# Genetic programming library in Java with application to Ms. Pacman

by

Nikita Kuzmin
1980623

School of Computer Science
University of Birmingham
United Kingdom

Email: nxk825@student.bham.ac.uk
Supervisor: Dr. Ian Kenny

10 September 2019

Report submitted in partial fulfilment of the requirements for the degree of MSc in Computer Science in the University of Birmingham

# Abstract

This project is aimed at developing a functioning and generic Genetic programming library in Java. The purpose of the library is to enable creating, running and evolving custom code that can be read and analysed by researchers. The application of the library has been done in the game Ms. Pacman, with the highest result of 32840 out of 100000 being achieved by the best AI program generated. The resulting library is a functioning piece of software that can produce comparable results to other research in the field, whilst remaining generic and easy to use.

Key words: Java, Library, Genetic programming, Ms. Pacman, AI.

# Acknowledgements

Thanks to Dr. Ian Kenny for supervising this project and guiding on the matters of literature research and project management.

# Contents

# Chapter 1

# Introduction

The main aim of this project was to develop a Java library that would help generically employ Genetic programming techniques. The main inspiration was the work of John R. Koza in his knowledge domain, whose approach was used as the basis of the developed library [3].

While the field of genetic programming is very broad and there exist complex commercially available libraries, that perform this AI technique, the main goal for this project was to make Java create and run custom code by itself. This means that it was of the highest priority to develop a framework that would be able to take smaller programs written by the user and mix them in an optimal way to solve a problem in a given domain. Furthermore, the outputted code should be run on its own and be presented in an easy to understand form.

With the creation of the library this project aimed at the successful application of said software to Ms. Pacman as a game of choice. This game, whilst being more complex than traditional Pacman tackled by Koza, has had genetic programming applied to it[1, 2]. Therefore, having other research as a base for further investigation helped in tuning the library to make it generate better code and prove its functionality.

The developed library is not supposed to compete with commercially available software products, but to be a representation of a concept, that envisions any object-oriented language being able to create its self-produced code, run it and mature it without technical errors. Lastly, the library, being tuned to tackle a specific game, remains generic and independent of a domain it is applied to, thus presenting a stepping stone for new researchers that are not familiar with the language LISP.

This report assumes that the reader knows the basic rules of Pacman style games as well as a moderate understanding of Java or any other object-oriented programming. The genetic programming basics are going to be addressed and put to the application at a later stage in the report.

The structure of the report is as follows: Background, Design and implementation, Application, Discussion and conclusion. Background section is going to give a better look at the work of Koza, whose framework is going to be used in the development, as well as other researchers, especially from the application side. Design and implementation will describe the development process of the library, with a guide for the usage of the software, description testing and project management techniques employed. In the application chapter the library is going to be applied to Ms. Pacman to verify its functionality and compared to other research. Lastly, the report will end with a discussion of the successes and deficiencies of the library that were achieved during the development and testing.

# Chapter 2

# Background

In this chapter, we are going to discuss previous work that inspired the project, as well as developments in the field that led to the decisions made when producing the software. Furthermore, we will look at the application of genetic programming to Ms. Pacman game that the built library will be tested on. This will ensure that we have a grasp of the results that we would like to achieve, as well as investigation methods used.

Some criticism will also be provided, to show that there is still progress to be made in the field of application of genetic programming.

## 2.1 Genetic programming background

The field of genetic programming is based around the idea of the evolution of small programs that are aimed at delivering a certain result when applied to a specific problem. The first accounts of the discussion around evolving programs date back to 1950s when Alan Turing first thought of the idea.

25 year later John Holland laid out the basic theoretical foundations of the science which was then empirically applied by Richard Forsyth in 1981, to perform classification of crime scenes in the UK.

This leads us to one of the most prominent figures in this science field and an inventor of a multitude of patented methods John R. Koza, who wrote 19 books and over 200 publications on the matter.

Koza created a framework representing his approach to genetic programming that is universally used by many academics who employ this part of

Artificial Intelligence in their publications, which is why the proposed library was chosen to be built according to his specifications. Furthermore, the decision to create such a library would allow to test it against Koza's work, and empirically prove the success or failure of it to deliver results that a classic genetic programming technique would.

The main reference used in this project is the "Genetic Programming: On the Programming of Computers by means of natural selection" by Koza, written in 1992 [3]. This book lays out the whole process used by him, as well as applications and different methods and mathematical structures used to make the system work.

The reason why this project is not a simple copy of Koza's work lies within the root of how the technique is employed: LISP. LISP is a language that Koza and many other academics chose to use due to its grammar being a representation of not only the code but data structures as well, making it simple to employ and tweak. Another advantage is the EVAL function that can instantly execute a set program, making running the programs very convenient and extracting their values efficient. This does not align with Java, which is not a scripted language and it cannot output runnable code.

However, Koza said during the discussion about his choice of language that: "... virtually any programming language is capable of expressing and evaluating the compositions of functions and terminals necessary to implement genetic programming", thus creating such library in Java would be possible, but some obstacles discussed in the later chapter would have to be avoided or dealt with [3].

Now that we know of the tools that Koza used, we have to talk more about the process and the structure of classical genetic programming and its principles. Firstly, we have to talk about the initial setup for the technique to work in a given domain, which leads to the representation of the programs that are going to be produced. The representation chosen are trees, which can be seen as the decision trees that can be run as scripts shown by the figure 2.1 below.

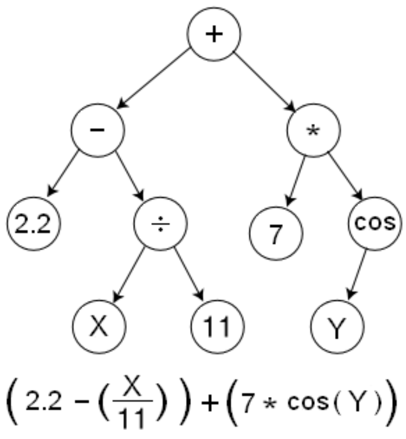

+
−
*
2.2
÷
7
cos
X
11
Y
$\left(2.2 - \left(\frac{X}{11}\right)\right) + \left(7 * \cos(Y)\right)$

Figure 2.1: Tree representation

The decision tree consists of 2 types of nodes: functional and terminal. Functional nodes are used to guide the program, as they consist of conditional clauses, and lead the program towards the last nodes of a branch, which are terminal nodes: the decision made by the tree. This structure allows Koza to create two types of programs, including the ones that route the computer with conditional statements and the ones that perform an action. This structure is well supported by LISP and is used for quick visual analysis of the outputted code.

This approach in terms of data structures is going to be used in the creation of the library, as it is intuitive to read and understand, however not as well supported in Java.

After we have chosen the data structure used for storing and executing the programs, we have to talk about the evolution process. Koza puts to practice the theoretical model of Darwinian evolution when creating new generations of trees, which means giving a higher chance of reproducing to better-fitted trees done by setting up a custom probability distribution. This implies using a certain type of fitness evaluation, which is going to be thoroughly described in the Design chapter.

Once the probability distribution is done and the trees for reproduction are chosen, Koza creates new trees via the crossover mechanic shown in Figure 2.2 below:

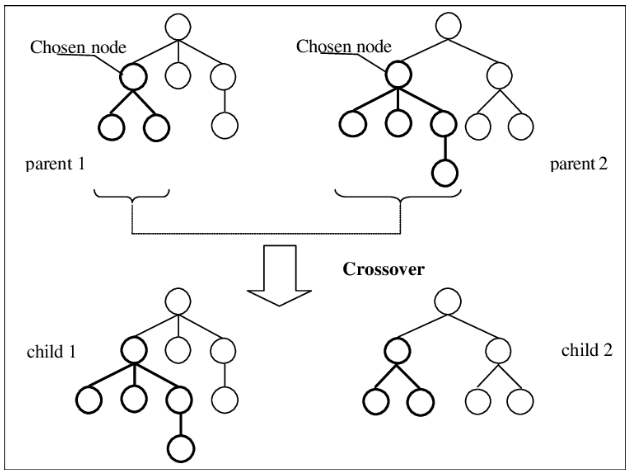


Figure 2.2: Tree crossover

A crossover takes two trees from the probability distribution and randomly exchanges nodes of those trees. This allows new tree structures to be created and provide for more diverse future generations.

In conclusion, this research has provided us with the goals that the library has to achieve and what structure it has to follow. The library has to create an efficient tree structure, that can be run and stored to be outputted to the user and support and equivalent of EVAL from LISP. The trees have to consist of functional and terminal nodes, that can guide the computer through them to make a final decision. Further, the library has to be able to create custom probability distributions to give an advantage to higher fitness trees to be able to reproduce and crossover. Lastly, the library should be able to run the trees when given by the user, for example, the best tree from one training cycle.

## 2.2 Genetic programming application

The application of genetic programming has been done an excessive amount of times especially towards Pacman games, which is very helpful in gauging the success of the library [2, 1]. Koza himself, in the book mentioned above, applied his technique to a classic Pacman setup. However, the project is going further in the application and is going to try and beat Ms. Pacman, which is a more complicated version of the game. Despite having a different game, Koza's approach has a valuable control setup, as his simpler methods can be tested against more challenging environment. This can then be extended, by adding methods of higher complexity seen in more modern research.

The other relevant research that this report is going to reference during the application stage is "Evolving diverse Ms. Pac-Man playing agents using genetic programming" by Atif M. Alhejali and Simon M. Lucas [1]. Their application of genetic programming to Ms. Pacman was done with the help of Koza's research, making it a continuation of his work, while using more refined programs and making use of more sophisticated hardware to create better solutions.

Alhejali and Lucas' approach involved employing LISP and the same tree data structures described above, that were standardised by Koza. However, having a more complicated game on their hands, they have increased the complexity of the domain methods they inputted into the technique, to better scale their results. This is very useful in the application stage of this project, as seeing the progression of the decision trees with more powerful methods can show the extent to which they truly impact the outcome of the evolution of produced programs.

On the other hand, their work has some downsides in their approach, being that they have included an arbitrary number to be used in their programs that is set at random for each decision tree. This decision undermines the consistency of the fitness produced by a said tree and secondly introduces a factor that forces the training to be done multiple times to statistically hit an optimal value. Moreover, they have adapted the generation and the crossover of the trees. This modification affects a functional node also used by Koza, where the node takes in 4 arguments and compares the first 2, to then choose either 3rd or 4th to be executed. They modified the node, such that only certain methods can be inserted into each slot, thus restricting the possible outcomes. This can hurt the production of optimal trees, as these researchers are trying to tailor their system to behave more logically according to them. This report, as well as Koza's application to Pacman, shows that such restriction is not necessary and if this way of building the tree is superior, the evolution process will develop it by itself, and on the contrary, will create a better outcome in terms of fitness.

This is shown in the best result achieved by Koza when beating Pacman, as there are multiple nodes as described above, that compare movement methods rather than physical distances. This may have caused spurious decision making by the tree, which is not appropriate according to Alhejali and Lucas but was shown to be successful.

Therefore, this report will stay away from such restrictions and force the evolution to create the best outcome it can, while staying true to the Koza's framework.

To test the library, a Ms. Pacman package has been used that was developed by a group of undergraduate students from Madrid as their final project. The group created a game together with a Ms. Pacman bot, that utilised Grammatical Evolution, considered to be a variety of Genetic Programming [4].

Whilst, their game has been a very great help to assess the success of the library described in this report, their results are going to be used to challenge the output of Genetic programming library. One of the main features of the library is that it contains different types of ghosts' controllers, such as the classic Legacy controller, Random and lastly Aggressive ghosts [4].

A big downside of their approach, however, can be discovered upon closer inspection of the distribution of the average fitness per generation results. Mantecon et al. have attempted to evolve their Pacman controllers against different types of ghosts' controllers, which led to a strong discrepancy in their results, as the success of their controller against Random ghosts has been much more superior compared to training against classic ghosts, that you would find in an arcade machine. Therefore, it may be concluded that their investigation was not successful, due to their Pacman not being able to beat a more sophisticated opponent.

The main take from these results is that the biggest challenge consists of beating the Legacy ghosts controller, as it is the hardest one, however, using other packages can be of use to train a certain type of behaviour to our Pacman controller.

# Chapter 3

# Design and Implementation

This chapter is going to describe the design process behind the creation of the Genetic programming library. This will include the challenges that were faced in production and implementation and also a general user guide to the library. While detailing all of the intricacies of the system for more comfortable future use and development, the section will be finished with the discussion around testing and project management.

## 3.1 Library specification

Genetic programming library has to adhere to the scientific specifications outlined by Koza in his work that was described above. The library then is limited in variation as it has to deliver results that are expected out of software that uses this technique of Artificial Intelligence.

These specifications below of the software describe expected functionality that was used by Koza and other researchers in the field, therefore, if the library provides the user with them, it can be considered to be suitable for some scientific application. However, this library's scope has to be considered to be centred around running custom code created via randomness and evolved using a Darwinist approach and then outputting the best runnable code. This scope does not include using genetic algorithms and more complex mutation processes, as such was not used by Koza at the time. Therefore, while this library should be fit to be used in this specific research, it is by no means aimed to be a complete genetic programming package. The expected functionality is as follows.

As previously stated the library has to operate in Java using a tree data structure. This data structure has to be created by the library, by employing user inputs to create an initial generation of programs. The trees have to be generated in a random fashion using the functional and terminal nodes, with all of the final nodes within the tree being terminal and parent nodes being functional. The library has to give an ability to the user for them to call a tree to perform a calculation. At a point defined by the user, the library has to switch an operational tree for the next one in the generation to be executed and communicate the end of the generation. At the end of the generation, given the accumulation of the fitness functions, the library has to create a custom probability function that will determine, which trees are chosen for reproduction and crossover per Koza (1992). The library has to give the user an ability to use an alternative of LISP's EVAL function when they are specifying their methods to be inputted into the library. Lastly, the library has to give the user the best tree after a training cycle, which would be a text representation of the tree with its result, together with an ability to run this tree on its own.

The user has to input the methods that correlate to the terminal and functional nodes, as well as the names for the methods to read the outputted code. The user has to have control of when to create an initial generation and to what depth the initial trees can go, as well as the size of the first generation. The user has to control when the library executes the decision tree, to give the user the result calculated. The user has to input the fitness of each tree within the code after the execution of said tree is no longer needed. The user has to command the library to create a new generation, to continue operation of the training.

## 3.2 Development

Since we have a clear set of specifications, which are provided for a generic genetic programming library, we can now discuss the development process and the crucial parts of the software that make it into a library.

### 3.2.1 Tree data structure

First off let us start with discussing the implementation of the tree data structure that will drive the creation and execution of the programs. The tree data structure does exist in Java, however, it does not imply manipulation of code, which means that we have to create our data structure such that it can handle having multiple branches going off of a node, as well as being used to execute code in different ways.

The choice fell on implementing a String manipulation setup, where the trees are going to be written in String class variables using LISP grammar that was used by Koza and other researchers in the field for reasons described in the Background chapter. LISP provides a simple way of reading code and stores the whole tree in a memory light String format. To implement this a parser has been built that would take this String and put it into an Arraylist of Strings, where each input is a name of a node or a bracket. The bracket structure in LISP allows the library, later on, to move the pointers across such lists and keep track of where it is in the tree, as they help defining the depth. An example of such a String tree is seen below in Figure 3.1.

```
( 4ifle
   ( distToEdibleGhost 4ifle
      ( distToEdibleGhost 4ifle
         ( distToNonEdibleGhost moveToFood
         moveFromPower moveToFood )
      distGhost moveFromNonEdibleGhost )
   distGhost moveFromNonEdibleGhost ) )
```

Figure 3.1: LISP tree

Once we have produced a parser that would put these strings into a more readable format for execution, we have to look at the generation of these trees for the initial structure, which represents the first generation. To start generating the trees the user has to provide the software with names of the methods they want to be included into the genetic programming, therefore there is a requirement for a data structure that would hold such information.

The library offers an Object *LispStatements*, which holds the information on the naming of the methods that will go into the creation of the decision trees. Two field parameters have to be filled in, one is an Arraylist of functional method names and another is an Arraylist of terminal method names. The functional method names have to have a number in front of the name that has to specify the number of branches that can come out of this node. The example is seen above in the form of "4ifle" node that always has 4 nodes branching out from it. Terminal nodes can be named anything, however, the user might want to name them appropriately for easier code reading later. Furthermore, the order of inputting the names is important as well, but we shall touch on it at a later stage of the development description.

Having acquired the method names from the user, we can now generate the first generation using grow tree and full tree methods. Grow tree method involves giving a program the names of the methods and the desired depth, where the method takes a random node from a combined list and naturally grows the tree until a branch stops growing if a terminal node was chosen or hits the depth limit and is forced to put a terminal node in. This creates a large variety of tree shapes at each level of depth and allows for much more dynamic evolution in the future. On the other hand, full trees are forced to grow each branch to the maximum depth provided. However, as per Koza, using either is not as efficient, therefore, the library in the creation of initial structure uses 50% of each for every level of depth until the maximum, that is set by the user.

The standard depth for the first generation used by Koza is 6, which is going to be the default for the library, but it can be changed during the setup by the user. The maximum depth through the evolution is going to be set at 17, which might not be enough for some applications, however, the research on the common applications of genetic programming does not indicate trees of depth 17 or more being prevalent. The initial generation would then consist of 200 trees per depth level, with 100 full-grown and 100 naturally grown trees. This should cover most of the variation in tree shapes and configurations and give a solid base for the trees to crossover and evolve in the future.

### 3.2.2 Evolution

Before we start applying the trees to an application of choice, we have to address the evolution mechanism and how we are going to store the data for the said trees to create our own probability distribution to choose the trees that would partake in the reproduction and crossover. Firstly, the library operates through a static class *Generation*, which acts as a database for the information regarding each generation. This class contains a list of all of the *Tree* objects that have the String of each tree and their respective fitness values. This allows us to have all of the information about the generation the library is working on in one place.

After we collect all of the fitness values for the trees, we consult Koza for the mathematical manipulation of them to create the probability distribution. This process requires the user to input standardised fitness which is described by the function below.

$$s(i,t) = r_{max} - r(i.t).$$

S represents the standardised fitness, which is calculated by taking the maximum possible fitness value and deducting the raw fitness of the tree. This means that the lower the standardised fitness is the better. Next, we have to adjust our fitness, such that if the values are always between 0 and 1. This will allow for the better trees in the generation to have higher values and get a better chance of getting chosen. Furthermore, to create a probability distribution, all of the fitness values have to add up to 1, which means they have to go through the normalisation process. Both adjustment and normalisation are seen in the functions below.

$$a(i,t) = \tfrac{1}{1+s(i,t)}. \quad n(i,t) = \tfrac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}.$$

Lastly, since Java does not have a built-in probability distribution creator a custom roulette has been built. To make it work, all of the trees were sorted according to their normalised fitness values and given a probability calculated by the accumulation of normalised fitness functions before it including itself, to determine a probability, depicted in the next formula.

$$p(i,t) = \textstyle\sum_{k=0}^{i} n(k,t).$$

Now the roulette picks a random number from 0 to 1000, and the tree that is first to have higher probability multiplied by 1000 compared to a random number will be chosen for crossover or reproduction. Due to the better trees having higher normalised fitness values, they have a larger chance of getting picked by this roulette and since the trees are sorted, we can easily tweak the system to choose the fittest trees only, which will be done later in the application stage.

Using this custom probability distribution the library fills the new generation with 80% crossover trees, where two chosen trees exchange 1 node each (Figure 2.2), such that it does not break the grammar of the LISP String and the maximum depth of any branch does not exceed 17. The rest of the new generation is then filled with the trees chosen by the same probability distribution, but not altered.

### 3.2.3 Execution

The library has to provide for the execution of the code, as it needs to collect the fitness information from the user to create new generations. This is done via a *LispExecutor*, which operates through calling the methods that the user has generated (Figure 6.3). To generate the terminal and functional methods the user has to implement an interface Executable, which contains several functional and terminal methods. The class that contains these methods will then be stored in the *Generation* static class, such that the library can access it with the user providing it to the library once.

The order in which these methods are filled in is very important, as the method names given by the user at an earlier stage using *LispStatements* object are going to reference these methods in the following way: as the *LispExecutor* reads the name of a method it will track its index in the *LispStatements* terminal or functional list and use this index to call an appropriate method. This leads to the first method name in the functional *LispStatements* list to reference the *functional1* method from the interface Executable. The same applies to the terminal methods, however, there is a discrepancy in the way that these methods have to be programmed.

The functional methods have to return a number, which would indicate what branch that comes out of the respective functional node the library has to access next. Therefore, the number returned has to be in the range of 1 to the number of children the node has. Terminal methods, on the other hand, have to return an integer value since such methods can be used for the adapted EVAL method from LISP, which we are going to talk about later. Once the *LispExecutor* reaches any terminal method, it stops, as it has made a decision. If the user needs another decision from the tree they have to call it again via the execute method in the *LispExecutor*.

For every generation including the initial structure, the user has to update the *Generation* class to start getting execution decisions from the next tree (Figure 6.3). Generation class has a method that will update the pointer, such that the next tree will be pulled from its list when the $LispExecutor.execute()$ method is called. Furthermore, before every update on which tree has to be executed next, the user has to input a standardised fitness of the tree they were just executing. This part is critical for the process as the tree otherwise will have no fitness and will not be considered in the creation of the new generation. Lastly, the *Generation* class can give the user a boolean answer for whether the full generation has been executed and if the user can call for the next one to be created. Before the new generation is created *Generation* class stores the best result that the process has outputted so far, such that the user can record if the past generation has created a better decision tree.

Lastly, the library provides an alternative to the EVAL function in LISP that can evaluate the outcome of any branch. This is useful when a functional method has to compare two of its arguments to make a decision. Therefore, each of the functional methods filled in by the user will get a node, that can be forwarded to the evaluation method in *LispExecutor*. This method also requires the indices of the nodes that have to be compared, in turn meaning that every terminal node has to return a value as said above. The evaluation method collects the values from the branches going out of the node calling it, returning a 0 if the first node is bigger and 1 otherwise.

## 3.3 User guide

Now that we have a detailed description of the library's operations we will briefly look at what is expected from the user side and then move onto testing.

The first step in the process is to write the appropriate programs by filling in skeleton methods provided within the interface *Executable*. Once at least one functional and one terminal method have been filled, the user has to create an instance of the object *LispStatements* and fill the terminal list with the names of terminal nodes in their respective order, with the same procedure for the functional nodes' names. When the method names have been submitted the user has to choose where in the application the decision from the tree will be called for by writing the *LispExecutor.execute*(). This method will not return anything for the user, therefore all of the manipulations that have to be carried out have to be done within the terminal methods beforehand. An example of this will be shown in the application chapter.

It is imperative for the user to give a tree a standardised fitness value, before ordering the library to get the next tree ready for execution. When the fitness has been inserted the next tree can be put into the executor by employing *Generation.pointerUpdate*(), which will move the pointer in the static class *Generation* and check for the emptiness of the fitness function value in the previous tree.

The user also has to check whether the generation is finished by asking the static class *Generation*, if all of the trees have their fitness values set. When this point in the process occurs, the user has to call for the new generation to be created and start the process all over again (Figure 6.3).

This procedure gives the user full control of the library operation and freedom to choose when the library has to perform certain actions and not forcing the user to be time-constrained. Furthermore, the library supports any domain, as the methods are written by the user and the decision is called for whenever desired, meaning that it is flexible and adjustable.

## 3.4 Testing

This section is going to discuss the testing of the library as well as error checking that would prevent the user from incorrectly using the package. It will be shown later in the application chapter how all of the features in the library can be used effectively providing us with all of the black-box testing, however, being a showcase it does not provide the full extent of testing that has to be done to check the correctness of the software. Therefore, here we are going to discuss the white box testing that was done to make sure that the library can handle extreme cases, as well as error checking, to make sure the user does not input any values that will confuse the library.

The library contains two testing classes that check for the functionality of non-random operations, as well as those that did not get fully tested by the application stage of this project. The testing tables are available in the appendix of the report, but we are going to show that the library operates in line with the specifications outlined above (Figure 6.4).

LISP parser is functioning as intended, providing for the conversion of generated String trees into an executable format. While it cannot be fully tested, the randomly generated trees have an appropriate structure, shown through black-box testing. The pointer in the static class Generation works appropriately and does not allow the user to create a new generation when some of the trees have not acquired their fitness functions. During the setup stage, the user is warned if wrong method names for functional methods are inputted, as well as one of the methods lists being empty. This prevents the library from running into issues before building the initial structure and while performing the evolution runs. If by mistake a negative standardised function is given to a tree, the library will throw an exception to warn the user that maximum fitness was not correctly configured. Both execution from the Generation class as well as execution of typed code via *LispExecutor.executeBest*(), worked as intended depending on the proper setup of the library.

While it seems that the testing of the software covers most of the functionality of the library, some of the edge cases can only appear once the library is used by many users with different applications. This will stress test its capabilities and provide for a better testing ground. Unfortunately, due to the time constraints and technical issues with 3rd party software, broader black box testing was not possible. Most of the games that were thought as good testing grounds appeared to be non-functional, with no time available to fix them.

Overall, the robustness of the library comes from the project management that was implemented, which is going to be discussed in the next section.

## 3.5 Project management

The project management from the start of the development aimed at delivering working parts of the library approximately every one to two weeks. This meant that all of the releases were in a working condition and delivered tangible and testable results. This strategy worked until the library was fully finished, but could not successfully continue into the application process. Once the library was finished, it was initially planned to get two different games that could benefit from their controllers to be built by the library, however a roadblock was hit, as most of the relevant software online was not working and only a Ms. Pacman game developed by Spanish students was up to a good standard. This meant that the project turned from a more agile style into a spiral, where the investigation of the library's capabilities brought up a new ways of its application to the same software.

Therefore, in the last weeks of development, not a large amount of new code was done, but rather a fine-tuning and application of different strategies were applied, to get the best results out of the Pacman controller. This decision was beneficial for the project as training of controllers took a lot of time due to hardware restrictions, making this prioritisation produce a variety of good results that will be discussed in the next chapter. If the pursuit to apply the library to more software was to continue, it would most probably lead to less than ideal evaluations of library's functions across several software packages, rather than one solid investigation.

On other hand, initial planning relied on the research that was done before project proposal was finalised. Thus, lack of understanding of the underlying problems that can come with such application heavy project degraded the quality of initial planning and led to unforeseen circumstances. If such challenges were predicted beforehand, other functionality of the library might have been developed, increasing the variety of its application possibilities. Such functionality would include steering away from keeping functional nodes to just being conditional statements, but rather also be computationally intensive. This, however, can be pursued in the future work.

# Chapter 4

# Application

In this chapter, we are going to be looking at the application of the library that was described above to the game Ms. Pacman. This application is going to form an investigation about the success of the library and the possibilities of getting better results than previous research by using genetic programming technique that this library provides.

## 4.1 Initial setup

It has to be made clear that creating a game of Ms. Pacman was outside of the scope of this project, therefore the game used in the application of library was created by another entity. In particular, the package that includes not only the game but useful methods to calculate relevant values, which were employed by the library, was written by a group of Spanish undergraduate students. A visual representation of their game can be seen in the Appendix (Figure 6.1). The goal of their project was to create a game and test how Grammatical evolution would do when controlling Pacman [4]. This means that the game had a controller for Pacman, which is exactly where the library would fit, as it was able to take over the controller and send moves to the game executor.

As the executor would call for the moves to be made it gave both ghosts' and Pacman's controllers a copy of the current game state, which could then be used to analyse the whole field and calculate all of the necessary values. The executor of the game would not advance the game until it received moves from both Pacman and the ghosts, meaning that there was no need to simplify the computation of the methods to cut on the time to make a decision, however, it was still not an issue, which will be discussed later. Having all of the information given to the controller class at every step of the game, it became a perfect place that would implement *Executable* interface and give control of method calling to the library.

Having set up the way our library is going to interact with the game, we can look at the first method setup that would allow us to test the effectiveness of the library. Most of the methods used in the investigation have been borrowed from the literature on the subject to see whether the library can create comparable results, however, some have been changed, to accommodate for more consistency.

First of all, we have to start with the main inspiration for the project: the work of John R. Koza. In his investigation of the application of genetic programming, Koza created a simple setup to try and beat the classic Pacman game, which is a perfect starting point for us. Below is the full description of this setup, as well as numerical values that the library uses in its calculations.

13 primitive operators are used by our *SimpleController*, where there are 2 functional operators and 11 terminals. Each of the terminal operators returns a value, however, the operators that are responsible for moves of Pacman, set a variable *nextMove*, which is then sent to the executor once the tree has finished making a decision. *nextMove* is set to be what the last move was, or at neutral if there was no last move unless any method called by the tree makes a change.

- 2ifblue - A functional operator that takes in 2 arguments and chooses 1st argument if the ghosts are currently blue and chooses 2nd argument otherwise.
- 4ifle - A functional operator that takes in 4 arguments and utilises an EVAL function, by comparing the 1st and 2nd argument. If the 1st argument is bigger than 2nd, the 3rd argument is chosen, with the 4th chosen otherwise.

Now we are going to look at the terminal operators that are going to be used by this controller. Firstly, interactions with the power pill.

- MoveToPower - gives a direction along the shortest path from Pacman to the nearest Power pill. In a scenario where there are 2 routes of the same size this method, as well as other similar methods, will make an arbitrary decision between the two. This function, together with functions that are responsible for movement will return the facing direction as a modulo 4 with 0 being north, 1 being east and so on as their value.

- MoveFromPower - gives a direction away from the closest power pill. This direction is the closest to topologically opposite of MoveToPower direction as possible.

- distToPower - gives a value for the shortest distance to power pill along the paths of the maze.

Next, we are going to talk about the methods that concern actions towards ghosts. This excludes ghosts who have been eaten by Pacman and are returning to their den.

- distToGhost - shortest distance to the closest ghost, whether it is blue or not

- distTo2ndGhost - shortest distance to the 2nd closest ghost, whether edible or not

- distToEdibleGhost - shortest distance to the closest edible ghost if there is one. If there are no edible ghosts, this method returns the maximum distance possible within the maze.

- distToNonEdibleGhost - shortest distance to the closest non-edible ghost if there is one. Similarly to the last method return maximum distance if there are none of these ghosts.

- moveToEdibleGhost - returns a direction towards the closest edible ghost along the shortest path towards it.

- moveFromNonEdibleGhost - returns a direction away from the closest non-edible ghost along the shortest path towards the furthest topological point away from that ghost.

Lastly, there are just 2 interactions with the food that is scattered around the maze for Pacman to consume.

- moveToFood - gives a direction towards the closest pill along the shortest path towards it.

- distToPill - shortest distance to the closest pill in the maze to Pacman.

Now that we have all of the methods for this simple controller we have to talk about the setup for the library. Staying closer to Koza's investigation we have chosen to set a maximum depth for the initial structure at 6. This means that with 200 trees generated at every level, each generation is going to consist of 1000 trees. With all 4 mazes being implemented, the maximum possible raw fitness is set to 100000 points, which can be reached by successfully eating all 4 ghosts after every eaten power pill and clearing out all of the food pills without losing any lives.

Talking about the evolution, 80% of it is going to be made from the crossover trees and the rest from pure reproduction. There are going to be 51 generations including the initial structure and the raw fitness being an average of 10 runs of the tree, to make sure that we do not allow a tree to get ignored due to one bad run. Furthermore, only the top 10% of the trees are going to make into the crossover process or reproduction, due to the initial experiments showing a large flaw of the technique without this restriction.

The flaw consists of the fact that at the initial stages most of the trees are earning close to no points of raw fitness. This has been seen by Koza in his research as well. However, this means that a lot of these low ranked trees can make it into the crossover and reproduction, polluting the next generations and giving non-effective nodes to higher performing trees. This led to 99% of the population having the same low result of 40, which meant that Pacman would get stuck in one place after eating 2 pills. Therefore, the decision was made to force the library to use much better trees for crossover and reproduction, which then yielded better results.

On the other hand, this flaw goes hand in hand with the complexity of the methods that are used by the library and as we raise the complexity, we are going to lift this restriction and see how much worse it is going to perform. For now, we cannot afford not having this restriction as the results that would appear are of no value to this investigation.

Lastly, there are different types of ghosts provided by the game package used, but for this initial experiment, it was chosen to go with the classic Legacy controller for the ghosts. This controller replicates the behaviour of the ghosts that one would find in the old arcade machines with all of the ghosts having their own strategy and "personality", making the game authentic and closest to the one Koza investigated.

Below is the summary of the setup that we are going to implement.

| | |
|---|---|
| Objective: | Find a computer program for Ms. Pacman that scores the maximum points in the game. |
| Terminal set: | MoveToPower, MoveFromPower, distToPower, distToGhost, distTo2ndGhost, distToEdibleGhost, distToNonEdibleGhost, moveToEdibleGhost, moveFromNonEdibleGhost, moveToPill, distToPill |
| Function set: | 2ifblue, 4ifle |
| Raw fitness: | Average of points scored in game over 10 games. |
| Standardised fitness: | Standardised fitness is the maximum number of points (100000) minus raw fitness. |
| Parameters: | M = 1000, G = 51. |

After running the library through the whole procedure, while also collecting data, we are met with the best tree with an average result of 5115. The code for the tree was the following:
( 4ifle ( distToEdibleGhost moveFromPower distToPower moveToFood ) )
The strategy that this tree employs is to eat as many pills as possible, while also eating the power pills on the way to deter the ghosts from itself. This is not a very consistent strategy as it's success strongly depends on the ghosts and their actions. Furthermore, this controller does not eat ghosts if only by accident, meaning that it is missing out on a lot of points. Lastly, it is evident from the average per generation graph below that the trees are struggling to get through the 2nd maze after completing the first one, as the minimum score to complete the first maze is 2200.

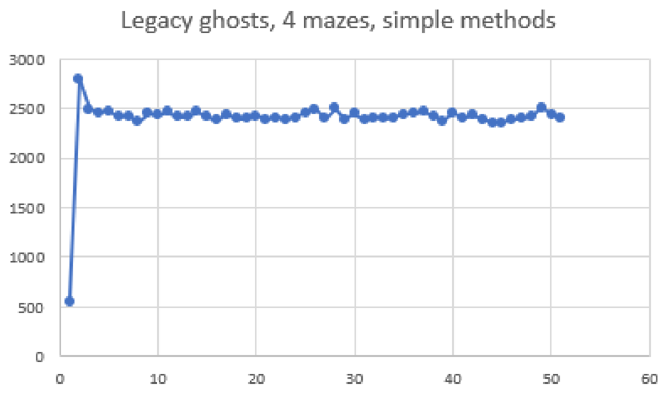


Figure 4.1: Average per generation: Legacy ghosts, 4 mazes, Simple methods

One of the explanations for this includes the lower complexity of the methods, whilst having the complexity of the game increased. Therefore, the trees being able to clear out 1st maze, as they did for Koza, when introduced to the 2nd maze are unable to adapt, such that they can't still complete the 1st maze and tackle the 2nd.

This is the reason why this investigation has to turn to other literature and more involved algorithms plugged into the tree methods, to get significant results.

## 4.2 Higher complexity investigation

In this section, we are going to investigate effects of not only choosing higher complexity methods but also apply different strategies with the main goal to get the highest score in the 4 mazes Ms. Pacman against Legacy controller for ghosts.

To give us a head start we are going to reference Alhejali and Lucas, who have done this experiment with Ms. Pacman using genetic programming. They have included more functional operators as well as a breadth-first search algorithm that would find the safest route for Pacman as one of their terminal operators. This level of complexity would most probably boost even the least fit trees to perform well, but that can only be checked once we acquire the data.

First of all, we have to introduce new functional operators:

- 2ifindanger - this 2 argument operator chooses the first argument if the nearest non-edible ghost is closer than 15 steps away and second otherwise.
- 2ifpowerpillscleared - this 2 argument operator chooses the first argument if there are still power pills in the maze and second otherwise.
- 2istopowersafe - this algorithm analyses whether the shortest path to the closest power pill is safe. The algorithm takes every point along this route and looks whether the Pacman or any other ghost is closest to it. If any ghost is closer to any point of the route than Pacman, the route is not considered safe. If the route is safe the first argument is chosen, second otherwise.

Now we are going to look at a couple of additional terminal operators included on top of the once used prior.

- DistTo3rdGhost - this method returns the shortest distance to the 3rd closest ghost.
- runaway - this method calculates the safest way for the Pacman to go, this means that a Breadth-First search algorithm is employed to check every route available and choose the safest one. If two routes are both equally safe the shortest is chosen. If they have the same length one is arbitrarily chosen.

Runaway method especially will become of value to the trees that will be generated in this experiment, as it has the power to save Pacman from being eaten and continue to get points[5].

It has to be noted that the application of the operators described above is going to be different, compared to the original application by Alhejali and Lucas. The library is set up in a way, which delivers a decision at every game state, which was not the case for the researchers [1]. This was a major criticism of their work, written by Brandstetter and Ahmadi [2]. In their work, the trees would generate a move after every step, dependent on the current state of the game, instead of relying on prediction. The developed library is going to have the same approach towards providing directions to the controller, as the results by Brandstetter and Ahmadi proved to be an improvement over previous research.

Having acquired all of the necessary methods for our experiment, we are going to include them into our previous setup and add some adjustments such as setting the starting maximum depth of initial structure trees to 8. The decision to set this value to 8 comes from having more methods at hand, therefore we need more variation that will affect the evolution and give new trees deeper and more complex nodes to work with.

Below are the results that this setup has achieved, which already shows a big improvement over a simple method setup.

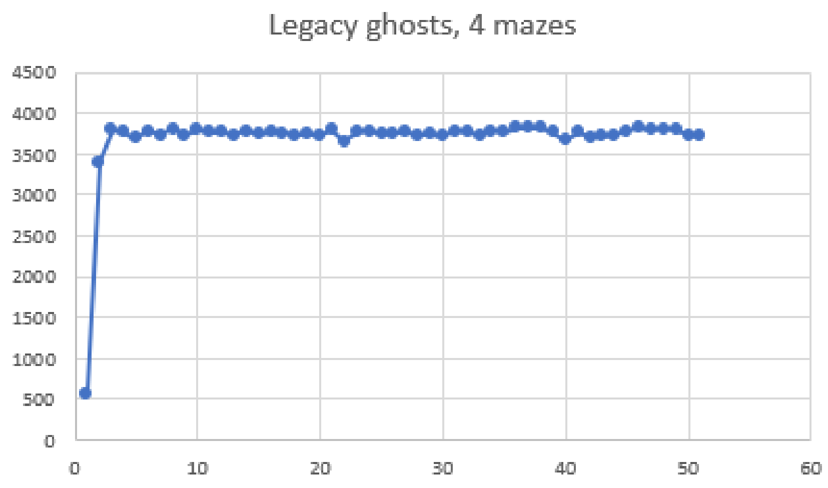


Figure 4.2: Average per generation: Legacy ghosts, 4 mazes

The main takeaway from this graph is not only a higher average per generation but also how the graph plateaued after the third generation. The reason for this can be seen after investigating the best result from this run. The best tree being
( 2ifindanger ( runaway 2istopowersafe (
moveToEdibleGhost moveToFood ) ) )
where the average result from 10 runs was 10923. After running this tree for 1000 times for statistical purposes, the highest result recorded was 20400, which is a very high result, however, there are very specific reasons for it.

From watching the best scoring game as well as analysing the code, the strategy of this tree was to keep itself safe as long as it is in danger and look to get normal pills. If there is a power pill nearby, the Pacman would eat it and unlike the simple method best result, this controller would chase the edible ghosts and get a lot of points. This resulted in a substantial amount of points being collected by the controller, however, what led to it dying multiple times is the disregard for the non-edible ghosts that spawned after being eaten. The timing of the game allows for such ghosts to appear before the not eaten blue ghosts turn back to the normal state. Thus, this Pacman would get eaten by spawned ghosts and not allow it to get past the third maze. Even though the controller could not get past the third maze, the number of points it was able to score was very significant and looked very close to an optimal strategy.

Now that we have established the results achievable by this classic setup, we can employ a different sort of training to see whether we can develop some other behaviours. To do this, we are going to take the approach of Alhejali and Lucas and train the controller on just the first maze instead of four. This led to comparable results in their research and can make our controllers benefit as well. Furthermore, to establish some more pronounced behaviours, we are going to train some controllers against a different type of ghosts. As we already know, the game is inherently very hard due to different strategies each of the four ghosts employ, however by making them behave similarly, we can make the game more predictable for the controller and get better results. The choice has fallen on the Aggressive ghosts controller, which makes all of the ghosts behave in a very hostile manner. While it may seem to be harder to beat, as the ghosts are going to constantly chase Pacman, the best controllers turned this behaviour into their advantage. Let us look at the averages of the tests against Aggressive ghosts in 1 maze and 4 maze tests.

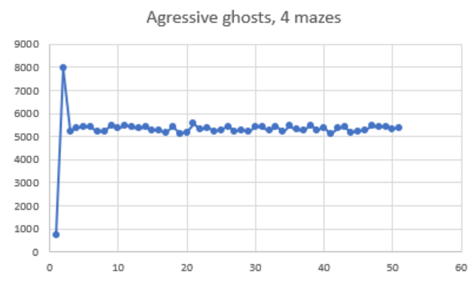


Figure 4.3: Average per generation: Aggressive ghosts, 4 mazes

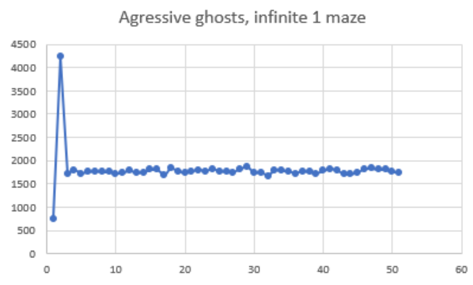


Figure 4.4: Average per generation: Aggressive ghosts, 1 maze

Here we can observe a very strong performance on average coming from the controllers that trained in 4 mazes. In 1 maze, however, the performance was not as good, but the controllers on average beat the maze at least one time, which is around the same as simple method approach (Figure 4.1), but evidently, restriction to one maze hurt the fitness output of this run.

When we look at the best trees, they are both identical in their approach to the game and have similar maximum scores over 1000 runs. The following trees of 1 maze and 4 mazes respectively are

( 2istopowersafe ( distTo3rdGhost moveFromNonEdibleGhost ) )

( 2ifindanger ( moveFromNonEdibleGhost distToPower ) )

These scripts have exploited one part of the game that was not known until the code for the game was analysed to a deeper extent. This part of the game is the time limit of 4000 steps, which limited the time per round. When this limit was reached, the game would advance to the next level. Therefore, the strategy that these scripts found was to stick themselves into a wall and wait for the ghosts to come close. When the ghosts are close enough the script tells Pacman to run away and since all of the ghosts are aggressive they follow Pacman without any deviations.

This yielded results of 28210 from 1 maze trained and 32840 from 4 maze trained trees when going up against aggressive ghosts in 4 mazes. Therefore, we see that having a very simple behaviour of the ghosts can become an advantage for genetic programming technique. However, whilst delivering the best scores so far, both of these programs fell short when going up against 4 mazes and Legacy ghosts, scoring 9220 and 10790 for 1 maze and 4 mazes trained respectively. When looking through the footage, it can be seen that these algorithms do not account for the other ghosts in the maze and only take action to get away from the most aggressive one, which behaves similarly to the ghosts they trained against. Thus, we can safely conclude, that while we can achieve better results against less complex ghosts, showing higher flexibility of the methods used and the library being able to adapt, we cannot use these outputs to beat our original high complexity setup.

Lastly, the choice of selecting only the best trees in terms of fitness, for crossover and reproduction has to be addressed, now that better methods are available. Below are the average statistics for two runs, where every tree from every generation has a chance of participating in the crossover, with one run going against Legacy ghosts in 4 mazes and the other against Aggressive ones also in 4 mazes.

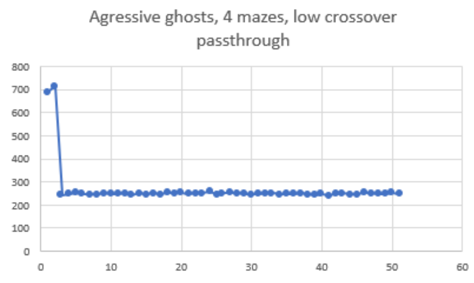


Figure 4.5: Average per generation: Aggressive ghosts, 4 mazes

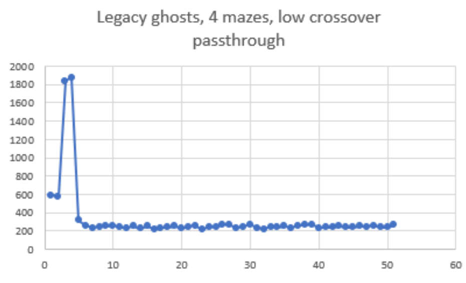


Figure 4.6: Average per generation: Legacy ghosts, 4 mazes

Here is an evident rise of average fitness per generation in both runs when it comes to the 2nd and 3rd generation, however, there is a significant drop in quality of trees, which is the same flaw that appeared when working with simpler methods. While both runs produced best results that can compete against simple methods best run, they are still worse compared to their counterparts, produced in a stricter environment (Figures 4.2 and 4.3). The reason for this is as explained above caused by low-quality trees introducing very harmful nodes into better trees, causing them to perform much worse and bringing the whole of the population down. Even when given the roulette system with a custom probability distribution, there is still a significant amount of them that make it into the next evolutions.

In conclusion, this investigation has produced results of various success, with some very potent trees generated, which proves that the library built is functional and can provide a user with genetic programming techniques suitable for their experiments. In the next section of this chapter, these results are going to be checked against previous research, for the behaviours generated as well as fitness results. Furthermore, checking against results that the creators of this game package achieved is crucial to gauge the performance of the library's output.

## 4.3 Results comparison

When comparing the results achieved against Koza, it has to be noted that 4000 points that were scored by his best controller came from eating fruit, that appears randomly in the maze. This part of the game is absent in Ms. Pacman, therefore these points are not part of the comparison. With 5220 points scored by Koza's controller considering the adjustment, it is beaten by the simple methods controller of this library, as it has scored 6460. While, it can be argued that our controller might not have beaten, or cannot be truly compared to Koza's, it can be stated with confidence that these results are of the same calibre. Furthermore, when we are looking at a more complex set of methods and the outcome of running them, Koza's controller seems to be underperforming, as it struggled to get all of the ghosts when they were blue, compared to our complex controller that was trained against Legacy ghosts in 4 mazes. Thus, we have established that using Koza's framework of the application of genetic programming and his methods of battling Pacman style games, we were able to get the library to produce good results that within a margin of error are very similar to what he achieved in his research.

Moving on to more relevant research, Alhejali and Lucas proved to have similar plateau in their average fitness per generation graphs, however, their results proved to be more successful, with the highest score achieved in a 4 maze game being 44560. This fitness, being very impressive and higher than the best result achieved by our controllers, can be explained by the deeper complexity of some of the algorithms written by the researchers. Furthermore, consistency of their controller is not high, due to their use of random distance values in the decision calculations, thus the average result being only at 16014 for their best tree, whereas the average for our best controller is 7191, being closer to the best result standing at 20400. However, higher consistency is due to the setup of the methods excluding randomness in the calculations, not a higher quality of methods. Considering the above, this library has not outperformed the researchers, whilst generating more consistent results using weaker search algorithms. Nevertheless, the game is difficult and trying to beat random behaviour of some of the ghosts with randomness in the controller can help more than consistency.

Lastly, looking at the creators of the game, there results turned out to be underwhelming, as they have only found success using Grammatical Evolution against Random ghosts, which does not represent the true game. The average being around 2000 points and the best-evolved controller showing only 12350 against Legacy style ghosts. This shows, that the results from our controllers are above what can be generated by other branches of genetic programming, and on average are outperforming the hardest game setup during the evolution cycle.

In conclusion, the results generated in this investigation, while not being the best in research, are of comparable success and show that the library can functionally produce good decision trees that can be studied to develop better AI software. Furthermore, this investigation has shown a correlation of more complex methods yielding better results, as well as how limiting the evolution to only the best trees leading to more consistent and fitter evolution cycles.

# Chapter 5

# Discussion and Conclusion

This chapter will provide for the final analysis of the successes and deficiencies that this project has. It will provide a deeper look into future work that can be done to enhance the library built and what other approaches could have been done in the application stage.

## 5.1 Discussion

Overall it can be said that the project has met its initial goals within the scope laid out and delivered comparable results to the previous literature. One of the major successes that were achieved was an ability to create custom code, that could be run independent of the methods' domain and it could be outputted for future use. Koza's framework outline became a crucial part of getting to understand how Genetic programming works and gave clear goals during the development process. As stated in the introduction this library was not built to compete with commercially available genetic programming libraries but exists as a proof of concept that you can still make Java output and run custom code that was randomly generated, without any compilation errors. Furthermore, making the code adapt and create fitter programs, proved to be a moderate success.

The main deficiencies of the software built is lack of support for genetic algorithms, which were outside of the library's scope, but are present in most of the available libraries that specialise in genetic programming. Furthermore, the limitation of the library to support a flexible amount of input methods is considered as a major downside. Due to the structure of Java, it was not possible to make as many methods as the user wanted, to be included in the executor of the decision trees. This leads to a problem, where if the user wants to have more methods than available in the interface *Executable*, some inner parts of the code have to be changed accordingly.

The application investigation of the project has shown very good results and proved that all of the functionality available to the user is working as expected. In a time span of 4 hours the library was able to test 51 generations of 1400 trees, with each test consisting of 10 runs per tree. However, whilst functional the investigation did not deliver the best results, especially compared to Alhejali and Lucas, who implemented similar methods to a very closely specified game. Moreover, new strategies that were created to test the Pacman controller, going against simpler ghosts, did not achieve better fitness in the classic game but became a proving ground to a behaviour discrepancy hypothesis.

Lastly, a part of the initial plan in terms of the application was not successfully developed. As stated in the project management section, application to other games, was not possible, as software that was found did not work as intended. The games would have to be rebuilt to make them serviceable, which was not possible in the time frame of the project. These problems did not allow for building a tournament based system, where the fitness function would be determined by having decision trees compete against each other. This would require only one class to be built and a functional 1 vs 1 player game. If there was more time for this project, this development would be of the highest priority.

## 5.2 Conclusion

In conclusion, this project has at least in part achieved what it has set out to deliver. The created library employs genetic programming techniques in Java and can create custom code that is functional and can be run on its own. The library delivers signs of evolution in the fitness of its programs seen in the research on the topic and can compete in terms of its best results against other AI decision scripts. While some of the initial goals have not been met, due to technical difficulties as well as time constraints, the library is a showcase of what can be achieved by Java and can become a good introduction to new researchers in the topic of genetic programming. Lastly, the relative ease of use of the library enhances its quality as a first stepping stone in understanding the basics of genetic programming.

# Bibliography

[1] A. M. Alhejali and S. M. Lucas. Evolving diverse ms. pac-man playing agents using genetic programming. *IEEE*, 2010.

[2] M. F. Brandstetter and S. Ahmadi. Reactive control of ms. pac man using information retrievalbased on genetic programming. *IEEE*, 2012.

[3] J. R. Koza. *Genetic Programming: On the programming of computers by means of natural selection.* MIT Press, 1996.

[4] H. L. Mantecon, J. S. Cremades, and J. M. T. Garrigos. A pac-man bot based on grammatical evolution. *Dep. Ingenieria del Software e Inteligencia Artificial Universidad Complutense de Madrid*, 2016.

[5] D. Robles and S. M. Lucas. A simple tree search method for playing ms. pac-man. *2009 IEEE Symposium on Computational Intelligence and Games*, pages 249–255, 2009.

# Chapter 6

# Appendix

The address of the git repository containing the source code:
https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2018/nxk825

The directory Genetic within the source code directory holds all of the Genetic Programming library, built for this project. All of the code in this directory was developed by the writer of this report.

Directory Results, holds of the statistical data collected by the writer during evolutionary investigation, as well recordings of the best runs from the controllers produced.

Directories pacman and parser are part of the project developed by a group of Spanish students [4]. All of the runnable files in pacman directory were written by the writer of this report, with exception of Executor, where only the main method is adapted by the writer.

GeneticController and SimpleController in the package controllers, are written by the writer employing the game's Abstract class Controller. This allowed the writer to control the game.

To run the test simulation, either Executor, ExecutorAgressive or ExecutorOne can be run.

To see visual representation of the best games that were produced in the evolutionary tests, the other runnable classes can be executed, where inside they refer to text files holding those recordings. These can be changed to the ones that are commented out.

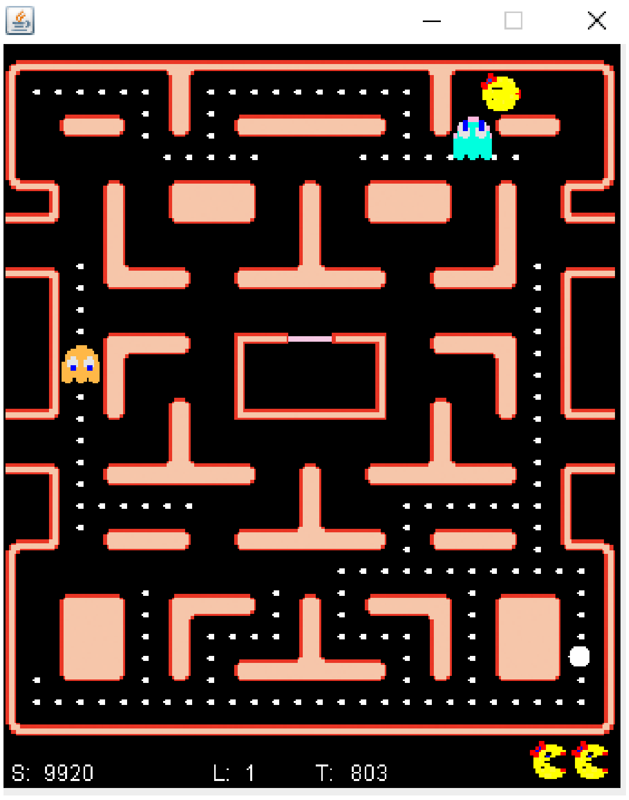


Figure 6.1: Example of the game running, controlled by the library

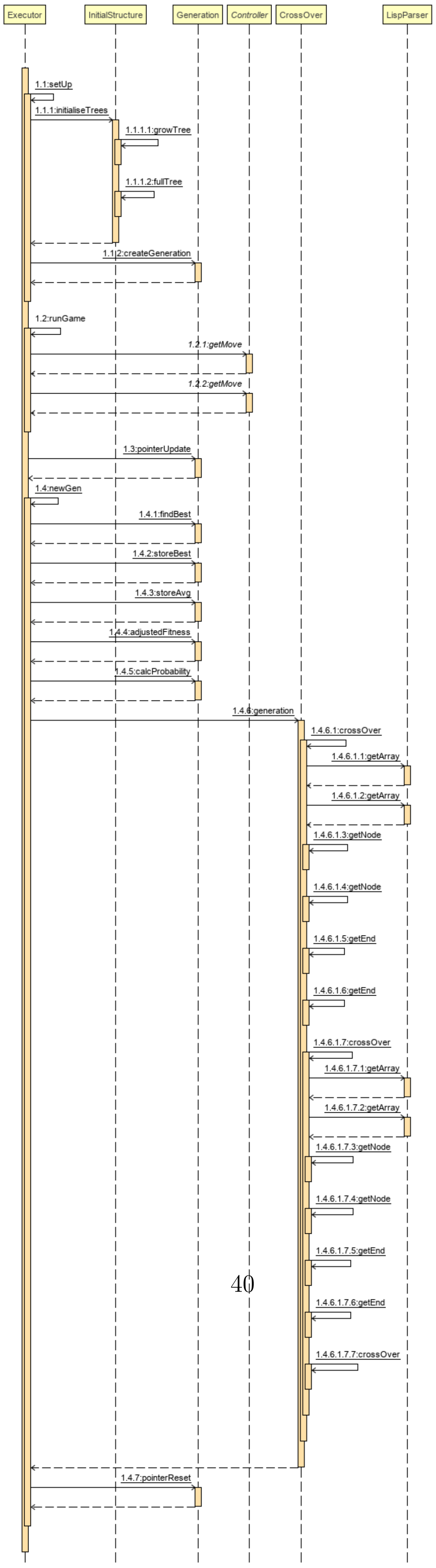


Figure 6.2: Generation cycle sequence diagram

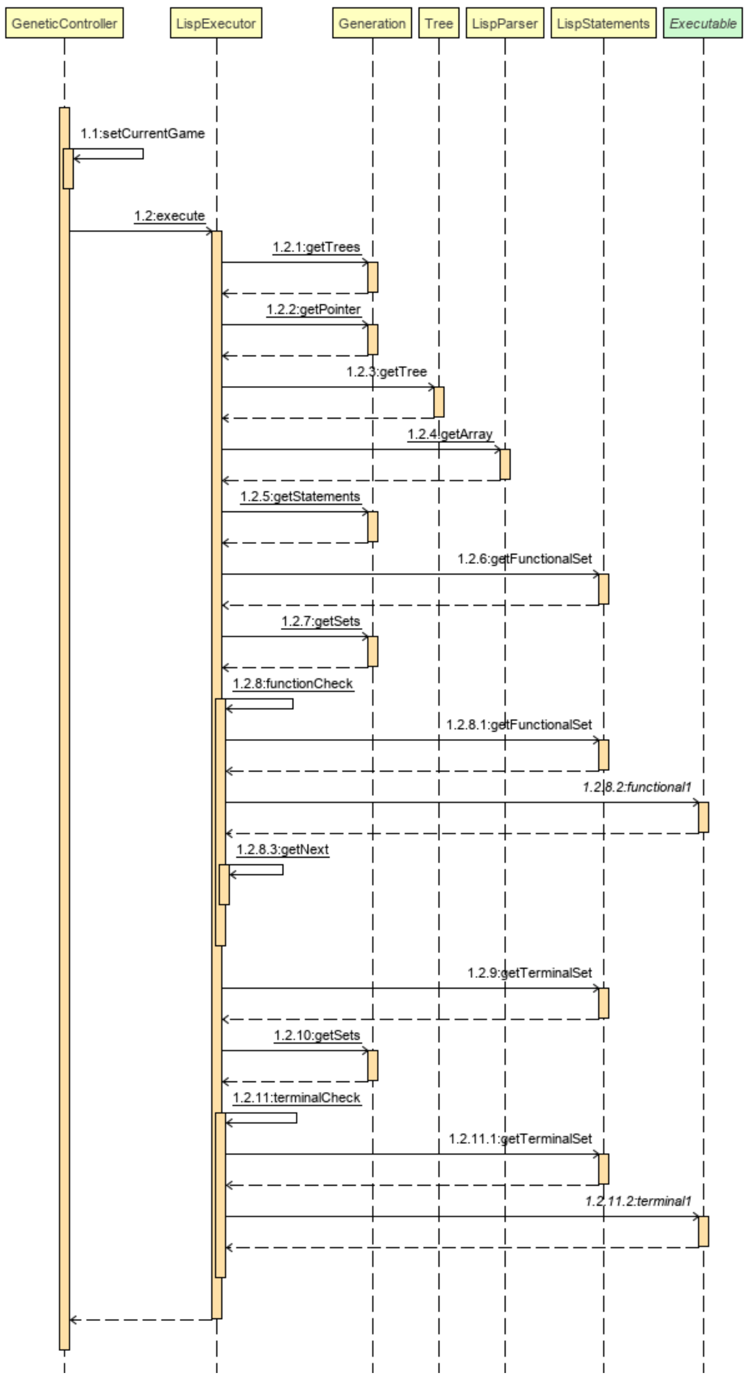
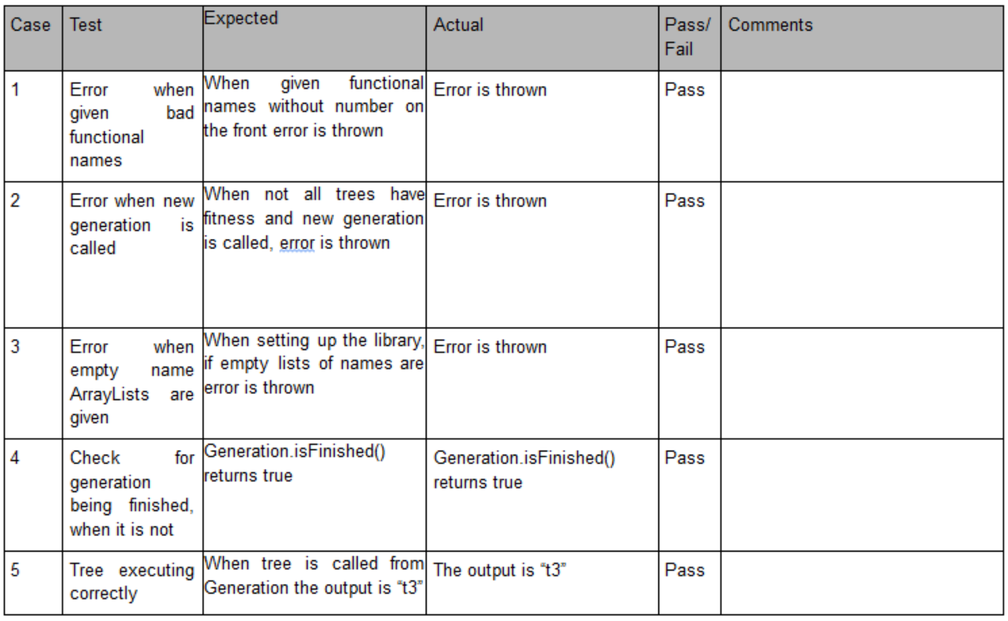
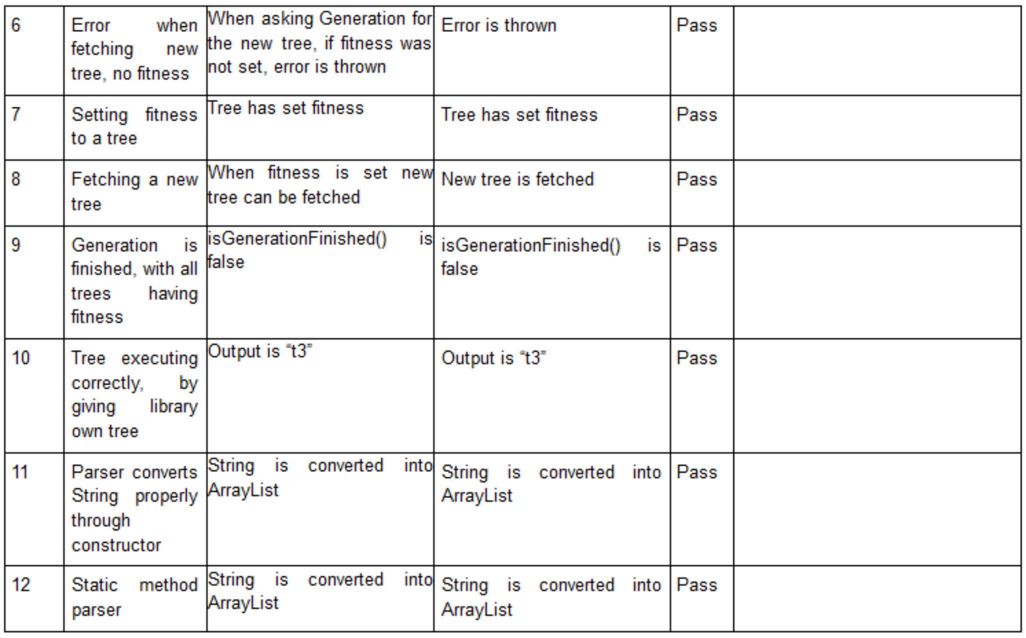


Figure 6.3: Decision tree execution sequence diagram



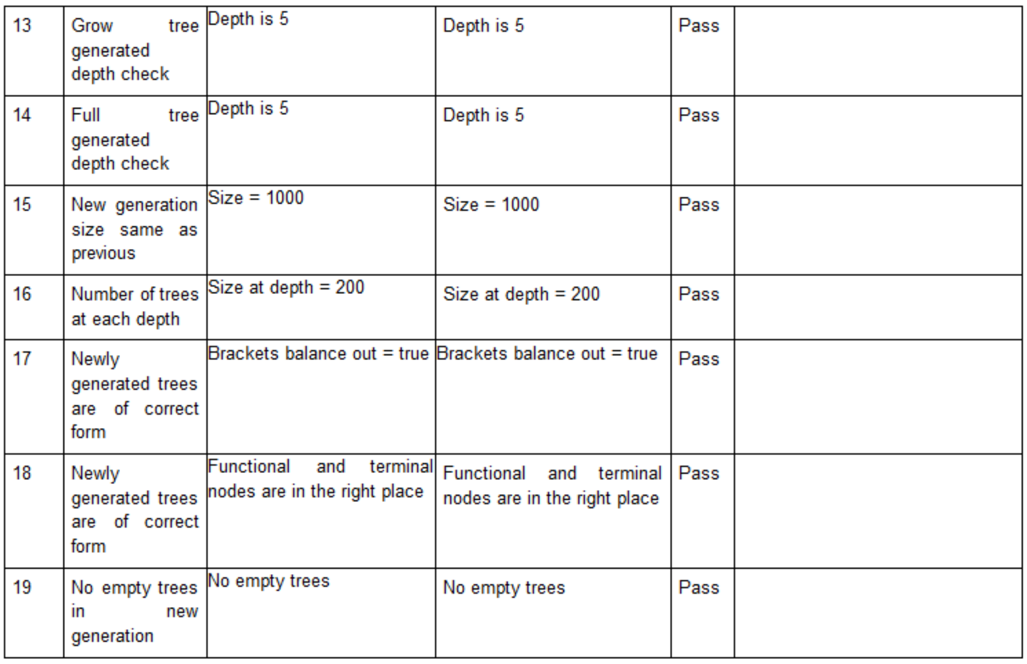
| Case | Test | Expected | Actual | Pass/ Fail | Comments |
|---|---|---|---|---|---|
| 1 | Error when given bad functional names | When given functional names without number on the front error is thrown | Error is thrown | Pass | |
| 2 | Error when new generation is called | When not all trees have fitness and new generation is called, error is thrown | Error is thrown | Pass | |
| 3 | Error when empty name ArrayLists are given | When setting up the library, if empty lists of names are error is thrown | Error is thrown | Pass | |
| 4 | Check for generation being finished, when it is not | Generation.isFinished() returns true | Generation.isFinished() returns true | Pass | |
| 5 | Tree executing correctly | When tree is called from Generation the output is “t3” | The output is “t3” | Pass | |

Figure 6.4: Test table 1



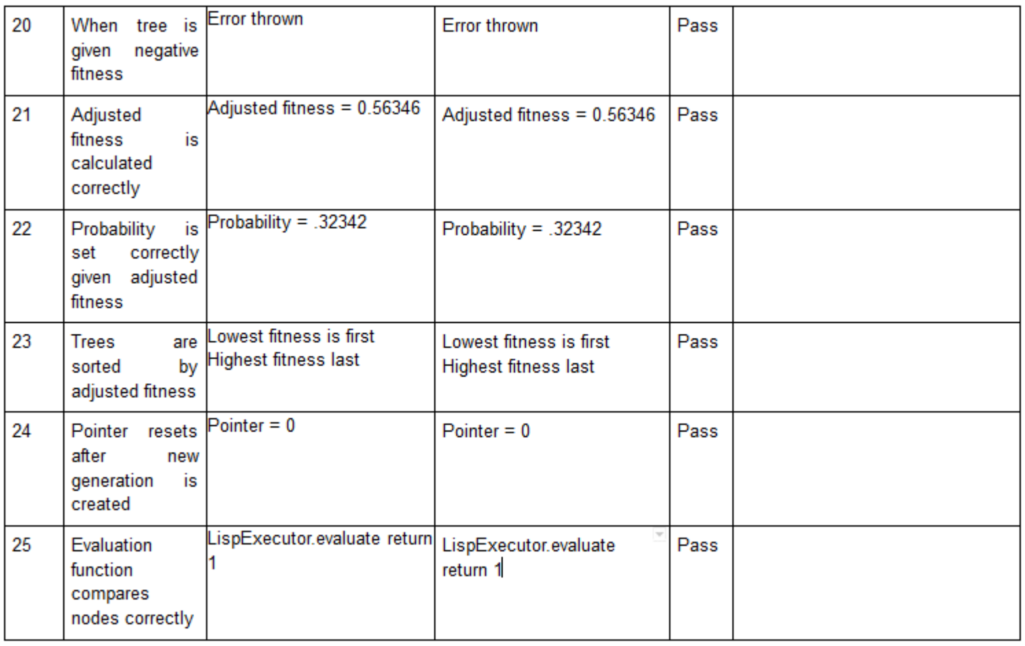
| | | | | | |
|---|---|---|---|---|---|
| 6 | Error when fetching new tree, no fitness | When asking Generation for the new tree, if fitness was not set, error is thrown | Error is thrown | Pass | |
| 7 | Setting fitness to a tree | Tree has set fitness | Tree has set fitness | Pass | |
| 8 | Fetching a new tree | When fitness is set new tree can be fetched | New tree is fetched | Pass | |
| 9 | Generation is finished, with all trees having fitness | isGenerationFinished() is false | isGenerationFinished() is false | Pass | |
| 10 | Tree executing correctly, by giving library own tree | Output is “t3” | Output is “t3” | Pass | |
| 11 | Parser converts String properly through constructor | String is converted into ArrayList | String is converted into ArrayList | Pass | |
| 12 | Static method parser | String is converted into ArrayList | String is converted into ArrayList | Pass | |

Figure 6.5: Test table 2



| | | | | | |
|---|---|---|---|---|---|
| 13 | Grow tree generated depth check | Depth is 5 | Depth is 5 | Pass | |
| 14 | Full tree generated depth check | Depth is 5 | Depth is 5 | Pass | |
| 15 | New generation size same as previous | Size = 1000 | Size = 1000 | Pass | |
| 16 | Number of trees at each depth | Size at depth = 200 | Size at depth = 200 | Pass | |
| 17 | Newly generated trees are of correct form | Brackets balance out = true | Brackets balance out = true | Pass | |
| 18 | Newly generated trees are of correct form | Functional and terminal nodes are in the right place | Functional and terminal nodes are in the right place | Pass | |
| 19 | No empty trees in new generation | No empty trees | No empty trees | Pass | |

Figure 6.6: Test table 3



| | | | | | |
|---|---|---|---|---|---|
| 20 | When tree is given negative fitness | Error thrown | Error thrown | Pass | |
| 21 | Adjusted fitness is calculated correctly | Adjusted fitness = 0.56346 | Adjusted fitness = 0.56346 | Pass | |
| 22 | Probability is set correctly given adjusted fitness | Probability = .32342 | Probability = .32342 | Pass | |
| 23 | Trees are sorted by adjusted fitness | Lowest fitness is first Highest fitness last | Lowest fitness is first Highest fitness last | Pass | |
| 24 | Pointer resets after new generation is created | Pointer = 0 | Pointer = 0 | Pass | |
| 25 | Evaluation function compares nodes correctly | LispExecutor.evaluate return 1 | LispExecutor.evaluate return 1 | Pass | |

Figure 6.7: Test table 4