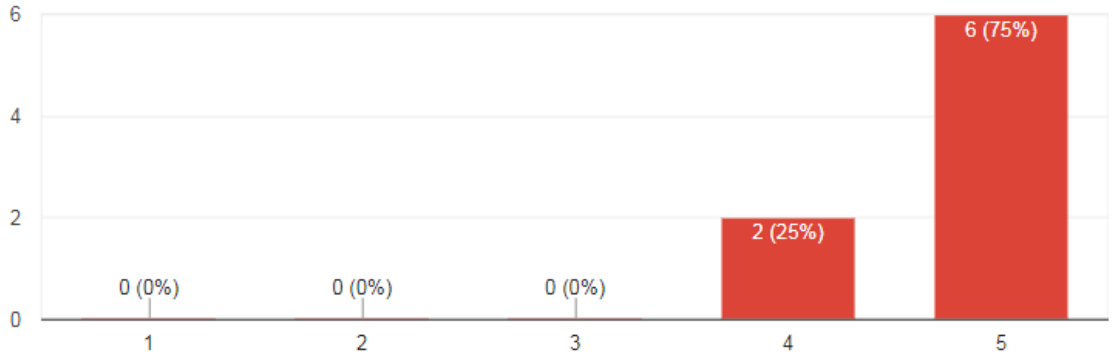
How did you find the syntax of the commands?

8 responses



I found the software interesting overall

8 responses



Any other comments on the software?

5 responses

Being able to use the text editor for commands was a great help.
It is very different from anything else I have used, but I did find it interesting and I think it could be very usefull
The software looks interesting but it is not clear what it does.
Showing the solution each generation would be interesting
The software overall is pretty awesome.

Any other comments on the guide?

7 responses

The guide was very well laid out however, a command was missing from the changing the fitness function which was a little confusing.

It was very good and easy to get though, the code examples helped a lot and were valuable. An example could be added to the changing the fitness function section, as I almost missed that I was supposed to execute some code there.

More in-depth with some parts. i.e. SelectionMethod

It needs to be clearer that changing the fitness function and selection method are parameters to a function

Make the fitness function section clearer

The guide was great however it should be clearer what commands should be run. Also the commands could be encapsulated a little bit better.

Make it clearer there are hidden parameters

15.8 Appendix H – Requirements with Colours Assigned

Requirement ID	Description	Priority
FR1-1	The package shall implement Cartesian Genetic Programming	Must
FR1-2	The package shall allow users to run the evolutionary process on a population of chromosomes	Must
FR1-3	The package shall allow users to step through each generation in the evolutionary process	Could
FR1-4	The package shall allow users to pause and resume the evolutionary process	Could
FR1-5	The package shall allow users to load a file containing the desired results	Must
FR1-6	The package shall measure the fitness of a solution compared to the desired result	Must
FR1-7	The package shall be able to construct classifier models for a given dataset	Could
FR1-8	The package shall be able to build regression models through evolution.	Must
FR1-9	The package shall allow existing R data structures to be passed into the program	Should
FR2-1	The package shall provide a symbolic regression function set consisting of mathematical operations	Must
FR2-2	The package shall provide a logical function set consisting of logic gates	Should
FR2-3	The package shall provide the $(\mu + \lambda)$ and Tournament Selection operators to be used	Must
FR2-4	The package shall provide mutation methods to be used	Must
FR2-5	The package shall allow users to choose a function set from the included choices	Should
FR2-6	The package shall allow users to choose a selection operator from the included choices	Should
FR2-7	The package shall allow users to choose a mutation method from the included choices	Should
FR2-8	The package shall allow users to define their own function set	Should
FR2-9	The package shall allow users to select a subset of the functions from a function set	Could
FR2-10	The package shall allow users to define their own selection operators	Could
FR2-11	The package shall allow users to define their own mutation methods	Could
FR2-12	The package shall allow users to change the parameters of the program such as: • Number of columns in chromosome • Number of rows in chromosome • Number of generations to run • The levels-back parameter • The population size • Random number seed	Must
FR2-13	The package shall provide multiple functions to be used for calculating the fitness of a solution	Should
FR3-1	The package shall display the results in a textual format in the R console	Must
FR3-2	The package shall create an output file containing the results	Should
FR3-3	The package shall display the results in a plotted graph	Could

FR3-4	The package shall output an R data structure containing the results through its return value to be used elsewhere	Could
FR3-5	The package shall allow users to load previous experiments to view the results as a graph	Could

Requirement ID	Description	Priority
NFR1-1	The packages computationally intensive parts shall will be written in C, C++ or Java to improve performance	Should
NFR1-2	Only active nodes shall be processed when decoding the chromosome to improve performance	Should
NFR1-3	The package shall stop execution when a solution has been found	Must
NFR2-1	The created graphs shall be visually clear and understandable to the user	Must
NFR2-2	The textual format shall be understandable and only contain necessary information	Must
NFR3-1	The package shall provide convenient methods of abstracting away from the lower level functionality	Should
NFR4-1	The package shall minimise errors that cause it to fail	Must