A project completed as part of the requirements for the

BSc (Hons) Computer Science

entitled

Creating Artificial Intelligence Agents for Spelunky using an Artificial Neural Network and a Genetic Algorithm

by

James Alan Kyte

james_kyte_36@hotmail.com

in the years 2014 - 2015
This project builds upon the original open source Spelunky source code for GameMaker 8 created by Derek Yu of Mossmouth studios as well as the Spelunkbots API that was created by Daniel Scales as part of his dissertation project with the University of Derby.

Abstract

The research in this paper will focus on the use of artificial neural networks and genetic algorithms in creating an artificial intelligence (AI) agent that is able to successfully handle the various problem domains that exist within an electronic game known as Spelunky. A problem domain being the specific area of challenge within Spelunky such as the domain of terrain navigation, the domain of combat, the domain of planning and others (Scales and Thompson, 2014).

This paper begins by giving an introduction which consists of the hypothesis of the paper as well as why this hypothesis is possible and worth researching. The introduction then lays out the aim and objectives of this paper that need to be met so a valid assessment of the accuracy of the hypothesis can be made.

A literature review is then conducted which details the concepts of artificial neural networks as well as genetic algorithms. It will also cover their implementation and an example of work they have been used in. Using games as a test bed for AI research will also be reviewed as well why Spelunky should specifically be used.

A methodology follows the literature review detailing how an artificial neural network will be implemented into Spelunky as well as what experimentations will be ran and what data will be looked for in the experiments as well as how data is to be retrieved from the experiments.

The results of testing are then displayed and an analysis of the results is conducted. A discussion is then conducted which details some of the issues that have been experienced in the research and what the testing data is telling us about the system that has been implemented. Conclusions are then made from all the gathered information about the performance of the AI agents created to tackle Spelunky, what limitations they have, how well met the aim and objectives were and finally the accuracy of the hypothesis of this research will be assessed.
Acknowledgements

I would like to thank University of Derby lecturer Dr. Tommy Thompson for the opportunity to conduct the research in this paper as well as for his help and support during its creation.

I would also like to thank University of Derby Students Daniel Scales and Jordan Rowe. Thank you Daniel for creating the Spelunkbots API that I used as a part of my project as well as your support while learning how to use it and help to fix any issues that occurred. Thank you Jordan for also helping me when issues arose with the API and for helping me to understand how to use it.
# Table of Contents

1.0 Introduction ......................................................................................................................... 12

1.1 Hypothesis ............................................................................................................................... 12

1.2 Project Rationale ...................................................................................................................... 12

1.3 Project Aim and Objectives .................................................................................................... 14

2.0 Literature Review ..................................................................................................................... 16

2.1 Introduction ............................................................................................................................... 16

2.2 Artificial Neural Networks ...................................................................................................... 16

  2.2.1 Concept .............................................................................................................................. 16

  2.2.2 Implementation .................................................................................................................. 16

  2.2.3 Flavours of Artificial Neural Network .............................................................................. 17

2.3 Genetic Algorithm ...................................................................................................................... 18

  2.3.1 Concept ............................................................................................................................ 18

  2.3.2 Implementation ................................................................................................................ 18

2.4 Example of Use ......................................................................................................................... 19

  2.4.1 Example ........................................................................................................................... 19

  2.4.2 Artificial Neural Network Implementation ....................................................................... 19

  2.4.3 Genetic Algorithm Implementation ............................................................................... 20

  2.4.4 Evaluation ........................................................................................................................ 20

2.5 Using Games for AI Research ................................................................................................. 22

  2.5.1 Challenges Games Provide ............................................................................................... 22

  2.5.2 History of AI in Games Research .................................................................................... 23

2.6 Spelunky’s Problem Domain .................................................................................................... 25

2.7 Why Spelunky ............................................................................................................................ 27

  2.7.1 Criteria ............................................................................................................................. 27

  2.7.2 Domain ............................................................................................................................. 27

  2.7.3 Domain Extensibility and Customisability ..................................................................... 27
2.7.4 Domain Recognisability ................................................................. 28
2.8 Conclusions .................................................................................. 28
   2.8.1 Key Issues ............................................................................... 29
   2.8.2 Refined Research Questions ...................................................... 29

3.0 Methodology .................................................................................. 31
   3.1 Introduction .................................................................................. 31
   3.2 Research Methods and Development Methods ................................. 31
      3.2.1 Research Methods .................................................................. 31
      3.2.2 Development Methods ............................................................ 31
   3.3 System Design ............................................................................. 32
      3.3.1 Artificial Neural Network Design ............................................. 32
      3.3.2 Genetic Algorithm Design ....................................................... 32
      3.3.3 Data Generation Design .......................................................... 33
   3.4 Experimentation Methodology ......................................................... 34
   3.5 System Construction ..................................................................... 35
      3.5.1 Artificial Neural Network ....................................................... 35
      3.5.2 Genetic Algorithm ................................................................. 37
      3.5.3 Data Generation .................................................................... 39
   3.6 Data Analytics Method .................................................................. 40
   3.7 Limitations .................................................................................. 40
      3.7.1 Testing Size .......................................................................... 40
      3.7.2 Learning Capability ............................................................... 41
   3.8 Conclusion .................................................................................. 41

4.0 Findings and Analysis .................................................................... 42
   4.1 Introduction .................................................................................. 42
   4.2 Data Caveat .................................................................................. 42
   4.3 Experiment Levels ........................................................................ 44
      4.3.1 Experiment Level 1 ................................................................. 45
4.3.2 Experiment Level 2 ................................................................. 46
4.3.3 Experiment Level 3 ................................................................. 46
4.3.4 Experiment Level 4 ................................................................. 47
4.3.5 Experiment Level 5 ................................................................. 47
4.3.6 Experiment Level 6 ................................................................. 48
4.3.7 Experiment Level 7 ................................................................. 49
4.3.8 Experiment Level 8 ................................................................. 49
4.3.9 Experiment Level 9 ................................................................. 50
4.3.10 Experiment Level 10 ............................................................. 51

4.4 Generation Analysis .................................................................... 51
4.4.1 Level 1 ..................................................................................... 52
4.4.2 Level 2 ..................................................................................... 53
4.4.3 Level 3 ..................................................................................... 54
4.4.4 Level 4 ..................................................................................... 55
4.4.5 Level 5 ..................................................................................... 56
4.4.6 Level 6 ..................................................................................... 57
4.4.7 Level 7 ..................................................................................... 58
4.4.8 Level 8 ..................................................................................... 59
4.4.9 Level 9 ..................................................................................... 60
4.4.10 Level 10 ................................................................................ 61

4.5 Parent Analysis ............................................................................ 61
4.5.1 Parent Set 1 .............................................................................. 63
4.5.2 Parent Set 2 .............................................................................. 63
4.5.3 Parent Set 3 .............................................................................. 64
4.5.4 Parent Set 4 .............................................................................. 65
4.5.5 Parent Set 5 .............................................................................. 65
4.5.6 Parent Set 6 .............................................................................. 66
4.5.7 Parent Set 7 .............................................................................. 67
4.5.8 Parent Set 8 ............................................................................................................. 68
4.5.9 Parent Set 9 ............................................................................................................. 69
4.5.10 Parent Set 10 ......................................................................................................... 70
4.6 Conclusions ................................................................................................................. 71

5.0 Discussion .................................................................................................................... 72
5.1 Introduction .................................................................................................................... 72
5.2 Use of an Artificial Neural Network and Genetic Algorithm in Spelunky .................... 72
5.3 Use of Spelunky as a Research Domain ....................................................................... 73
5.4 Agent Performance ...................................................................................................... 73
5.5 Conclusion .................................................................................................................... 73

6.0 Conclusions and Recommendations ........................................................................... 75
6.1 Key Issues ...................................................................................................................... 75
6.2 Conclusions .................................................................................................................... 76
6.3 Recommendations ........................................................................................................ 78

References ......................................................................................................................... 80

Bibliography ....................................................................................................................... 82

Appendix A: Time Limit ‘bot’ Source Code ...................................................................... 83

Appendix B: Software User Guide ..................................................................................... 84
B.1 Controls ......................................................................................................................... 84
B.2 Testing a Generation ..................................................................................................... 87
B.3 Testing a Parent Set ....................................................................................................... 89
B.4 Creating Data Graphs ................................................................................................... 90
B.5 Function and Script Call Order .................................................................................... 91
Table of Figures

Figure 1.0: A Simple Multilayer Feed-Forward Neural Network ........................................17
Figure 2.1: Co-Evolution Test Results, Population Fighting Each Other........................21
Figure 2.2: Co-Evolution Test Results, Population Fighting Standard Game Enemies ....21
Figure 3.1: Spelunker Using a Bomb to Destroy Terrain .............................................25
Figure 3.2: Spelunker Using a Rope to Reach Higher Terrain .......................................26
Figure 4.0: TanH activation function equation using the mathematical constant $e$ .....36
Figure 5.1: Level Entrance .........................................................................................44
Figure 5.2: Level Exit .................................................................................................44
Figure 5.3: Terrain ......................................................................................................44
Figure 5.4: Background ..............................................................................................44
Figure 5.5: Spikes ........................................................................................................44
Figure 6.1: Experiment Level 1 ..................................................................................45
Figure 6.2: Experiment Level 2 ..................................................................................46
Figure 6.3 Experiment Level 3 ....................................................................................46
Figure 6.4: Experiment Level 4 ..................................................................................47
Figure 6.5 Experiment Level 5 ....................................................................................47
Figure 6.6: Experiment Level 6 ..................................................................................48
Figure 6.7 Experiment Level 7 ....................................................................................49
Figure 6.8 Experiment Level 8 ....................................................................................49
Figure 6.9: Experiment Level 9 ..................................................................................50
Figure 6.10: Experiment Level 10 ..............................................................................51
Figure 7.1: Level 1 Results ..........................................................................................52
Figure 7.2: Level 2 Results ..........................................................................................53
Figure 7.3: Level 3 Results ..........................................................................................54
Figure 7.4: Level 4 Results ..........................................................................................55
Figure 7.5 Level 5 Results ..........................................................................................56
Figure 7.6: Level 6 Results ........................................................................................................57
Figure 7.7: Level 7 Results ........................................................................................................58
Figure 7.8: Level 8 Results ........................................................................................................59
Figure 7.9: Level 9 Results ........................................................................................................60
Figure 7.10: Level 10 Results .....................................................................................................61
Figure 8.1: Parent Variability .....................................................................................................62
Figure 8.2 Parent Performance Set 1 .......................................................................................63
Figure 8.3: Parent Performance Set 2 .......................................................................................63
Figure 8.4: Parent Performance Set 3 .......................................................................................64
Figure 8.5: Parent Performance Set 4 .......................................................................................65
Figure 8.6: Parent Performance Set 5 .......................................................................................65
Figure 8.7.1: Parent Performance Set 6, Levels 1-5 ..................................................................66
Figure 8.7.2: Parent Performance Set 6, Level 6 .......................................................................66
Figure 8.8.1: Parent Performance Set 7, Levels 1-5 .................................................................67
Figure 8.8.2: Parent Performance Set 7, Levels 6 and 7 ...........................................................67
Figure 8.9.1: Parent Performance Set 8, Levels 1-5 .................................................................68
Figure 8.9.2: Parent Performance Set 8, Levels 6-8 .................................................................68
Figure 8.10.1: Parent Performance Set 9, Levels 1-5 .................................................................69
Figure 8.10.2: Parent Performance Set 9, Levels 6-9 .................................................................69
Figure 8.11.1: Parent Performance Set 10, Levels 1-5 ...............................................................70
Figure 8.11.2: Parent Performance Set 10, Levels 6-10 .............................................................70
Figure 9.0: ‘bot’ Source Control ............................................................................................83
Figure 10.1: ANN Controls ......................................................................................................85
Figure 10.2: Maximum Tests Control ......................................................................................85
Figure 10.3: Population Related Controls ...............................................................................85
Figure 10.4: Number of Generations Control .........................................................................86
Figure 10.5: Number of Parents Control ..................................................................................86
Figure 10.6: Level Controls ......................................................................................................86
Figure 10.7: Data Production Control........................................................................87
Figure 11.1: Saved Parents File ..................................................................................88
Figure 11.2: Score Room .........................................................................................89
Figure 12.1: oPlayer1 Room Start Scripts .................................................................91
Figure 12.2: PlayerChoice Agent Control .................................................................92
Figure 12.3: Operations after Character Death .........................................................93
Figure 12.4: Operations after Level Completion .......................................................95
Table of Tables

Table 1.0: Level Objects .........................................................................................................................................44
1.0 Introduction

The aim of this introductory section will be to inform the reader of the research that will take place within this paper, why this research should take place as well as the hypothesis that this paper will work towards and the aim and objectives that will be used to work towards assessing the accuracy of this hypothesis.

1.1 Hypothesis

The research in this paper will focus on the use of artificial neural networks and genetic algorithms in creating an artificial intelligence (AI) agent that is able to successfully handle the various problem domains that exist within an electronic game known as Spelunky. A problem domain being the specific area of challenge within Spelunky such as the domain of terrain navigation, the domain of combat, the domain of planning and others (Scales and Thompson, 2014).

My hypothesis therefore is I believe that artificial intelligence agents can be created that can successfully play Spelunky and act as solutions to its problem domains through the use of an artificial neural network and a genetic algorithm.

1.2 Project Rationale

This rationale will serve two purposes, the first being why I believe this hypothesis is possible and why this hypothesis and area of study deserves to be studied.

Stating with the use of games for AI creation it can be seen that there have been a number of games used with the purpose of developing artificial intelligence agents to play the games such as, Mario, Ms. Pac-Man, Starcraft and Unreal Tournament (Thompson, 2014). Other games have been created specifically to experiment with AI use such as NERO (NERO) and Galactic Arms Race (GAR). Notably Mario, Ms.Pacman and Unreal Tournament were actually used in competitions which were aimed at the AI community to develop AI that could play the games. There also exists events that specifically study the creation of AI systems that can play games such as the Computational Intelligence and Games conference (CIG) and the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE).

Moving on the use of artificial neural networks and genetic algorithms in games it can be seen in the past there has been work in using these to create AI that can handle specific problem domains such as in 2001 where they were used to play a clone of the Pac-Man game
(Thompson, 2014) and in 2003 where an AI learned to play Pac-Man using an evolutionary learning approach (Thompson, 2014).

More instances of their use are in works such as Tommy Thompson’s EvoTanks in 2005 (Thompson, T[b]) where they were used to create agents that would fight either each other or game controlled enemies. Galactic Arms Race which used an artificial neural network and an evolutionary algorithm to create procedurally generated content for the play. The developers of Galactic Arms Race published a paper in 2009 at the Computational Intelligence in Games conference about the inner workings of the game in the (GAR). NERO which involved using a artificial neural network and an evolutionary algorithm that taught agents that the player created to perform specific tasks such as combat or territory acquisition. (NERO) This was published in 2005.

Another point to this rationale is while it has been confirmed that games can be used for the testing of AI systems, why specifically should Spelunky be chosen.

Based on the information expressed by Scales and Thompson in their paper entitled Spelunkbots API - An AI Toolset for Spelunky (Scales and Thompson, 2014) it is stated that they believe that due to the game mechanics they discuss in the paper and the overall design of the game they believe that Spelunky provides and interesting challenge to AI research in games. They also state that they believe Spelunky “…presents an evolution from recent competition domains such as Ms Pac-Man and Super Mario Bros.” because like with Ms Pac-Man Spelunky shares traits such as evading nearby enemies which can display different behaviours, however this is expanded by the need for players to “…Strategise their navigation and also the destruction they inflict upon the environment…” With regards to super Mario bros Spelunky shares the traits of being a 2D platformer where running and jumping is used to navigate the environment. A way this expands from super Mario bros but is not expressly stated by Scales and Thompson is that in Mario you only run to the right hand side where as in Spelunky you can travel left and right and must actively search for the end of a level providing a level of exploration that Super Mario Bros does not have.

From this rationale we have learnt that games have been used previously in the development of AI agents and systems to play games or be a part of a game. That artificial neural networks and genetic algorithms have been used within previous pieces of work and have allowed the creation of agents that can learn and operate within an environment. We have also learnt that according the Scales and Thompson that Spelunky does provide an interesting space in which to test AI systems and agents.
From the this information we have obtained we can now see why the hypothesis given in section 1.1 is possible as Spelunky has good domains in which we can test AI agents and also that artificial neural networks and genetic algorithms show the capability of controlling agents and allowing them to learn to become better at dealing with tasks. We can also see why this area of study as well as the hypothesis is worth studying because it allows us to contribute towards the state of the art within AI research and develop more tools to make more sophisticated AI capable of handling more complex tasks.

1.3 Project Aim and Objectives

Based on the hypothesis the aim of this will be to develop a system that incorporates an artificial neural network and a genetic algorithm with the purpose of creating agents that can play Spelunky and find a solution to its problem domains, as well as produce agent performance data that can be used to assess the accuracy of the hypothesis. To achieve this aim the following objectives are to be achieved:

- A literature review to help understand the concepts of artificial neural networks and genetic algorithms as well as how to implement them into a system and whether they show any ability to create agents that can handle problem domains within games. As well as a literature review of games as a test bed for AI research to determine if there are any benefits in using games as a test bed as well as if there is any benefits in using Spelunky specifically.
- A system which incorporates an artificial neural network as well as a genetic algorithm which has the ability to play Spelunky and assess agent performance so better agents can be created to play Spelunky. This system will be constructed using GameMaker, which is what Spelunky was created in, as well as a C++ DLL (Dynamic-Linked Library) which is where the artificial neural network and genetic algorithm will be contained.
- The execution of a series of tests that test agents in multiple testing levels which get gradually more complex one after the other. The created system at the end of each level will produce the best agent it has found for the level which will then be used for the next testing level. These best agents will also be tested through previous testing levels to see if they retain the ability to handle the challenges in each level. All this testing will be conducted to see if solutions for problem spaces in each level can be created and if those solutions can retain the knowledge of how to succeed in previous testing levels. If these agents can retain the ability to succeed in multiple levels with multiple problem areas then
they will be more likely to succeed at playing Spelunky as they have knowledge of how to handle its various problem domains.

- The production of data from the aforementioned testing to assess agent performance and whether our aim has been met and how valid the hypothesis made is. The data will also be assessed to see how the artificial neural network and the genetic algorithm affect the population and see what is typical behaviour for the system. An example would be if the artificial neural network and genetic algorithm make agents that very poor solutions or very good solutions or whether the agents all learn at the same rate. The point of this is, if the typical behaviour of the system is known then it is known what to expect from future tests and the capabilities of the system in general when it comes to agent intelligence.

- The gathering of data to determine if an agent or agents exist that can handle all previous testing levels they will be tested in a special level that contains a combination of the challenges faced in all the previous testing levels. If the agent or agents can pass this then we have a candidate or candidates who are potential solutions to Spelunky’s problem domains.
2.0 Literature Review

2.1 Introduction

In this chapter we will be looking at firstly the previous work that has been done in the fields of artificial neural networks and genetic algorithms as well as why these fields would be of good use for creating agents in Spelunky. Secondly we will then be looking at the artificial intelligence problem space within games and why games are a good problem area to study artificial intelligence. Thirdly we will look at why Spelunky provides an acceptable problem space in which to conduct testing and development.

This chapter’s aim is to try and impart an understanding of the work that has been done previously in the fields mentioned above as well add more credence to the aforementioned hypothesis and why it is worth studying as well as why it should be studied in the way this paper intends.

2.2 Artificial Neural Networks

2.2.1 Concept

The first work into artificial neural networks was done by Warren McCulloch and Walter Pitts (Russel and Norvig, 2010) in which “They proposed a model of artificial neurons in which each neuron is characterized as being as being “on” or “off”, with a switch to “on” occurring in response to stimulation by a sufficient number of neighbouring neurons (Russel and Norvig, 2010, p16). With this model McCulloch and Pits went on to show that “…any computable function could computed by some network of connected neurons…” (Russel and Norvig, 2010, p16) and even suggested that suitably defined networks could learn. This was shown by Donald Hebb through the use of an updating rule which changed the strengths between neurons, this rule is called Hebbian learning (Russel and Norvig, 2010).

2.2.2 Implementation

Using the explanation provided by Russel and Norvig, 2010, p728, the construction of an artificial neural network model as described above would be achieved by the use of nodes that receive inputs from adjacent upstream nodes. These inputs will consist of the value produced by each upstream node’s activation function as well as the weighting each node has associated with it. Since a node could have multiple upstream nodes attached to it a summation of all the inputs received from the connected upstream nodes is made and then that data is fed into an
activation function. This activation function value is then fed to any connected downstream nodes and this process begins again. Let’s use an example to better illustrate this implementation. Using a diagram from Thompson T[a]:

![Diagram of a simple multilayer feed-forward neural network](Figure 1.0 – A Simple Multilayer Feed-Forward Neural Network)

(Source: Thompson T[a])

We can see that (starting from the top) Hidden node one will receive inputs from Input node one, two and three which will consist of Input node one, two and three’s weights and activation function values. This data will be summarised and then passed through an activation function which will then produce an output that will get sent to Output node one. Output node one will do the same process as Hidden node one in that Output node one will gather all the inputs from Hidden nodes one, two and three, summarise them and put them through an activation function.

2.2.3 Flavours of Artificial Neural Networks

The diagram and implementation used is of a simple feed-forward neural network which is a network where the connections only flow in one direction and downstream nodes receive data from upstream nodes (Russel and Norvig, 2010. p729). A feedforward network also does not have any kind of state other than the weight of each node and “…represents a function of its current input…” (Russel and Norvig, 2010. p729).

Another kind of network is a recurrent network which feeds its outputs back into its own inputs which means the activation levels of the network form a dynamic system which can reach a stable state, produce oscillations or exhibit chaotic behaviour. The networks response to inputs is affected by its initial state and therefore can display short term memory which
makes it a more accurate model of a brain but also makes it harder to understand (Russel and Norvig, 2010. p729).

As well as being able to choose whether you wish to use a feed-forward or recurrent network you can also choose how many layers you wish to have in your network as when a network has its inputs connected directly to its outputs it is called a perceptron network (Russel and Norvig, 2010. p729). Whereas a network with multiple layers (like in figure 1.0) is called a multilayer network which consists of hidden layers between the input and output layers (Russel and Norvig, 2010. p729).

The activation function also has different implementations changing how the neural network will act (Russel and Norvig, 2010. p729). Activation functions come in either a hard threshold known as a perceptron or a logistic function known as a sigmoid perceptron the difference being that the sigmoid perceptron is differentiable (Russel and Norvig, 2010. p729).

### 2.3 Genetic Algorithms

#### 2.3.1 Concept

Originally known as machine evolution, genetic algorithms started as the belief that “…by making an appropriate series of small mutations to a machine code program, one can generate a program with good performance at any particular task” (Russel and Norvig, 2010. p21). This is done through the use of a selection process which preserves mutations that seem useful (Russel and Norvig, 2010. p21).

#### 2.3.2 Implementation

Using the example given by (Russel and Norvig, 2010. p127) they start by creating a randomly generated set of states known as a population and they use a string to represent each individual in the population. To produce the next generation they state that a fitness function is needed which should return a higher value for better states and in their example of the genetic algorithm an individual’s probability for being selected for reproduction is proportional to their fitness score.

To breed the individuals they randomly select two pairs in accordance with their probabilities based on their fitness scores. They then choose a crossover point randomly in the string representation of each individual in a pair. Looking at the diagram on page 127 we can see that the characters after the crossover point just get swapped from one member to the other creating two offspring (this happens for both pairs). Russel and Norvig state that this
crossover can produce offspring with a state that is very different from that of the parents however the population is supposed to be diverse in the early processes so crossover frequency takes larger steps in the state space early so then it can take smaller steps later when most individuals will be quite similar.

The offspring are then subject to a random mutation based on a small probability in what seems like an attempt to further mutate the offspring based on the quote “…genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads” (Russel and Norvig, 2010. p128).

Even though this is not stated we can then assume that the population would be tested again, new fitness values produced for the new population and then new members selected for the breeding process, this would happen again and again until some point of termination was reached.

### 2.4 Example of Use

#### 2.4.1 Example

A modern piece of work that incorporates both an artificial neural network (not using Hebbian learning) and also using a genetic algorithm is the EvoTanks project that was created by a student named Tommy Thompson during his time as an undergraduate at the University of Strathclyde and then improved upon from there (Thompson, T[b]). Upon inspection of the EvoTanks toolkit (Thompson, T[b]) it uses a feed-forward multilayer artificial neural network to process inputs and then produce values from those inputs which then drive agent behaviour within the EvoTanks Game.

#### 2.4.2 Artificial Neural Network Implementation

Inputs are sent to the network where they are normalised within input neurons, these neurons then send that input data to connected neurons which intern runs the TanH activation function (see Benjio, 2003) on this data. This process is continued through all the intermediate neurons, the number of which can be specified at the start of the program, until the output neurons are reached where this activation function is ran 1 more time in each output neuron. The resulting data is then taken and assessed to see what actions will be run e.g. if value > 0.04 then turn right.

The weighting of each neuron is actually represented by a set of values in each member in the population so if your neural network required 10 weights then each member of the population...
would have a set of 10 weights that would get plugged into the network essentially giving each population member a different “brain”.

2.4.3 Genetic Algorithm Implementation

The genetic algorithm is implemented slightly differently to the example given by Russel and Norvig in section 2.3.2 in that while there is a fitness function that rates the score of each population member, when breeding occurs a population member can only breed through either crossover or mutation, not both. So based on fitness scores a group of the population known as the parents are selected (which are the best performers) and then based on a random number generator they are selected for breeding by either crossover or mutation.

If mutation is selected they have a weighting modified based on a mutation probability and a value from another random number generator e.g. if the mutation threshold is 0.4 and the random number generator produces 0.3 the weight will not be modified however if it was 0.4 or higher than the weight will be mutated.

If crossover is selected on a parent then a population member will be randomly selected to breed with and their set of weights will be split at a randomly chosen point and the values crossed over.

Based on the toolkit as well as information on EvoTanks provided at Thompson T[a] we can also see that there seem to be two selection methods for the bots, Tournament Selection and co-evolution. Tournament selection seems to involve a population being put against a normal enemy opponent controlled by the game engine, the selection process then grades how well a population member did against the enemy e.g. how close they get, how much damage they did, did they kill the enemy. The best population members then breed and the selection process begins again.

Co-evolution is similar to tournament however population members are pitted against each other, the reason for this is because Thompson wanted to try and overcome an issue with tournament selection where the evolved agents were becoming trapped in local maxima. This is where the best possible tanks that could be evolved against a particular enemy were being reached but that tank would not perform well against enemy tanks it was not trained to fight against. Co-evolution causes the population members to adapt to each other creating an “Evolutionary Arms Race” (Thompson T[a]).
2.4.4 Evaluation

Now that it is clear how the EvoTanks system works we now need to see how well it did. Based on the co-evolution performance data (Thompson T[a]):

(Figure 2.1 – Co-Evolution Test Results, Population Fighting Each Other)

(Source: Thompson T[a])

This graph tells us that the tanks fighting against each other were able to adapt to each other fairly successfully and the average performance after a number of evaluations seems to level out between the 75% and 80% mark which is a positive performance.
This graph shows the performance of the co-evolved population against the standard enemy tanks and as can be seen the average performance quickly rises and levels out between the 65% and 70% mark which again is a positive performance.

From the data shown we can conclude that this implementation method of an artificial neural network as well as a genetic algorithm creates artificial intelligence agents that have the ability operate and learn creating better and better agents to tackle the problem domain.

Based on this it could be concluded that an artificial neural network and genetic algorithm have the potential to create agents for Spelunky that can handle the various problem domains within the game. Based on EvoTanks operation if inputs could be defined for the agent and then outputs that would drive agent behaviour be created we could have an agent that would be able to navigate a Spelunky level. Based on this agent’s performance we could then selectively breed agents that display behaviours suitable to the problem domain e.g. finding a door, avoiding an enemy, finding treasure, to create better agents that can complete the level or complete it faster than previous agents.

### 2.5 Using Games for AI Research

#### 2.5.1 Challenges Games Provide

When trying to ascertain why games would make a good problem domain for an artificial intelligence a good starting point is asking why do games create good problem domains for actual intelligences e.g. Humans (Thompson, 2014). Games create a challenging environment for our brains because they can provide a large number of challenges that need thought and deliberation (Thompson, 2014). For instance there are a number of different gaming genres such as Action, Adventure, Action-Adventure, Role-Playing, Simulation, Strategy, Casual and Massively Multiplayer Online (Fritts).

The challenges that can be found in these genres are varied such as in the action genre where the player would need quick reflexes as well as careful timing in a real time environment (Fritts). On top of these challenges the player would also need to specialise to a particular challenge such as the challenge of racing, fighting, shooting or platforming (Fritts). This contrasts to genre of adventure where the player has a need for exploration, puzzle solving and collecting items in a turn based environment (Fritts). Strategy games would require the
player to manage resources which could include constructing buildings and units as well as making resource decisions such as how now that I have a building/unit how am I going to use them (Fritts).

The genre of simulation can provide an even more challenging environment as the player will have to follow a set of real world rules and use those real world rules to play the game such in Flight Simulator X (Fritts).

Games can also require the challenges of lateral thinking, determinism and accessibility of Information (Thompson, 2014).

Now if all these problem domains are available and create a challenge for humans then surely by extension they would also provide good challenges for an AI to figure out how to overcome (Thompson, 2014) as well as create a safe and risk free environment to create AI in e.g. a flight simulator game can be made/used to test an auto pilot AI instead of having to use a real aeroplane that could have real world consequences (Thompson, 2014).

2.5.2 History of AI in Games Research

Originally games such as chess were being used to test AI but due to the lack of progress gained in this area using games to create AI fell out of favour and funding was lost for research (Thompson 2014). Then in the early 1990’s a game named Pac-Man was chosen as a potential useful research space for AI creation however nothing was ventured into this until 2001 where Aditya Kalyanpur and Mohan Simon used genetic algorithms and neural networks to create an AI that could play a fundamental copy of Pac-Man that they had created (Thompson, 2014). This then lead on to Marcus Gallagher and Amanda Ryan creating an agent that learnt to play Pac-Man using an evolutionary and rule-based approach (Thompson, 2014). This was done as a part of the proceedings of the 2003 Congress on Evolutionary Computation (CEC) and the Pac-Man game that Gallagher and Ryan used was a more faithful clone of Pac-Man but still not Pac-Man (Thompson, 2014), the reason for using clones of Pac-Man is because games creators are very reluctant to hand over their source code so people can run experiments on them as they are giving away their intellectual property (Thompson, 2014).

AI research then moved on to the successor of Pac-Man which was called Ms. Pac-Man which largely operated the same as the original Pac-Man but the enemies differed in that they were now non-deterministic as they had the ability to do something random which meant players couldn’t learn their behaviour patterns (Thompson, 2014).
A set of researchers decided that instead of working on a clone of Ms. Pac-Man they wanted to work on the real Ms. Pac-Man and so created a framework that could see and play the game by taking screenshots of the screen, analysing the contents and then sending inputs to the game (Thompson, 2014). They then formed a competition using this framework by opening up its use to researchers around the world and inviting them to create the best Ms. Pac-Man player they can.

A further area of interest that cropped up from this was trying to play as the ghosts instead of Ms. Pac-Man as the ghosts should be working as a team and it would be interesting to develop AI for the ghosts to be able to do this (Thompson, 2014). A clone of the game was made for this as nobody could figure out how the ghosts worked and also there was no ability to control the ghosts in the original Ms. Pac-Man game (Thompson, 2014).

After this in 2009 research moved onto Super Mario Bros which concentrated on trying to develop a player that could evolve to play a clone of Super Mario Bros using an evolutionary algorithm (Thompson, 2014). This was done by Julian Togelius, Sergey Karakovskiy, Jan Koutnik and Jürgen Schmidhuber for the Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG) (Thompson, 2014).

Creating AI for Mario then lead onto the Mario AI Competition held between 2009 and 2012 which had a number of different “tracks” one of which was gameplay which involved creating your own AI Mario (Thompson, 2014). Submissions were made and the winning submission was by a researcher called Robin Baumgarten which used a pathfinding algorithm called A* search to find the best route through the map at any given moment (Thompson, 2014). This submission caused the competition to segregate learning and gameplay into different research tracks because of the performance was so good and the learning methods could not keep up (Thompson, 2014).

A different track also tasked researchers with creating an AI that could procedurally create Mario levels that would be different every time a player played through them (Thompson, 2014).

Research for Mario eventually ended due to a cease and desist released by Nintendo (the owners of Mario) but lives on as the PlatformerAI competition (Thommy, 2014).

Another tournament then arose in the form of the 2K BotPrize competition in 2008 which focused on the genre of first person shooters and was designed for the game Unreal Tournament 2004 (Thompson, 2014). This competition involved researchers developing an
AI agent that was good enough to play against human opponents but that would be able to fool humans into thinking it was itself a human (Thompson, 2014). This was done via players being assigned as judges who would then have a judging gun which they could then use to shoot the player they believed to be an AI, these AI of course would be fighting against other players and the judges would have no idea who the AI are (Thompson, 2014). This provided an interesting problem as not only did you have to create an AI that could beat a human player but you also had to create an AI that could pass as a human which would mean it would have to fallible (Thompson, 2014). This was achieved in 2012 and the researchers received $7000 for their effort (Thompson, 2014).

Research in games still continues today with a game named StarCraft which is a real-time strategy game (Thompson, 2014). The research is the creation of an AI that can play the game and beat human opponents, which as it stands at the moment some AI can beat some human players but can’t beat professional players (Thompson, 2014).

2.6 Spelunky’s Problem Domain

Spelunky is a game where the player is expected to navigate a level where there are aggressive and passive enemies, hazardous terrain and traps as well as sections of the map that are walled off but can be passed via destroying the wall (Scales and Thompson, 2014).

(Figure 3.1 – Spelunker Using a Bomb to Destroy Terrain)

(Source: Scales and Thompson, 2014)
The player is also expected to obtain gold and other treasure, weapons, non-player characters and the player also has the choice of using a store to buy items (Scales and Thompson, 2014). The player can also traverse the terrain through the use of ropes to reach higher places and sprinting to reach further places when jumping (Scales and Thompson, 2014).

![Image of Spelunky gameplay](image)

(Figure 3.2 – Spelunker Using a Rope to Reach Higher Terrain)

(Source: Scales and Thompson, 2014)

The player also has a score the objective of which (like any score) is to maximise at the end of each level when the player reaches the exit door (Scales and Thompson, 2014). Furthermore the levels of Spelunky are procedurally generated which means no two levels will ever be the same providing more complexity for any AI agents as they simply won’t be able to just memorise the levels (Scales and Thompson, 2014).

While Spelunky is similar to say Ms. Pac-Man in that the player is expected to avoid enemies while navigating a maze and maximising their score, the player is also expected to strategies their navigation as well as the destruction they inflict on the environment allowing them to reach more desirable treasures and objectives (Scales and Thompson, 2014).

Spelunky is also similar to Super Mario Bros in that there is a platforming element where you jump to traverse the level some enemies can be killed by jumping on them (Scales and Thompson, 2014). Where Spelunky is different is that some enemies require weapons to be used to defeat them, the player will need to backtrack through the level as well as focus on
particular areas to obtain treasures. The player will also need to search and find the exit to the level whereas is Mario the player simply had to move to the right (Scales and Thompson, 2014).

So while Spelunky is similar to Mario and Ms. Pac-Man the above shows us that it differs from them enough to introduce new challenges for AI that should make for an interesting problem space. Spelunky also satisfies the final criteria of a benchmark domain, “A benchmark that challenges current AI capabilities and methodologies”, as it provides challenges that solutions have not been made for yet based on previous research that has been stated above e.g. working towards an objective that at play time is unknown, searching for and then helping non-player characters and unlocking areas of the map through destruction to gain rewards.

2.7 Why Spelunky

2.7.1 Criteria

Each of the competitions described above became a benchmark for AI researches as they produced problem areas that were interesting and challenged the state of the art (Scales and Thompson, 2014), looking at the competitions above the range of benchmarks that have created typically have the following characteristics. They provide a domain where uniform solutions are not evident and therefore produce interesting problems to solve (Scales and Thompson, 2014). They provide a domain that is extensible and customisable so problems can be scaled up or down in complexity (Scales and Thompson, 2014). A domain that is recognisable in gaming culture (Scales and Thompson, 2014). A benchmark that challenges current AI capabilities and methodologies (Scales and Thompson, 2014).

2.7.2 Domain

Based on the information given in section 2.6 it is evident that Spelunky provides a domain where uniform solutions are not evident and therefore produces interesting problems to solve.

2.7.3 Domain Extensibility and Customisability

Spelunky is extensible through the use of an API that has been developed for it which uses a combination of GameMaker 8.1 Pro and C++ DLLs (Dynamic Linked Libraries) which means that C++ and GameMaker’s GML (GameMaker Language) can be used to create an agent that can play through Spelunky as well as gather data about the world in Seplunky (Scales and Thompson, 2014).
Spelunky is also customisable through the use of a level editor that has been created to allow users the ability to create test levels that can be made to fit any level of complexity and difficulty the game can produce (Scales and Thompson, 2014).

### 2.7.4 Domain Recognisability

The final criteria is then the one that states that “A Domain is recognisable due to its place in gaming culture.

Spelunky was released as freeware for windows PC in 2009 and has been remade on the Xbox 360, again on the PC, PlayStation 3 and the PlayStation Vita (Scales and Thompson, 2014). The fact that is has been remade on a range of consoles would suggest a certain amount of popularity with the game as new works/recreations would not be possible if there was not a demand for them.

Spelunky as stated before has a domain that is similar to Mario and Ms. Pac-Man which would also suggest that the domain itself would be recognisable by researchers.

Both of these points would mean that Spelunky as well as its domain would be easily recognisable to people through its place in gaming culture as well as its roots in gaming culture e.g. platforming.

### 2.8 Conclusion

Based on the information previously given it would be reasonable to conclude that creating an artificial intelligence agent to play Spelunky using an artificial neural network and a genetic algorithm is indeed possible as it has been shown that neural networks can be used to control agent behaviour and actions within games. A different learning method such as Hebbian learning could be used however with the success of EvoTanks the combination of artificial neural network and genetic algorithm seems to be an effective method of controlling agent behaviour and suitability.

Genetic algorithms should be able to analyse agent suitability to the problem domain and should be able to select appropriate agents for breeding. This means an agent that can successfully complete Spelunky levels/test levels should be possible to obtain.

The API in Spelunky should give the resources necessary for the creation of the agents, neural network, genetic algorithm as well as any data and functionality needed to drive these.
Spelunky itself due to the domains it is similar to as well as the differences it has should provide an interesting enough domain that creating agents for it will be a challenge and won’t simply be recreating work that has otherwise already been done through Mario and Ms.Pac-Man.

With regards to the previously stated aim and objectives of this paper it is feasible that a system can be created that incorporates an artificial neural network and a genetic algorithm which can be used to create agents that can play Spelunky. Those agents will be able to be tested as well as tested in custom levels, performance data on agent performance should also be able to be collected and then more tests done based on that data.

2.8.1 Key Issues

Due the size of Spelunky’s problem domain as described earlier there are a lot of different scenarios that can be tested for. It seems rather unfeasible to create a system that will be able to tackle every single aspect of Spelunky in the time this research has to be created.

What could be seen as an issue is A* search. The reason for this is because of how well it did during the Mario AI competition where it basically solved the problem domain which would mean research into using artificial neural networks and genetic algorithms for Seplunky is not necessary.

This however this does not seem to be of much concern for Spelunky as A* search is just a path finding algorithm meaning that given location A and Location B travel to location B from A the best way possible (t2thompson.com).

For Mario this works well as the player is only expected to travel to the right which luckily is where the exit is. Spelunky provides a problem with A* search as you are expected to traverse the level discovering new objects e.g. gold, the exit. This means that at play time A* search has no B to go to and therefore a planning/decision system would need to be implemented to make the agent search the terrain looking for objects to give A* objectives to traverse toward. Therefore A* on its own is not good enough and this research can continue.

2.8.2 Refined Research Questions

Due to the vastness of Spelunky’s problem domain I believe a better research question would be can an agent be created that can handle a specific aspect of Spelunky’s problem domain as opposed to all of it. A good starting point for this is can an agent handle the terrain in Spelunky as well as given an objective can the agent traverse to it no matter which direction it
is in. Granted A* search would be able to achieve this but as stated when the testing complexity moves to unknown objectives causing the agent to search for objectives A* search would not work. The idea is that training the neural network to travel to an objective is the starting point to then get the neural network to search for objectives and then traverse to that objective once one is found.
3.0 Methodology

3.1 Introduction

This section will detail the steps to be taken to develop a system that incorporates an artificial neural network as well as a genetic algorithm. It will also cover how the agents created from this system will then be tested to identify their suitability towards a specific aspect of Spelunky’s problem set as well as how we will get metrics from this testing to better understand their suitability. Details of how these metrics are to be measured will also be given as well as any ethics that are involved and any general Limitations that may occur.

3.2 Research and Development methods

3.2.1 Research Methods

The research methods that will be used are experimental and statistical this is because experiments will need to be created to test agents and assess their suitability to the problem domain and whether artificial neural networks and genetic algorithms are a good choice for Spelunky.

Statistical research will need to be employed because there will need to be objective data gathered from the experiments so the agents’ suitability can be accurately assessed and compared if we want to see how well agents are handling experiments of increasing complexity.

3.2.2 Development Method

For the creation of the system that will be required for the subject matter at hand an agile style development methodology is the most appropriate as there will need to be a number of different functions built into one complete system. It would be wise to develop each function in turn and test its functionality and then once it is acceptable develop the next function to work alongside the previous function.

To clarify this point this system is going to need an artificial neural network, a genetic algorithm, a fitness function, a data generation function, a method to save agent data so they can be loaded into new experiments or run agents through older experiments to see if they perform as well or have potentially been bred out in favour of agents that handle the new experiments.
Based on these elements as stated it would be wise to develop them one by one and assuring their functionality before building more features.

3.3 System Design

3.3.1 Artificial Neural Network Design

As discussed in section 2.2.3 there are a couple of different flavours of artificial neural network and based on this information and work done previously by Tommy Thompson (Thompson, T[b]) the best method of creating an artificial neural network would be similar to the one that Thompson used.

This is because the neural network used a configuration that worked well for driving the behaviour of the tanks. The other alternatives such as a single layer perceptron network might not be as effective due to its smaller size which would have less weightings to affect calculations that might produce more varied results in agent behaviour.

The feed-forward aspect of Thompson’s neural network also seems appropriate as the learning is supposed to be being done by the genetic algorithm instead of data being fed back into the system to promote some form of learning that way.

Finally the activation function used by Thompson which is differentiable would be more useful as it can produce more varied results from the network which can drive more agent behaviour when certain constraints are put around what actions are to be taken based on the output data of the network.

3.3.2 Genetic Algorithm Design

The genetic algorithm that will be employed will be the same as the genetic algorithms that Thompson used (Section 2.4.3) as well as Russel and Norvig (Section 2.3.2), this is because both the crossover and mutation breeding methods seem appropriate methods of modifying data between parent and offspring enough to produce different behaviours. These methods however are rather destructive due to their either complete rearrangement of genetic data or their mutation of values that may in fact be crucial to beneficial behaviour. Respective of this Thompson’s method will be used over Russel and Norvig as the mutation of data will be restricted to a small percentage based change and the cross over breeding method will also only happen via a percentage change and therefore has less chance to comply rearrange data and potentially destroy positive behaviour of an agent.
The fitness function to be used will be the tournament style selection that Thompson used in EvoTanks, this is because this style of function assessed how capable agents were against standard game enemies and then through elitism breed the ones that were the best suitability. This would be the best option for Spelunky as we will need to assess which agents have the best suitability for a given problem domain and then breed them accordingly so the best traits are preserved.

The other Genetic fitness function that could be used was Thompson’s co-evolution method which pitted members of the population against each other but has no relevance to Spelunky as the agent’s will not be tested against enemies in this research due to the vastness of the problem domain that was discussed earlier and that it would be best to start with terrain navigation as that is a big part of Spelunky being a platforming game.

3.3.3 Data Generation Design

There will need to be a way to gather data from these experiments to identify how well the agents are doing against particular experiments.

A method could be employed that gathers the data on each population member and states how well they did on average so each population member could be compared alongside another. This method however would not be feasible as population sizes and the number of generation sizes you would like to run could make the number of results to display and analyse very large. For example in the case with EvoTanks the standard test set was 200 agents over 20 generations which creates 4000 test results. This would make getting an overview of how well each agent did compared with the other agents in other generations as well as how the populations did in general very difficult.

Another method for gathering data that could be applied is one where the average for the generations are gathered as well as the standard deviation shown so an idea of how well the population in general are doing as well as the standard deviation between population members showing if the population as a whole is learning or whether very suitable agents and very unsuitable agents are being created. This would also allow us a good overview of how each generation compares to another generation and whether the generations are getting smarter over time or are again just creating suitable and unsuitable agents.

If there are a high number of unsuitable agents this will bring the average and standard deviation down so it would also be useful to display the highest score achieved in that generation so we know if solutions to the problem domain are being found.
This second method where data is measured per each generation as opposed to each agent is a better solution as the data is a lot more condensed and we get a better overview of how well the generation are performing however the hypothesis of this research states that artificial neural networks and genetic algorithms can be used to create suitable agents for Spelunkys problem domain. The generation data tells us how well the generations did in each specific experiment level however it does not tell us if the best agents created will perform just as well in the previous experiments or have just adapted to suit this problem domain. Therefore the best agents at the end of an experiment will need to be ran through all previous experiments to see if they are able to handle multiple different environments or just the one they were trained in.

To show if the best agents can handle previous levels, data will need to be gathered on each individual agent’s average performance in each level to show whether there is one or a set of agents that perform consistently well across each experiment level. If such an agent or set be found then we have potential candidates that can be the start of a solution to solving the current problem area within Spelunky.

A function will need to be put in place so only parents are tested and are not bread from level to level as we only want to test the parents.

3.4 Experimentation Methodology

Since terrain navigation has been selected as the problem domain for experimentation an outline of what experiments will be run is necessary to better understand the testing and data that will be produced.

Since in Spelunky one of the main aspects is collecting treasure, items or getting to an objective such as the exit (section 2.6.2) it would be a good start to test an agent’s navigation ability to an objective. For this several experiments will be conducted where an exit is placed somewhere in a testing level and it is up to the agent to traverse the terrain and reach the exit door. The testing levels will ideally increase in complexity each time to make the problem domain a little harder and hopefully train agents that are able to handle a number of different terrains.

Each experiment will come in sets of 2, this means that if experiment 1 has the agent running to the right experiment 2 will be the same level of complexity but will have the agent running to the left. This is so we are more likely to obtain agents that are running towards an objective
and not just happen to be running towards where the objective is like in Super Mario Bros where the agent only has to run to the right.

As with EvoTanks there will be a population of a certain size and a number of generations to be tested. Each population member will be tested a certain number of times.

When creating a new generation in an experiment we select a set of best agents from the previous generation, this set becomes known as the parents which then breed and a new generation is created. For each experiment the best agents from the previous experiment will be selected as parents and will be used to seed a new starting population so positive behaviour gathered from previous experiments will be used to help the new generation tackle the new problem area. These parents will also be used to seed a starting population in previous experiments to gauge current agent suitability through previous levels.

### 3.5 System Construction

#### 3.5.1 Artificial Neural Network

The implementation of the artificial neural network is going to be similar to that of Tommy Thompson’s implementation in EvoTanks (Thompson, T[b]) however Thompson’s was customisable with how many inputs and outputs you have.

The artificial neural network will be of a fixed size that is not customisable by the user at runtime. It will consist of 4 inputs which are the distance to the objective, the height of the objective from the agent, the direction of the objective from the agent and if any obstacles are in the way.

It will contain 2 hidden layers as this should create some variability in the output values of the network. These layer numbers can be changed in the program code to test if the number of hidden layers changes agent characteristics significant or marginally. Each hidden layer will have the same number of nodes as the input layer which again can be modified in the program code to see if it affects agent behaviour, this number of nodes is simply a starting point.

The outputs will have as many nodes as the input nodes the idea being that each output will produce value based on the inputs that are available e.g. input 1 is distance to an objective so output 1 will produce a value for distance to an objective, input 2 if for height to the objective so output 2 will produce an output for height to the objective. In reality this isn’t so as you could pick any of the outputs for values as all the outputs are connected to all the nodes to the
left of them and they are connected to all the nodes to the left of them and so on right down to the inputs the same as how the neural network in EvoTanks was.

The values of outputs are assessed and then actions are driven from them, the threshold to activate the run right command for the agent will be >0 the value for run left will be <0, the value for jumping will be >0 and the value for jumping over an obstacle will be >0.8. the value for judging whether to press up which is what allows an agent to go through a door is if the received value is smaller than 0.1 but bigger than -0.1. These are all just starting values and can be changed to see if they affect agent behaviour.

The idea between making some of the activation thresholds a value bigger or smaller than 0 was to make those actions more common whereas constraining some of the other activation values to specific ranges was so they would reservedly used e.g. the agent is jumping when in front of an obstacle because an obstacle is there and an agent is going into a door because the door is there not because it just happened to be pressing up at the time.

The weightings that will be placed into each one of these nodes to affect calculated values and drive agent behaviour will be obtained from each population member. Each population member will have 48 different weight values between -1 and 1. The reason for 48 different values is that there are 12 nodes in the neural network that need values. These nodes will each have weights for the 4 nodes that are to the left of them so the output nodes will contain the weights for each of the 4 hidden layer nodes next to them. Those hidden layer nodes will have the weightings for each of the input nodes. So if 12 nodes need the weightings for the 4 nodes next to them that is 12 x 4 = 48. The reason for the value between 1 and -1 is because a number size is needed that constrains the possible range of numbers that can be used to represent input values but are also differentiable. This range is also used for the activation function and was also used in in EvoTanks.

The activation function to be used in this network in the TanH transfer function which will constrain the values being passed to nodes in the neural network to between the -1 and 1 range we are using (Bengio, 2003). This transfer function is expressed as:

\[ T = \frac{e^x - e^{-x}}{2} \frac{2}{e^x + e^{-x}} \]

(Figure 4.0 – TanH activation function equation using the mathematical constant e)
e being a mathematical constant equalling
2.718281828459045235360287471352662497757… (MathWorld[b])

and x being a value, in our case the summarisation of all the connected nodes weights and values. This is the same activation function that EvoTanks used.

The method being used to actually get data to the neural network is the Spelunkbots API which can be used to send data from GameMaker 8.1 to the DLL being used to run the neural network. This way we can pass information about the agent to the neural network and then normalise that data to fit between our -1 and 1 range to then be passed on up the network. GameMaker also allows the DLL to pass data back to it via a function call so values can be sent back to GameMaker which can then be interpreted and subsequent actions performed on the agent e.g. run right.

The Spelunkbots API will need some editing so it can perform things such as:

- Updating the artificial neural network with agent data such as their location and the location of the object they are going towards.
- Updating of the artificial neural network to use different population members.
- The initiation of breeding,
- Calculating agent fitness.
- Saving parent data.
- Printing parent and generation performance data.

These functions will be implemented by creating functions within the DLL that the Spelunkbots API can call so data can be passed between the Spelunkbots API and the DLL to determine when the functions described above need to happen.

### 3.5.2 Genetic Algorithm

The breeding part of the genetic algorithm will operate by taking the just tested population and then ordering them from best fitness score to worst fitness score. A selection of the population will be taken for breeding and will be known as the parents. As stated in the design a threshold will be set to decide whether a particular parent will breed via crossover or mutation. In mutation another threshold will be set to decide whether a particular weight held by a parent will be modified when placed into an offspring, in crossover the parent will breed with the parent under it, both parents will have a randomly determined chunk of their
weighting data swapped with each other unless we get to the last member in the population which will automatically be mutated as there are no more parents under it to breed with.

The reason for this choice of forced mutation is the parent above the last parent could have been selected for crossover breeding and would have been bred with the last parent, so there is no point in allowing the last parent to crossover breed as it would just produce a child that would be similar to a child that would have already been created as the last parent and the parent above it would have already been bred before. In addition there is no point in allowing the last parent to crossover breed with any of the higher scoring parents as effectively, if the last parent has a low score, you will just be watering down all the good qualities of the higher scoring parent with the lower scoring parent making poor offspring.

The idea of crossover breeding with the parent below it is that both parents should be of a similar score which means they will contain similar weighting data. The combining of this data will make an offspring that will be similar to each other’s traits and potentially combine good traits of both parents making a better offspring instead of potentially watering down weightings that make the agent perform well with weightings that will make the agent perform unsuitably.

The breeding threshold value for crossover will be 30% which gives mutation a probability of 60%. The reason for this is an individual weightings mutation probability is only 30% meaning that mutation has a smaller chance of destroying valuable weights that produce good behaviour. Since crossover breeding has the potential to destroy a combination of weights that make the agent perform well be replacing a section with an entirely different set of weights, it makes sense that mutation be given a bigger selection chance as it is much less destructive. These threshold values however can be modified to see how differing the breeding method can affect the next population’s suitability.

Since navigating the terrain was going to be the area of focus in this research the suitability to be measured will be given an exit to the experiment level can an agent navigate its way across the terrain successfully.

The fitness function will operate by measuring how close an agent was to the level exit when the level ended as well as how long the agent took to reach the exit. The closer the agent is to the exit when the level ends the higher the agents score will be, if an agent is in front of the door when the experiment ends they will be awarded 100 points however a time penalty of a maximum of 1 point will be penalised from every player depending on how long they took in
the level. This means that no player can ever reach 100 points and it gives a clear separation between which solutions to the problem area are faster than others. An example of how many points to penalise would be if a level has a time limit of 10 seconds then if one agent completed the level in 5 seconds it would only get 0.5 points deducted whereas if another completed the level in 7 seconds it would get 0.7 points deducted.

Time limits have been imposed because as stated before they allow us another level of measuring so we can measure the best agent suitability but also some agents might not ever reach the door and so will need a limit of how long it is reasonable to take.

As agents will be tested multiple times the recorded suitability score that will be assigned to an agent and then passed onto the genetic algorithm will be an average of the scores an agent received over a number of runs.

3.5.3 Data Generation

To gather the data on how well the population has done on average the record of all the averages that is generated for the breeding part of the genetic algorithm will be averaged, the standard deviation calculated and the highest score taken from the averages record. This data will be written to one text file after every generation, when running a different experiment level the data will be written to a separate text file so the two are easily distinguishable. When an experiment level is finished the parents chosen to breed the next generation that will be used for the start of the next level will have their weighting data saved to a text file, this can be used for parent tests.

Similarly, when creating the data for assessing the suitability of the parents when tested through previous levels, their averages data will be taken from the averages record sent to the breeding function of the genetic algorithm and written to a text file. This will happen for each level and the data will be written to the same text file, when a new set of test levels are selected this averages data will be written to a different text file to make both sets of tests distinguishable.

The function of allowing parent testing to happen will be achieved via the creation of a folder where the user can put the seed file that was generated containing the parent weighting data that was discussed above. If the population size to test is then constrained to that of the parent number size then no breeding will happen as children would exceed the population size. The number of generations to be tested would also need to be set to 1. This would mean that only the parents should be tested from experiment level to experiment level, a flag will then be set...
to indicate that the parents weighting data should be used inside the neural network and not the weighting data from a new set of randomly generated population members like at the very start of a normal run of experiments. This flag will also not allow the parents’ weighting data to be saved from completed experiment levels and that the parents’ average data should be printed rather than the other printed data which contains the population’s average as well as the standard deviation and top fitness score.

3.6 Data Analytics Method

The way of analysing all the data that will be produced from the testing will take the form of graphs from Excel, the files that are generated will be formatted as CSV (Comma Separated Values) file for easy insertion into excel where a line graph with error bars will be used to show population averages and the standard deviation. This form of graph should give a good overview of how well each generation did when compared with another as well is the generations are getting smarter over time.

The parents’ data will also be displayed using graphs from Excel and the data will be inserted into Excel the same however the graph will take the form of a bar chart. The experiment level numbers will be at the bottom of the graph and then the parents’ bars will be displayed above the level number. This method should give us a good overview of how well the agents are doing per level and also if there are any of the agents who consistently reach 99% or above.

3.7 Limitations

3.7.1 Testing Size

This system will be constrained by the time it takes to run experiments based on population size, the number of tests of each population member, the number of generations and the time limit for each test. Based on an increase of room speed that is possible in Spelunky a level (see Figure 4.1 in section 4.3.1) that was created took approximately 2 seconds to complete when using a “bot” created in Spelunky’s source code (Appendix A). This bot was simply programmed to run right and hold up so it would walk in front of the door and go through.

Using this 2 seconds to complete the level let’s give the bots 4 seconds to complete it so bots that are a little slower can still make it. Using the standard generation, testing and population sizes of the experiments in EvoTanks we have 200 population members, 30 generations and 50 tests per population member. This gives us a total potential completion time calculation of
4x50x200x30 = 1200000 seconds which equals approximately 333 hours or nearly 2 weeks to run one experiment level.

Therefore the experiments conducted will have to be drastically smaller to allow a reasonable experimentation time.

3.7.2 Learning Capability

As discussed above in section 2.4.3 Thompson created another experimentation method called co-evolution which pitted agents against each other which was able to stop them from getting stuck in a local maxima due to each agent having to develop better and better skills to cope with other agents developing better and better skills. Experimentation for Spelunky however will not have this testing method or anything like it and runs the risk of agents being created and bred that only actually handle the current problem domain they are in. This means that when testing parents through previous experimentation levels we may see them failing a number of levels because they are not a collection of all the good traits learned from all the levels but instead a collection of traits that are good at handling the current level.

3.8 Conclusion

The proposed methodology appears to meet all the requirements necessary to create a system that provides an artificial neural network and a genetic algorithm for the creation of agents that can play Spelunky. It also shows a decent ability to assess these agents and then produce data that can be used to gauge agent performance as well as then display that data for analysis and discussion.

The limitations expressed do seem to have a potential impact on the performance of the agents and will need to be assessed as data is gathered from experimentation to see if they are a legitimate concern.
4.0 Findings and Analysis

4.1 Introduction

In this section we will be looking at the data gathered from testing the artificial intelligence agents through 10 different test levels. As described before we will be looking for whether the artificial neural network and the genetic algorithm can find solutions to problem domains and whether those solutions have the ability to succeed in previous problem domains that led up to the domain those solutions came from. The population as a whole will also be looked at to gauge whether the population is learning as a whole.

This section will contain a caveat on the data being used as well as a description of the experimentation levels, analysis of the data gathered to assess the points made in the previous paragraph and finally a conclusion of the information gathered from the experiments.

4.2 Data Caveat

With the data being used to show how well each generation did on average there was a bug caused by GameMaker that caused certain scripts not to run at the start of a new level. These scripts would involve the gathering of data which would drive inputs within the artificial neural network. Without this data the artificial neural network would not work correctly and the first agent would score lower due to being temporarily handicapped. This issue is normally resolved after the level reloads and GameMaker will run the scripts, so if the number of tests per population member was 10 then 1 out of 9 tests for the first agent would be inaccurate.

This issue was thought to be resolved before testing as a fix was implemented that would simply reload the level twice so the scripts would run properly. In the circumstance of loading new levels when running a continuous experiment e.g. test agents in level 1 and then level 2 and so on without the need to close the application to specify new levels. This issue was observed to not happen and was believed to only affect the very first level of the set of experiments. When testing the parents to see how well they handled previous levels however it was clear that this issue still persisted on the loading of every new level due to the data that was gathered. Typical results for testing a parent in the domain they came from would be that the parents would all consistently score between 99 and 100 however there were instances of the parents scoring below 99 or just failing the experiment all together with a score of 0. We would expect every parent to reach between 99 and 100 when being tested in the domain they
came from as those parents were specifically selected in that domain meaning they are solutions for that domain.

Odd instances that were recorded were instances of multiple parents failing the domain they came from, the maximum observed was 3. This is strange because as noted the issue above only affected the first test of the first population member. Possible explanations for this phenomenon are it could be caused by an error in which the wrong experiment levels were selected making the results look like the parent was tested in their domain when they weren’t. It could also possibly be the result of instances where on the testing machine if a high amount of CPU and hard disk space were in use by a process that wasn’t linked to Spelunky then the game would render slowly slowing down agents and making them perform poorly.

The fix that was implemented to reload the levels upon their first start so the appropriate scripts would run was modified so it would work on each level. The parents were then retested in the same way they were tested when the abnormal data was gathered and they displayed data one would typically expect to see when parents were tested in the domain they came from in that each parent scored between 99 and 100. This shows that the issue is now fixed however the anomalous results gathered that showed parents failing their domains is still unexplained as this did not happen when the parents were retested.

The data that will be used for analysis to gauge the performance of each generation will be the data gathered when the issue from GameMaker was still present. There are two reasons for this, one is simply because due to the amount of time experimentation takes there was not an opportunity available for retesting. The other reason is that with the number of generations, tests and population members that were being used in the experiment any affect this issue would have would be minimal as it would only affect the first generation of an experiment, it would also only potentially affect between 1 and 3 members of that generation. This means the data in all other generations would be accurate and only the average and standard deviation of the first generation would be skewed slightly. This issue would also not affect non continuous level testing as the level would be restarted twice from the beginning anyway and no next level would be loaded. Solutions were also gathered at the end of each of the experiment levels so any affect the bug would have had are minimal as it did not affect the system’s ability to breed out bad behaviour and find solutions.

The data gathered when testing the parents however did have the fix described earlier that reloads every level twice because accuracy of data in this experiment needed to be exact so we could gauge if any of the parents were successfully able to handle previous domains.
Incorrect data could mean a solution was there that could handle previous domains but because of the bug from GameMaker it was misrepresented.

### 4.3 Experiment Levels

Altogether there are 10 test levels that deal with the problem domain of terrain navigation. The levels start simple i.e. walking to a door and progressively get more complicated by adding terrain to climb and obstacles to avoid.

To begin a description of the objects you will see in each level will be given.

<table>
<thead>
<tr>
<th>Level Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Level Entrance](Figure 5.1)</td>
<td>Level entrance door – a grey square that looks similar to a stone slab with marks on it. This is where the agent will be placed upon level start.</td>
</tr>
<tr>
<td>![Level Exit](Figure 5.2)</td>
<td>Level exit door – a square with a black inside and a grey outline. This is where the agent must go to exit a level.</td>
</tr>
<tr>
<td>![Terrain](Figure 5.3)</td>
<td>Floor block – a brown square with what look like clumps of dirt squashed together. Is a standard terrain block that an agent can walk on.</td>
</tr>
<tr>
<td>![Background](Figure 5.4)</td>
<td>Background Block – A dark grey/black square that looks like rocks compacted together. Is a standard background block and is not intractable.</td>
</tr>
<tr>
<td>![Spikes](Figure 5.5)</td>
<td>Spikes – A white set of pointy triangles. Is a standard obstacle in the game and causes player damage when fallen on.</td>
</tr>
</tbody>
</table>

(Table 1.0 – Level Objects)
4.3.1 Experiment Level 1

(Figure 6.1 – Experiment Level 1)

The Problem domain of this level is simply given the exit door as an objective can the agent make its way to it.

When seeing these levels the reason for entrance doors being randomly placed in the environment (see top left of figure) is because when entering the level editor where the pictures were taken from, the user is immediately given a block to add to the environment. In this case it was an entrance door and the actual levels only have one entrance door which will be located somewhere on the standard terrain blocks.
4.3.2 Experiment Level 2

(Figure 6.2 – Experiment Level 2)

This level covers the same domain as level 1 however the exit is in the reverse direction as we are attempting to train agents that are trying to find the door not simply run in a direction that the door happens to be in.

4.3.3 Experiment Level 3

(Figure 6.3 – Experiment Level 3)
This level moves on to a new problem domain which tackles, given an obstacle that the agent will need to jump up to navigate can it reach the exit.

4.3.4 Experiment Level 4

(Figure 6.4 – Experiment Level 4)

The problem domain for experiment 4 is the same as experiment 3 however the agent need to run to the left.

4.3.5 Experiment Level 5
This level changes the domain to can the agent avoid a deadly obstacle to reach the door. By this point we hope to have an agent that can reach the door and jump over obstacles to reach the door. We now want to see if the agent can actively avoid a threat to reach the door.

4.3.6 Experiment Level 6

(Figure 6.5 – Experiment Level 5)

(Figure 6.6 – Experiment Level 6)

This level covers the same domain as level 5 but the agent must now run to the left.
4.3.7 Experiment Level 7

(Figure 6.7 – Experiment Level 7)

The domain being tested here is now that we know the agent can respond to a threat such as a spike pit can the agent navigate a threat such as a potential deadly drop. The objective here if for the agent to jump up to the door rather than just not perceiving a threat and falling off the edge where the gap is.

4.3.8 Experiment Level 8
This domain is the same as experiment 7’s however the agent must run to the left.

4.3.9 Experiment Level 9

(Figure 6.9 – Experiment Level 9)

The domain here is the same as level 7’s however there is an extra jump that needs to be made to reach the door. This is so a solution can be found that is more likely to be responding to the threat of a drop rather than just randomly jumping at the right time to miss the first gap.
4.3.10 Experiment Level 10

(Figure 6.10 – Experiment Level 10)

The domain being tested here is the same as Level 9’s however the bot now needs to run to the left.

4.4 Generation Analysis

For the following experiments the following values were used:

- Number of tests per population member – 10
- Number of population members per generation – 50
- Number of generations per experiment level – 20
- Level 1 time limit – 4 seconds
- Level 1 time limit – 4 seconds
- Level 3 time limit – 8 seconds
- Level 4 time limit – 8 seconds
- Level 5 time limit – 4 seconds
- Level 6 time limit – 4 seconds
- Level 7 time limit – 4 seconds
- Level 8 time limit – 4 seconds
- Level 9 time limit – 8 seconds
- Level 10 time limit – 8 seconds
The first set of testing involved creating a random population and then testing them through the levels that were shown above, the average performance of the population, the standard deviation and the best result were recorded. Experiments 5 and 6 were run continuously as well as 7 and 8 so experiments 6 and 8 may have suffer from the GameMaker bug described earlier and their data may not be truly representative of the full capabilities of the population.

4.4.1 Level 1

![Population Suitability & Deviation Level 1](image)

(Figure 7.1 – Average Population Performance with Standard Deviation and Best Score for level 1)

Based on the results from experiment level 1 we can see that the population over time does not seem to consistently get smarter each generation. Looking at the generations the generation fitness trend line seems to move up and down and does not show a clear trend of each generation doing a little better than the previous generation or the trend line typically moving upward.

Looking at the first generation we can see that the generation average is just over 20%. Since there are 50 population members and 10 parents that means that the parents comprise 20% of the total population. Looking at the data we can estimate that around 10 solutions were found in generation 1 which would then explain why there is a sudden jump in average population fitness in generation 2. This is because agents would have been bred from the 10 best that
were selected from generation 1 and those offspring would have had traits that were more suitable to the problem area. This also explains why the fitness seems to level out but still with slight variation as you consistently have 20% each generation because of the parents and so then it is luck after that whether the offspring is a good solution or a bad solution which will raise and lower the average population fitness based on the ratio of good solutions created to bad solutions.

Looking at the standard deviation of each generation (represented by the blue error bars) it also appears that the generations over time do not seem to be getting more consistent in their intelligence meaning that the agents being created are not of a similar intelligence. Instead the large standard deviations would suggest that the agents being created are either capable of solving the problem domain or not cable.

Looking at the highest score trend line we can see it stays consistently just under the 100% mark in each generation meaning that at least one member of that generation was a solution to the problem domain in that experiment level.

4.4.2 Level 2

(Figure 7.2 – Average Population Performance with Standard Deviation and Best Score for level 2)
Looking at the results for level 2 the results are similar to level one in that the population did not grow more intelligent or consistent as a whole over time. The standard deviation is large meaning that either capable or not capable solutions were being created but not solutions that performed similarly to each to each other. Solutions were indeed found in each generation though.

4.4.3 Level 3

(Figure 7.3 – Average Population Performance with Standard Deviation and Best Score for level 3)

Experiment level 3 displays the same trends as experiment levels 1 and two however the first generation of the experiment scored an average under 20 which the first two experiments didn’t meaning that the parents from level two that were passed into this level struggled to solve this problem domain. Solutions were however found within the first generation.

The standard deviations of the first 3 generations were smaller than the standard deviations we saw in the first two tests which means that the generations as a whole were quite consistently poor at solving the problem domain.
These results are similar to level 3’s results. The first generation struggled but the population gradually got smarter as new offspring were being made that were more capable of solving the problem domain thanks to parents being selected that were more capable of solving the problem domain.

Experiments 1, 2, 3 and 4 are starting to show an interesting trend where the generations will not score higher than a certain amount. For levels 1 and 2 it was around 60% and for 3 and 4 it was around 40%.

(Figure 7.4 – Average Population Performance with Standard Deviation and Best Score for level 4)
4.4.5 Level 5

(Figure 7.5 – Average Population Performance with Standard Deviation and Best Score for level 5)

The data in this level shows a generation average trend line that varies less than the other experiments which had a tendency to start off poor and then get better eventually levelling off. Since the fitness score for each generation is low, only reaching between 20% – 30%, the data would suggest that the generations found this problem domain very challenging as the generation as a whole didn’t score very high and ever generation after them also did not score very high.

This suggests that this test is good at eliminating agents that would have scored higher through luck even though they may not have been actual good solutions for the problem domain. An agent that knows to run right but not that there is an obstacle in front of it would have scored higher just by luck in tests 1 and 2 as they would have made it nearer or two the door. Agents that can’t detect a threat aren’t given a chance to do better by luck as they are eliminated quickly.

Otherwise the data is still similar to that of the tests run previously as the population as a whole doesn’t get smarter, the standard deviation suggests that many capable and non-capable
agents are being created and the population is not reaching a consistent intelligence and also that solutions are being found.

4.4.6 Level 6

(Figure 7.6 – Average Population Performance with Standard Deviation and Best Score for level 6)

Generation 1 in this experiment did very poorly and not only that but the best agent in this generation was only capable of reaching halfway to the door, which looking at figure 4.6 is where the spikes are meaning that for the first 4 generations there were no capable solutions found. This is however one of the experiments that would have been susceptible to the GameMaker bug, it is however unlikely that the bug would have persisted for 3 further generations meaning that the population did not have artificial neural network configurations that could cope with this problem domain.

By the fifth generation however a solution is found and the general fitness of the generations does start to climb after each generation. This could be due to more numbers of solutions being found and then being selected as parents which then get passed into the next generation’s experiments. Since these parents will be able to solve the problem domain as the number of parents increases the average fitness of the population would increase.
The standard deviation also gets larger and larger indicating that the starting generations where all consistently bad solutions however as the number of good solutions increases we can see that the standard deviation for each generation becomes large like in other experiments meaning that the generations create good solutions and bad solutions but does not seem to create a lot of agents with similar finesses.

### 4.4.7 Level 7

![Population Suitability & Deviation Level 7](image)

(Figure 7.7 – Average Population Performance with Standard Deviation and Best Score for level 7)

Level 7 produced results that are at first similar to level 6’s in that at the start no solutions are found for the problem domain however once solutions are found the average fitness of the population increases. Once the 4\(^{th}\) generation is reached the results are similar to previous experiments in that the fitness of each generation seems to level off and does not go above a certain fitness, in this case it is between 30% – 35%.
4.4.8 Level 8

(Figure 7.8 – Average Population Performance with Standard Deviation and Best Score for level 8)

This is another level that would have been susceptible to the GameMaker bug and as we can see a solution is found after the first generation. This could either be due to the bug affecting a number of parents so the population as a whole did very poorly or it could just be that the parents largely weren’t affected and they were just not adequate solutions to this problem domain like in the results for level 7 which were similar.

The results however are similar to that of level 6’s and 7’s in that they start off poor but steadily climb as more solutions are found.
4.4.9 Level 9

(Figure 7.9 – Average Population Performance with Standard Deviation and Best Score for level 9)

The results here emulate the results we have seen before, at first the parents from the previous level are poor at the problem domain. A few solutions are found and population fitness steadily gets larger until you reach the 20% mark which means you have all 10 parents and after that the trend line levels off and there is slight variations due to the level of suitable agents and non-suitable agents being created on top of the 10 solutions you already have.

The standard deviation also does the same as has been seen before where it starts of relatively small due to the population being consistently unsuitable but gradually gets larger as the ratio of good solution to bad solutions evens out.
4.4.10 Level 10

(Figure 7.10 – Average Population Performance with Standard Deviation and Best Score for level 10)

Looking at the results for level 10 not much needs to be said as the results closely emulate the results of level 9.

4.5 Parent Analysis

For the following experiments the following values were used:

- Number of tests per population member – 10
- Number of population members per generation – 10
- Number of generations per experiment level – 1
- Level 1 time limit – 4 seconds
- Level 1 time limit – 4 seconds
- Level 3 time limit – 8 seconds
- Level 4 time limit – 8 seconds
- Level 5 time limit – 4 seconds
- Level 6 time limit – 4 seconds
- Level 7 time limit – 4 seconds
- Level 8 time limit – 4 seconds
- Level 9 time limit – 8 seconds
- Level 10 time limit – 8 seconds

The parents will be tested in sets which consist of the parents that were selected at the end of an experiment level being run through all previous experiment levels. So set 1 will be the parents from level 1 ran through level 1. Set 2 will be the parents from the end of level 2 ran through levels 1 and 2. Set 3 will be the parents from the end of level 3 ran through levels 1 – 3. Sets 4 – 10 follow the same pattern.

A caveat to the parent performance data that will be shown is that there is a slight variation in the performance of the parents. Looking at figure 6.1 below it can be seen that the same parent on the same level performs slightly differently each time. The cause for this is unknown however all the parents score 99% or above when tested in the domain they were selected in so the difference is only how long it took them to complete a level and not whether they completed it or not.

(Figure 8.1 – Parent Score Variability for Level 1)

This caveat has been expressed to inform the reader that the data to be shown is representative of a parents capabilities and does not show the exact capability of each parent. If a person wishes to recreate these parent experiments this caveat has been expressed to also inform the reader that their results will not necessarily match 100% the results shown here but will be similar.
4.5.1 Parent Set 1

(Figure 8.2 – Parent Set 1 Performance Scores, Level 1)

When tested through level 1 we would expect to see all the parents score 99% or over which is indeed what we see here meaning that the parents selected were indeed good solutions for their problem domain.

4.5.2 Parent Set 2

(Figure 8.3 – Parent Set 2 Performance Scores, Levels 1 and 2)

As the data shows us only parent 3 was capable of running levels 1 and 2 successfully and the other parents failed. This would suggest that the other parents may have just been running to the right in level 1 and not actively searching for the door whereas in level 2 only parent 3 was actively searching for the door. The other parents just ran off the edge of the terrain or if some were searching for the door the parent did not go through the door, ran past it and then
ran off the edge of the terrain. All parents other than 3 must have been killed by running off the terrain as they all have scores of 0%. If they stopped near the door or had at least moved towards the door they would have scored higher than 0%.

4.5.3 Parent Set 3

(Figure 8.4 – Parent Set 3 Performance Scores, Levels 1 - 3)

The results are similar to that of parent set 2 in that when ran through previous levels the majority of parents do not cope with the previous levels and one parent managed to succeed in all previous levels.
4.5.4 Parent Set 4

(Figure 8.5 – Parent Set 4 Performance Scores, Levels 1 - 4)

In this set of results the majority of parents succeeded in the tests levels except level 3 which is different than before because they mostly failed in each previous levels. In this experiment unfortunately none of the parents were able to successfully make it through every previous level however there are 4 parents in level 3 that did at least express traits that moved them closer to the door.

4.5.5 Parent Set 5

(Figure 8.6 – Parent Set 5 Performance Scores, Levels 1 - 5)

In these results the parents seemed to do a lot better of the first level than in previous sets, even in the sets where the parents came from a level where they were running to the right.
Level 3 seems to still be a problem for the agents to navigate. There are still no agents that can handle all previous levels successfully.

**4.5.6 Parent Set 6**

(Figure 8.7.1 – Parent Set 6 Performance Scores, levels 1 - 5)

(Figure 8.7.2 – Parent Set 6 Performance Scores, level 6)

There are still no parents that can handle all the previous levels successfully but a trend does seem to be appearing where, with the exception of levels 3 and 4, if the level has the parent running in the same direction as the level they were selected in the parents are able to handle it successfully.
4.5.7 Parent Set 7

(Figure 8.8.1 – Parent Set 7 Performance Scores, levels 1 - 5)

(Figure 8.8.2 – Parent Set 7 Performance Scores, levels 6 and 7)

The trend mentioned in the previous section 4.5.6 has changed from if the level has the parent running in the same direction as the level they were selected in the parents are able to handle it successfully, to, parents are able to handle it more successfully. This is because if you look at level 5 not all the parents are able to complete it so all the parents are not able to handle it successfully however there are still a larger number of parents completing it than the levels where the parents need to run to the left e.g. 2, 4 and 6. This means that the parents are able to handle levels that have them run in the same direction as the level they were selected in more successfully as more parents are completing them.
We still do not have any parents that are able to successfully complete each level although parent 2 at least scores in each and every level.

4.5.8 Parent Set 8

(Figure 8.9.1 – Parent Set 8 Performance Scores, levels 1 - 5)

(Figure 8.9.2 – Parent Set 8 Performance Scores, levels 6 - 8)

Like as seen before levels that have the agents running in the same direction as the level they were selected in seem to have more parents scoring than on levels where they are running in the opposite direction to the level they were selected in.

Parents 1, 2 and 10 achieved scores in the highest number of levels when compared with the other parents however we still don’t have any parents that can handle all levels so far.
The same direction trend as stated before still seems to hold and we can see parent 2 is at least scoring in each level.
With this final piece of data we can that a definite trend has emerged. The parents definitely seem to perform better when running in the same direction as the level they are selected in however the levels where the agents run to the right seem to be producing at least one parent that can handle most if not all of the previous levels, see section 4.5.9’s results in comparison to this sections results.

None of the parents however have been able to complete each previous level successfully and have at best at least scored in each level.
4.6 Conclusions

The generation testing confirmed the system’s ability to create and then test agents, selecting the best ones to be bred as a way of breeding good solutions to problem domains. The generations themselves did not reach a standard intelligence level and the data suggested that there were just as many suitable agents being created as unsuitable agents although this ratio is skewed in favour of the unsuitable agents as the later tests did not manage to get above 30% fitness rating. Due to this low fitness rating is suggests that the majority of children created were unsuitable for the problem domain.

The experiments do seem to follow a trend that once the 20% mark is reached how well each generation does is fairly random as each generation varies from the next also the more difficult an experiment level gets the less well each generation does on average which again suggests that the majority of offspring are poor at solving the problem domain.

Solutions were however found for each level and a full 10 parents were selected at the end of each experiment level demonstrating the system’s ability to find solutions to problem domains. The majority of the solutions found, based on the parent testing data, however do not seem to be able to handle previous levels well and at best at least score in each level and score best on levels where they are running in the same direction as the level they were selected in. The parents that are most adaptive also seem to be coming from experiment levels where they run to the right.
5.0 Discussion

5.1 Introduction

In this discussion the objectives that have been laid out previously will now be assessed to see whether they have been met, the study as a whole will also be assessed to gauge its adequacy at tackling the subject matter of artificial neural networks and genetic algorithms as well as their use in Spelunky.

5.2 Use of an Artificial Neural Network and Genetic Algorithm in Spelunky

The artificial neural network operates in accordance with the artificial neural network that was described in section 2.2 as well as works similarly to that of Tommy Thompson’s in EvoTanks described in section 2.4.

The Genetic algorithm also works as described in section 2.3 in that it has a fitness function that assessed agent performance and then selectively breeds agents that display positive behaviour just like Tommy Thompson’s in section 2.4.

The results displayed above show that the artificial neural network does work in its ability to control agents and that the genetic algorithm is able to select and breed good agents for the purpose of solving the current problem domain. The system can also produce data that can be used to analyse the performance of agents and generations as well as allow populations to be seeded prior to new experiments.

The artificial neural network does display odd behaviour though because as stated in the section 4.5 each parent’s performance was different each test even though they were on the same level. It also seems experimentation with the output thresholds of the neural network could be needed as agents had a tendency to travel by jumping rather than walking which in Spelunky would make them likely to jump into an enemy or try and jump through an area that is too small to travel through by jumping. The current configuration of the artificial neural network however served its purpose of controlling agents so they could tackle Spelunky levels.

The genetic algorithm seems to be very limited in its ability to select parents that have traits that are suitable for previous domains as well as breeding a new population that are all of a consistent intelligence. It is not currently clear whether the latter is a bad thing as there is a trade off with modifying agents as do it too much and you will destroy good traits but don’t
modify them enough and you get agents that are similar copies of each other and there is no
genetic diversity allowing some to complete a level faster or allowing offspring to develop
and learn to traits e.g. searching for a door rather than running right or left.

5.3 Use of Spelunky as a Research Domain

Spelunky does offer a wide range of problem domains that makes creating agents using
artificial neural networks and genetic algorithms challenging to do as is shown by the fact that
a parent could not be found that would handle all previous experimentation levels. This is
further confirmed by the fact that the experiments conducted here were very controlled
scenarios with little complexity and danger when compared with the range of traps, enemies
and level sizes that a normal Spelunky game has.

5.4 Agent Performance

The agents when the generations were being tested performed adequately and to the end that
was needed of them. All that was needed was for agents to be created that could complete a
level by reaching the exit door and those agents then be bred so better agents could be
selected.

While the generations didn’t really learn as a whole and get progressively more suitable each
generation that wasn’t the aim of this study which was just to see if agents could be created to
tackle Spelunky’s problem domain, specifically terrain navigation.

The parents however did not perform as well as had been hoped and were quite limited in
their ability to handle previous levels, the majority of which couldn’t. The parents did all do
well in the levels they were selected in or levels that were similar so it is clear that the parents
that are being selected are simply the best solution for the given problem domain and not
necessarily a combination of all useful traits that make them adaptable to different domains.

5.5 Conclusion

The study has met the aim that it was created for in that an artificial neural network and a
genetic algorithm has been used to create agents that can be used to play Spelunky. Agents
have been found which can complete Spelunky levels and parents have been found that can
complete a number of similar levels as well as one parent that was capable of scoring in every
level.
The limitations of this study are that an agent could not be found that could handle all Spelunky experiment levels meaning that the objective of creating an agent that can play Spelunky is not met as the area that the agents have been introduced to has not had a solution produced for it.

The artificial neural network also displays some odd behaviour and the agent control thresholds that send data to GameMaker about what moves should be made will need adjusting to see how it affects agent behaviour e.g. constant jumping to travel.

The genetic algorithm has no ability to seek out agent traits that allow them to be adaptable to each level and also the mutation and crossover thresholds will also need adjusting to see how they affect offspring quality.
6.0 Conclusions and Recommendations

6.1 Key Issues

There are 4 key issues that have been experienced during this research, these issues will be summarised for the purposes of clarity as well as ascertaining any impact they have had on any research conducted to give the conclusions in the next section more explanation as to why they have been reached.

The first issue that will be stated is the GameMaker bug that was described in section 4.2, this bug caused scripts at the start of a level not to be run so the artificial neural network didn’t receive data it needed to control the agent. This meant that an unknown number (potentially as many as 3 based on the number of parents that were tested in their own domain but failed the domain, however there could more) of agents at the start of a level could perform badly due to this bug and skew the population results.

This bug however was only present during the generation testing and was not present during the parent testing as a fix was implemented when it was clear something was wrong. It would only occur when multiple levels were ran one after the other, rather than running all test levels separately by having to select them from the level selection screen. The affect this bug had on the results of this research is minor as only a couple of levels tested would have been capable of experiencing this bug and only the first generation of the level would have had this bug affect it, the other 19 generations as well as most of the population members in the first generation would not have been affected by this bug. Therefore in most cases the system was able to operate properly and produce agents capable of solving their problem domains.

Another issue was the variability of the performance of parents being tested on the same level which was described in section 4.5. As shown the parents’ performances were different for each of the three same tests they were used in. There are currently no explanations for why this is happening but the differences in the performances of the parents is very small and is not enough to bring them below their 99% score.

A possible explanation for this is that maybe due to the distribution of resources on the testing machine Spelunky may be given different amounts of CPU time when each parent was running, so a fraction of a second less or more to render a frame or run a calculation could cause the agent to be a fraction of a second quicker or slower causing their score to be different each time as time is a factor in their score. Since each agent was able to stay within
their 99% score however it seems this issue has little impact on agent performance and the results gained are still accurate.

This issue like the last seems to be negligible towards the results gained and has little impact on the performance of the agents or the conclusions that can be gained from their results.

The next issue was the one mentioned in section 5.2 where the output thresholds for the neural network could do with experimentation. This is because observed current agents have a tendency to jump as they travel which in a normal game of Spelunky makes them likely to jump into traps or off of the edge of terrain as well as make it difficult for agents to navigate tight spaces. The experimentation of the threshold values could have the potential to produce agents that are more likely to run rather than jump and only jump as necessary. This is possible as currently an agent only needs to receive a value that is more than zero in a certain output to jump, if this threshold was made higher this would make a combination of weights that would produce an output high enough to reach the threshold less common.

This issue however does not make the results gathered from the testing done invalid though nor the conclusions that can be gathered and just simply shows that experimentation into the best configuration of output thresholds for the agents is needed to produce agents that will perform the most effectively. The same experimentation is also necessary for the mutation and crossover percentage values used in breeding to see which percentages create the smartest offspring, like the threshold values these breeding percentage values do not make the results gained invalid nor the conclusions that can be gained from them.

The final issue was the one mentioned in section 5.2 in that the genetic algorithm seems to have a poor ability to select parents that have traits that allow them to handle previous testing levels. The genetic algorithm selects parents that are good solutions for the current domain they are in which is why parents seem to perform well in levels that have them running in the same direction as the level they were selected in.

This issue does not make the test results invalid and instead leads us to a conclusion about one of the limitations of the current system which is it is poor at creating agents that are capable of handling previous testing level and previous problem areas.

6.2 Conclusion

In this conclusion it is time to see how well the aim and objectives of this research have been met and what the information gathered means for the hypothesis that was given.
With regards to the first objective given it has been met successfully as a literature review was conducted that explained the concepts of artificial neural networks, genetic algorithms and using games a test bed for artificial intelligence research. It was shown in Tommy Thompson’s EvoTanks that artificial neural networks have the ability to control agents and produce agents that can act as solutions to specific problem domains, as well as show promise that neural networks and genetic algorithms can be used to produce solutions to Spelunky’s problem domains. The literature review also showed why Spelunky itself was a good choice as a game to use as a test bed for artificial intelligence research.

The second objective was also met successfully as a system was indeed created that incorporated an artificial neural network as well as a genetic algorithm which had the ability to play Splunky and assess agent performance so better agents could be created to play Spelunky.

The third objective was also met as the system was able to test agents in multiple levels to gain the best agents for those levels and those best agents were tested through all previous testing levels before them.

The fourth objective was met as results were gathered that allowed the assessment of agent performance and whether they were able to maintain knowledge that allowed them to succeed in previous testing levels. The results also showed what the typical behaviour of the artificial neural network and genetic algorithm was e.g. creating agents that had a tendency to jump or create generations that did not all learn at the same rate.

The fifth objective however was not met as a candidate that could succeed in all previous test levels was not found and so there was no point in testing any of the parents gained in a special level, containing a combination of the challenges from other levels, as none of the parents would be able to complete it.

From the inability to achieve the fifth objective the aim of this research, which was to develop a system that incorporated an artificial neural network and a genetic algorithm with the purpose of creating agents that can play Spelunky and find a solution to its problem domains, was never found. Therefore the aim of this paper was never met however candidates were found that were solutions to each of the test levels that were created, these test levels contained challenges that were found in Spelunky and therefore individual solutions were found to specific problem areas within the problem domain of terrain traversal within Spelunky.
The hypothesis of this research which is that artificial intelligence agents can be created that can successfully play Spelunky and act as solutions to its problem domains through the use of an artificial neural network and a genetic algorithm has therefore not been disproved. This is because agents were still found that can handle specific problem areas within Spelunky and can handle multiple similar areas just not all of them, therefore since agents can be created to handle specific problem areas of Spelunky more research needs to be done to see if one solution can be found to all the problem areas instead of solutions to each individual problem area. The hypothesis is not disproved but instead requires more research.

6.3 Recommendations

There are a few recommendations that can be given to anyone interested in continuing this research which are:

- **Artificial neural network and genetic algorithm configuration experiments**: This recommendation involves experimentation with the artificial neural network output threshold values as well as the genetic algorithm crossover and mutation breeding percentages. It would be useful to experiment with these to see if there is a configuration of values that work best in creating agents that perform effectively as well as breed without losing traits that could allow them to handle previous testing levels.

- **Agent performance variation explanation**: Finding out why when the parents were tested they produced different results each time would be useful to look into. This is because it would identify if there is a problem with the artificial neural network which was not found after construction and the problem is not potentially caused by resource usage on the host machine.

- **Knowledge Loss research**: This research would involve looking into ways of keeping the knowledge of how to complete the previous testing levels. The main problem with the current system is that the knowledge of how to complete previous levels gets lost with breeding and a solution is created which knows how to handle a specific problem domain and similar domains but not all. The creation of a genetic algorithm that can identify which traits are useful and which are detrimental would be useful. A look into artificial neural network learning methods such as Hebbain learning may also be useful as it would allow the artificial neural network to adjust itself to it can create the optimal solution for controlling agents within Spelunky.

- **GameMaker Bug Fix**: Finding a fix to the GameMaker script running bug would be useful as it would stop the need to create work arounds so testing can be done properly.
• **GameMaker Interface Removal**: The removal of the GameMaker interface may allow for tests to be done quicker and population sizes to be increased. When using Thompson’s EvoTanks you have the option to turn the interface to the game on or off, when turning this interface off the speed of testing rounds increases dramatically. If the same could be done for Spelunky the need to not render the game to the screen could speed up the rate at which the agents can move and may speed up testing.
References

AIIDE. *Artificial Intelligence and Interactive Digital Entertainment*. [Online]. Available from:
http://www.aiide.org/
[Accessed: 17\textsuperscript{th} May 2015]

http://bengio.abracadoudou.com/lectures/old/tex_ann.pdf
[Accessed: 27\textsuperscript{th} April 2015]

CIG. *Computational Intelligence in Games*. [Online]. Available from:
http://www.cig2014.de/doku.php/start
[Accessed: 17\textsuperscript{th} May 2015]

Fritts, J. *Computer & Video Game Genres*. [Online]. Available from:
http://cs.sl.edu/~fritts/csci142/schedule/csci142_game_genres.pdf
[Accessed: 25\textsuperscript{th} April 2015]

GAR. *Galactic Arms Race*. [Online]. Available from:
http://galacticarmsrace.blogspot.co.uk/
[Accessed: 17\textsuperscript{th} May 2015]

MathWorld[a]. *Hyperbolic Tangent*. [Online]. Available from:
http://mathworld.wolfram.com/e.html
[Accessed: 21\textsuperscript{st} May 2015]

MathWorld[b]. *e*. [Online]. Available from:
http://mathworld.wolfram.com/e.html
[Accessed: 27\textsuperscript{th} April 2015]


[Accessed: 25th April 2014]

[Accessed: 26th April 2014]

Thompson, T. 2014. 'AI in Games' [Season 1 Episode 1] - Artificial Intelligence Research in Games. [Online]. Available From: https://www.youtube.com/watch?v=hkradQTE7kQ
[Accessed: 23rd April 2015]

[Accessed: 23rd April 2015]

[Accessed: 25th March 2015]
Bibliography

[Accessed: 18th May 2015]
Appendix A: Time Limit ‘bot’ Source Code

```python
}  
global.playerJump = true;
global.lookUp = true;
} else if (!isRealLevel()) & & isRoom("zHighscores")
{
    // HANDLE HIGH SCORE ROOM
    global.lookUp = true;
} else
{
    global.goRight = true;
    global.lookUp = true;
}
```

(Figure 9.0 – ‘bot’ Source Code)

The above figure displays the source code for the ‘bot’ that was created to run through the door in the level shown in Figure 4.1 in section 4.3.1 to determine a time limit for the level. The code above is taken from the PlayerChoice script within SpelunkBots and the area of the above code that controls the bot is the very last else statement as the code above that else statement handles navigating the title screen or high score room. From looking at the code that controls the bot we can see it causes the bot to travel to the right and also look up which is needed so the bot can enter the door when it reaches it.
Appendix B: Software User Guide

This software user guide covers how to successfully run experiments within the system that has been built as a part of this research. To learn how to do things such as create bots within spelunkbots, create your own functions to interact with the DLL or other things specific with the function of the spelunkbots API then it would be advisable to look at the resources provided on the webpage provided by the reference page entry t2thompson.com.

To begin with, the SpelunkBots folder accompanying this paper should contain 7 folders:

**DLL Solution** – This contains the C++ DLL source code that when built gets placed directly into the “spelunky1_1” folder in the “Source” folder. The DLL’s source code can be accessed and edited via visual studio.

**Source** – This folder contains Spelunky, the spelunkbots source code and other files related to spelunkbots and the built DLL file.

**SeededStart** – This folder acts as a container for the seeding file that the user wishes to use for seeding the starting population of an experiment level. The seeding file within this folder should be called Saved-Parent.txt and contain the weighing values/genes of each parent agent.

**GenerationStats** – This folder houses all the data produced from experiments whether they be generation testing or parent testing.

**GameMaker** - This folder contains the GameMaker application needed to run spelunkbots.

**Spreadsheets** – This folder contains the spreadsheets you can use to create graphs out of the generation testing data and parent testing data.

**Experiments Data** - This folder contains all the data that was used in section 4.0 Findings and Analysis and gives an example of the file structure and file naming that you would see in the GenerationStats folder should you run multiple experiments and not delete the previous experiment folders.

**B.1 Controls**

The controls available in the C++ DLL to control testing are available in the form of variables. These variables are located towards the top of the Spelunkbots.cpp file which can be accessed by opening up the DLL source code in Microsoft’s Visual Studio.
These are the variables in the C++ DLL source code that control the configuration of the artificial neural network. As the variable names suggest inputs controls the number of input nodes, hiddenLayers controls the number of hidden layers, hidden controls the number of hidden nodes in each hidden layer and outputs controls the number of output nodes.

```cpp
    //ANN
    int inputs = 4;
    int hiddenLayers = 2;
    int hidden = 4;
    int outputs = 4;
```

(Figure 10.1 – ANN Controls)

maxTests controls the maximum number of tests you wish to run on a population member before moving on to the next population member.

```cpp
    //number of tests per population member
    int maxTests = 10;
```

(Figure 10.2 – Maximum Tests Control)

MaxPopulation states the maximum number of members within a generation, the current setting will make it so each generation will contain 10 population members.

```cpp
    //population
    int maxPopulation = 10;
    int mutationProb = 30; //30% chance of mutation
    int crossoverProb = 30; //30% of crossover breeding
    bool seededStart = true; //check for whether you want a seeded start
```

(Figure 10.3 – Population Related Controls)

mutationProb controls the probability of a gene being mutated within an offspring.

crossoverProb controls the probability that crossover will be picked as the breeding method, the current setting makes it so crossover breeding has a 30% chance of being picked and mutation breeding has a 70% chance of being picked. If mutation is picked then each gene in an offspring has a 30% chance of being mutated based on mutationProb’s value.

seededStart controls whether you want the starting population to be created via breeding from a set of already obtained parents or whether you want the starting population created randomly. Setting the variable to true specifies a seeded starting population and false specifies a randomly created starting population. When choosing a seeded start make sure to set the parentNum variable to the number of parents contained within the seeding file or lower. If the value is set to a higher number than is available in the seeding file the system will fail.
generations controls the number of generations you want to test.

parentNum controls the number of population members you wish to select as parents for breeding. Make sure parentNum can divide into maxPopulation without a remainder, this is so children can successfully be created for each parent so that the maxPopulation number can be reached e.g. if maxPopulation is 50 and parentNum is 10 then each parent can have 4 children if maxPopulation was 45 and parentNum was 10 then each parent can have 3.5 children and obviously it is impossible to have half a child. If parentNum cannot divide into maxPopulation the system will not fail but no children will be produced at the end of a generation.

levelTimes is a vector that contains a list of all the time limits you wish to impose for each level, each level you want to run has to have a time limit present in the levelTimes vector and vice versa. In the current configuration the level “FIND DOOR” will have a time limit of 4 seconds. If another level was added to levels then another time limit will need to be entered into levelTimes. Not doing so will cause the system to fail.

levels is a vector that contains the level names as they appear in spelunkbots and allows you to pick the levels you wish to run as part of an experiment. Each name entered into levels will run the level with the same name in spelunkbots and each level will be run sequentially. In levels there always needs to be “” entry at the end of the vector to indicate to the system that testing has ended and for the score room to appear in spelunkbots. Not adding the “” entry will cause the system to fail.

exitLevels is a vector that contains all the level names as they appear in spelunkbots that have the objective of exiting the level through the exit door. This vector lets the fitness function know which fitness test to run. In the system’s current implementation the only fitness test
available is exit levels but the fitness function can be modified to test for other fitness criteria, if this was done then a new vector would need to be created for all the levels that need to be assessed by the new fitness test and the CheckFitnessTest function in the spelunkbots.cpp would need updating so it would check to see if levels needed this new fitness test.

(Figure 10.7 – Data Production Control)

parentTest controls the type of data that is reported at the end of each generation. If parentTest is true then data for parents will be produced which represents each parent’s average score for the level they are in also the parent data file that gets produced at the end of all the experiments does not get produced. If parentTest is set to false then the data for each generation’s average score is produced as well as the standard deviation and the best achieved score for that generation. The parent data file containing the best found agents is also produced at the end of the experiments.

B.2 Testing a Generation

To test a generation you must first open up the DLL source code in visual studio and enter in the settings you wish to have in the control variables described above. For example let’s say we want to test 5 generations of 10 population members with 10 tests each and have 2 parents selected. We want to test them on the level find door and give them 4 seconds to complete the level. To create these conditions we would set the following control variables to equal the following values:

```
int generations = 5
int maxPopulation = 10
int maxTests = 10
int parentNum = 2
vector<char*> levels = { "FIND DOOR", "" }
vector<int> levelTimes = {4}
bool parentTest = false
```

If we want a seeded start then we would set seededStart = true and put a saved parents file like the one below in Figure 8.8 into the seededStart folder described above at the start of this appendix. The parents file needs to be renamed to Saved_Parents so the system can recognise it and use it and the file needs to be a TXT file. Remember the number of parents in the saved parents file needs to be the same or less than parentNum.
If you do not want a seeded start however and wish for your starting population to be created with random genes then set seededStart = false.

Once all this has been done then you want to go to the build tab in visual explorer and click on build solution. This should cause the DLL to build in the spelunky1_1 folder in the source folder which is where spelunkbots is kept. Once this is done you want to go into the GameMaker folder and start GameMaker. You will then need to open up spelunkbots into GameMaker and then press the green arrow towards the top of the application, hovering over the arrow should produce the txt “Run the game”. Two windows will open, one will be a console window that will display output to the user in the form of messages such as the ANN being configured properly and level testing completion information. The other window will be the actual game and you will want to click into this window to allow the game to run as clicking outside spelunkbots causes the game to pause.

Once you are on the title screen which is the screen with the three doors and the word Spelunky on it you will want to press F3 to bring up the level select menu. If you do not press F3 then the already created spelunkbots AI will take over and run the character to the level entry door which will put you in an actual Spelunky level which the current system built in the DLL is not designed to work with. Select the level with the same name as the very first entry in the levels vector in the Spelunkbots.cpp page in the DLL and then click “Load”, in our case we would select “FIND DOOR” and then click on Load. This will start the experiment and the experiment will not finish until all test levels have been ran, when they have all be ran then the score screen shown below in figure 8.9 will appear and you can then press escape which will take you eventually back to the high score room.

The spelunkbots AI that was stated earlier will again take over and start running the character through the exit to the high score room and then back through the entrance of the high score room. From here you can either press F3 again and select the start level like you did before and run the same experiments if you wish. If you want to run a different set of experiments then you will need to close spelunkbots using the X on the current window like...
you would if you wanted to close other applications. You will then need to go back into the DLL within visual studio to change the control variables to fit the new experiment you wish to run and then do the same as you did to run the first experiment i.e. build the DLL, open spelunkbots and run the first test level. Upon closing spelunkbots you may receive an error stating that spelunkbots is not responding, this is nothing to worry about and is a common issue that has not been fixed. If this does happen just tell windows to close the application and way you like.

![Image](you-made-it-score-room.png)

(Figure 11.2 – Score Room)

**B.3 Testing a Parent Set**

Testing a parent set is largely the same at testing a generation so to start off you will need to open up the DLL source code in Microsoft’s visual studio and enter into the Spelunkbots.cpp file. In this instance let’s say you wanted to test 10 parents through two levels at 5 tests each.

The settings you would need for this are:

```c++
int generations = 1
int maxPopulation = 10
int maxTests = 5
int parentNum = 10
vector<char*> levels = { "FIND DOOR", "FIND DOOR 2", "" }
vector<int> levelTimes = {4, 4}
bool parentTest = true
bool seededStart true
```
By stating that you only want one generation tested you will only test the parents once as that is all that is needed to gauge their stability in a level. You will set the maxPopulation to 10 and the parentNum to 10 so only the parents will get tested and no children will be able to be created when testing multiple levels. You would then set parentTest to equal true to you can get parent data printed out to output files and seededStart to true as you will need to input the parents you want to test into the system. You would enter these parents the same as you would in generation testing by which you would grab a saved parents file and put it in the seededStart folder making sure to name the file Saved_Parents and ensure it is a TXT file.

After this the process is the same as in generation testing in that you want to build the DLL, start spelunkbots up in GameMaker, press F3 on the menu screen and select the first level in the levels vector which in our case would be “FIND DOOR”. To change the experiment you will have to close the application and change the DLL like in generation testing and the crash issue may still appear.

**B.4 Creating Data Graphs**

Once testing is done you will notice that the GenerationStats folder will have a folder or folders inside it that say “Experiment” and have a number next to them. Inside the Experiment folder there will either be generation testing data or parent testing data. Generation data will have the name “GenerationData-Level” and then a number indicating the level that the data in the file is from. The numbers start at 0 so level 0 is the first level, level 1 is the second and so on so if you tested 3 levels you would expect to see 3 files saying GenerationData-Level-0, GenerationData-Level-1, and GenerationData-Level-2. There would also be 3 files containing saved parents data and they follow the same numbering structure except the file name says Saved-Parents instead of GenerationData.

This data can be imported into the spreadsheets in the Spreadsheets folder, the generation testing data would be imported into the spreadsheet called Standard Deviation and the parent testing data would be imported into the spreadsheet called ParentPerformance.

To import this data in Excel 2013 you select the cell that you would like to start the import and then go to the Data tab and then click the From Text button in the Get External Data section of the ribbon. The generation data is comma delimited and the parent data is simply
new line delimited. As long as the data has been imported correctly and is in the correct sections then the graph in the spreadsheet should update immediately.

**B.5 Function and Script Call Order**

When looking in the Spelunkbots.cpp it can be seen that there are many functions that have been created. These functions will be called at certain points within spelunkbots so an idea of when and why these functions are being called will aid the user to understand the operation of the system.

Firstly there is a function that is run inside the scrInit script within spelunkbots located inside the Scripts folder in the directory list on the left hand side. This script is original Spelunky source code written by Derek Yu and Mossmouth studio however it has been added to by Daniel Scales and myself. The function we are interested in is the Initialise function (not the initialiseDLL function). This function calls the Initialise function within the DLL which is responsible for configuring the ANN, setting up which fitness test we are testing for and also setting up a file structure for data to be output to.

We now move on to the oPLayer1 object within spelunkbots which is the object for the Spelunky character that the player plays as. This object can be accessed by expanding the Objects folder in the directory list on the left hand side. Inside the objects folder should be oPLayer1. In oPlayer1 you can designate scripts that you want to run at certain points within game execution. The starting point we are interested in is the Room Start event which is called at the start of a level. We are interested in this because this is where values to be used in the artificial neural network and genetic algorithm start to be assigned.
(Figure 12.1 – oPlayer1 Room Start Scripts)

Firstly when the room starts the PathFinder script runs which passes a range of map values to the DLL such as where spikes are, the terrain, where the entrance and exit to the level is and others. This code was written by Daniel Scales and also the functions used in this script were also written by Scales apart from the CheckTerrain function, the functions in this script can be found under the artificial neural network and genetic algorithm code apart from the CheckTerrain function, the start and end of the ANN and genetic algorithm code is marked by a succession of forward slashes in the DLL.

The SaveMap script is then run which saves the gathered map data into the PlayerInputs object in the DLL. The PlayerInputs object will search through this map data for any items of interest such as the exit location.

A piece of code is then ran which uses the StartingPosition function in the DLL which saves the Spelunky character’s starting position for use in the fitness function within the genetic algorithm.

The next event we are interested in is the step event where the PlayerChoice script will be called. The PlayerChoice script contains the code for the movements of our AI. This code is not all mine and I have added to it.

```cpp
else
{
    if(global.firstRun)
    {
        global.runCount = global.runCount + 1
        dead = true;
    }
    else
    {
        if(TimeUp(global.customLevelName))
        {
            dead = true;
        }
        else
        {
            CalculateMoves(global.playerPositionX, global.playerPositionY);
            global.goRight = GetMoves(0);
            global.goLeft = GetMoves(1);
            global.playerJump = GetMoves(2);
            global.lookUp = GetMoves(3);
        }
    }
}
In Player choice this is the code that will control the agent within test levels and the way it operates is it first keeps a count of how many times the current level has been run, this is so it can kill the agent twice so the levels can restart and fix the script bug that stops the scripts in the room start event from running.

After this is done a function in the DLL called TimeUp is called to see if the time allotted for the completion of the level has been reached, in the DLL the function is called TimePassed. If not then a DLL function called CalculateMoves is called which uses the artificial neural network to determine what moves should be made next. After that a function called GetMoves in the DLL is called for each character control variable to see if that control should be used or not.

The agent will continue to run until the agent either goes through the exit door or the agent dies, the agent can die either by running out of time or performing an action that kills it e.g. falling from a great height, landing on a spike etc. If the agent dies then this will be detected by the first code execution script after the “Action” comment which is denoted by an exclamation point in a yellow triangle in the Step event. The code in this section is not all mine and I have added to it.

```c
// Game Over

if(global.customLevel and dead and global.repeatLevel)
{
    //Does not calculate fitness for the first run as the Room Start event
    if(not global.firstRun)
    {
        CalculateFitness(global.playerPositionX, global.playerPositionY);
        CheckANN(); //checks to see if the ann needs reconfiguring.
    }
    else
    {
        if(global.runCount == 2)
        {
            global.firstRun = false;
        }
    }
}

GlobalClear Globals();
nextLevel = CheckCurrentLevel();
if (string(nextLevel) != string(global.customLevelName))
{
    global.currentLevel = nextLevel;
    global.firstRun = true;
    global.runCount = 0;
}

nextCustomLevel = nextLevel;
room_goto(rLoadLevel);
```

(Figure 12.3 – Operations after Character Death)
In this code execution script the area that will detect the agent’s death is the one shown above in figure 9.3 and is triggered if the current level is a custom level, the player is dead and the levels have been set to repeat. You can turn repeating levels off in the SetupPlayerVariables script located in the BOTSCRIPTS folder inside the AI Toolset folder inside the Scripts folder although generally you would want the levels to repeat as you would normally be testing more than one agent more than once.

If this code section is triggered then a check is done to see if the agent has been killed the needed number of times to fix the room start bug. If it has then the CalculateFitness function in the DLL is called to calculate the fitness of the current agent. After that the ANN is then checked to see if the ANN needs updating with another population member’s values, if breeding needs to happen and if generation stats need printing.

The CheckCurrentLevel DLL function is then run to see if the level needs changing, if the parents need saving and also checks to see if the fitness function needs updating for the next level.

Afterwards a section of code is ran to check and see if a different test level is being loaded than the one that is currently being used, if so then reset some variables that will cause the next level to reload twice to fix the room start script bug and then load the next level.

If the agent doesn’t die however and successfully manages to go through the exit door then the code execution script inside the Animation End event inside the oPlayer1 object is ran. The code in this section is not all mine and I have added to it. The same functions as shown in figure 9.3 are ran however the level switching code looks a little different as shown in figure 9.4 below. The section of the code that we care about is the code in the else if (global.repeatLevel == true) statement.
if (global.testLevel != "")
{
    scrClearGlobals();
    room_goto(zLevelEditor);
}
else if (global.currLevel == 5) room_goto(zTransition1x);
else if (global.currLevel == 9) room_goto(zTransition2x);
else if (global.currLevel == 13) room_goto(zTransition3x);
else if (global.levelType == 1) room_goto(zTransition2);
else if (global.levelType == 2) room_goto(zTransition3);
else if (global.levelType == 3) room_goto(zTransition4);
else if (global.levelType == 4) room_goto(zTransition4);
else if (global.repeatLevel == true)
{
    nextLevel = CheckCurrentLevel();
    if (string(nextLevel) != string(global.customLevelName))
    {
        global.currentLevel = nextLevel;
        global.firstRun = true;
        global.runCount = 0;
    }
    global.nextCustomLevel = nextLevel;
    room_goto(zLoadLevel);
}
else room_goto(zTransition1);

(Figure 12.4 – Operation after Level Completion)