A project completed as part of the requirements for the BSc (Hons) Computer Games Programming

entitled


By

Josh Hadley

In the years 2015-2016
This project builds upon the original open source Spelunky source code for GameMaker 8. (Mossmouth, 2009)

Abstract

This report investigates the implementation of Forward-Facing Artificial Intelligence (AI) algorithms, in particular the Monte-Carlo Tree Search (MCTS) algorithm, into the SpelunkBots API (Thompson, 2015) before offering a comparative look between a DLL Bot written in C++ and a bot driven by MCTS, to evaluate both the implementation of the MCTS algorithm and its feasibility inside the game, Spelunky, to control AI bots.
Acknowledgements

Thanks to Dr Tommy Thompson for supervising this dissertation.

Special Thanks to Dr Minsi Chen and Dave Voorhis
Table of Contents:

Chapter 1 ........................................................................................................................................... 8  
1.0  Introduction ................................................................................................................................. 8  
1.1  Project Aim .................................................................................................................................. 9  
1.2  Project Objectives ....................................................................................................................... 9  

Chapter 2 ........................................................................................................................................... 10  
2.0  Spelunky Overview ...................................................................................................................... 10  
2.1  ‘Spelunky Dude’ Actor ............................................................................................................... 11  
2.2  Level Environments ................................................................................................................... 13  
2.3  Items and Enemies ...................................................................................................................... 14  

Chapter 3 ........................................................................................................................................... 16  
3.0  Literature Review ....................................................................................................................... 16  
3.1  Existing Game AI Implementations ............................................................................................ 16  
3.2  Monte-Carlo Tree Search .......................................................................................................... 18  
3.3  What does MCTS offer? ............................................................................................................. 19  
3.4  Existing Implementations of MCTS .......................................................................................... 20  
3.4.1. Monte Mario .......................................................................................................................... 20  
3.4.2. Ms Pac-Man .......................................................................................................................... 21  
3.4.3. Total War: Attila ................................................................................................................... 21  
3.4.4. AlphaGo Google DeepMind Project ................................................................................... 22  
3.4.5. The GVG-AI Competition .................................................................................................... 22  
3.5  Evaluation of MCTS Implementations ....................................................................................... 23  
3.6  Considering Spelunky ................................................................................................................ 23  
3.6.1. SpelunkBots Framework ...................................................................................................... 25  

Chapter 4 ........................................................................................................................................... 26  
4.0  Methodology and Implementations ............................................................................................ 26  
4.1  Considerations ............................................................................................................................ 26  
4.1.1. GameMaker ......................................................................................................................... 26  
4.1.2. Utilizing the Spelunkbots API ............................................................................................... 27  
4.1.3. The Problem Space .............................................................................................................. 27  
4.1.4. Implementing the Forward-Model ....................................................................................... 28  
4.1.5. Goal of the AI ....................................................................................................................... 29  
4.2  Implementation ............................................................................................................................ 29
### 4.2.1. MCTS Algorithm

![Image](image-url)

---

### 4.2.2. 'Action' Object

![Image](image-url)

---

### 4.2.3. Action Sequence

![Image](image-url)

---

### Chapter 5

5.0 Testing Conditions

5.1.1. Test Levels

5.1.2. Existing DLL Bots

5.2 Results

5.2.1. Test One

5.2.2. Test Two

5.2.3. Testing Level

5.3 Analysis

5.3.1. Bot Intelligence

5.3.2. Game Performance

5.3.3. Behaviour Analysis

5.4 Summary of the Results

### Chapter 6

6.0 Conclusion

7.0 Recommendations and Improvements

7.1. GameState Modification

7.2. Greedy MCTS

7.3. Better Pruning

7.4. Detailed Debugging Statistics

7.5. Summary of Conclusion

### References

### Bibliography

### Appendices

Appendix A: MCTS Bot Implementation

Appendix B: Botscripts

Appendix C: How to run the MCTS Bot in SpelunkBots
Table of Figures:

| Figure 2.1. Spelunky Main Menu | ................................................................. | 10 |
| Figure 2.2. The Spelunky Dude Sprite | ................................................................. | 11 |
| Figure 2.3. Example of a Rope in use | ................................................................. | 11 |
| Figure 2.4. Example of a situation requiring a bomb | ................................................................. | 12 |
| Figure 2.5. An Example Level | ................................................................. | 13 |
| Figure 2.6. An example of another environment. (Spelunky Wikia, 2009) | ................................................................. | 13 |
| Figure 2.7. An enemy encountered during a level | ................................................................. | 14 |
| Figure 2.8. Example of some enemies that may be encountered | ................................................................. | 14 |
| Figure 2.9. Treasure and Items that may be encountered | ................................................................. | 15 |
| Figure 3.1. Outline of a Monte-Carlo Tree Search. (Chaslot, et al., 2008) | ................................................................. | 18 |
| Figure 3.2. UCT equation | ................................................................. | 19 |
| Figure 4.1 Example of a Search Tree | ................................................................. | 28 |
| Figure 4.2 Action-based State Transition | ................................................................. | 28 |
| Figure 4.3 GameState values | ................................................................. | 29 |
| Figure 4.4 Action Enum Object | ................................................................. | 30 |
| Figure 5.1 Bot Completion of Test One | ................................................................. | 36 |
| Figure 5.2 Ratio of Demonstrated Behaviours during bot failure | ................................................................. | 37 |
| Figure 5.3 Bot Completion of Test Two | ................................................................. | 38 |
| Figure 5.4 Number of nodes available to MCTS before failure | ................................................................. | 39 |
| Figure 5.5 Bot Completion of Testing Level | ................................................................. | 40 |
| Figure 5.6 Average CPU used per Bot | ................................................................. | 42 |
| Figure 5.7 CPU Usage by the MCTS Bot and the ‘JordanBot’ DLL Bot | ................................................................. | 43 |
| Figure 5.8 All Failure Behaviours displayed by Bots | ................................................................. | 44 |
| Figure 5.9 Travel Distance of the Bots against their Height Achieved | ................................................................. | 45 |
| Figure 0.1 MCTS Bot C++ Code | ................................................................. | 55 |
| Figure 0.2 JordanBot script | ................................................................. | 56 |
| Figure 0.3 DiscoveryDan script | ................................................................. | 58 |
| Figure 0.4 FishBot script | ................................................................. | 59 |
| Figure 0.5 GoldDigger script | ................................................................. | 62 |
| Figure 0.6 Indie script | ................................................................. | 64 |
| Figure 0.7 NotSoSolidSnakeBot script | ................................................................. | 66 |
| Figure 0.8 SeanBeanBot Script | ................................................................. | 67 |
| Figure 0.9 PlayerChoice Location | ................................................................. | 68 |
Table of Tables:

Table 1 Game AI Implementations.................................................................17
Table 2 Testing Level Examples and Conditions ...........................................32
Table 3 List of Existing DLL Bots .................................................................35
Chapter 1

1.0 Introduction

The use of Artificial Intelligence (AI) has been a subject of great interest to researchers for many years. As both technology and research improved over the years, AIs gradually have become more complex in nature and can now access areas that previously were deemed either too complex or too expensive to integrate. Through AI Research and Competition, the investigation into what limits an AI algorithm have been driven forward, encouraged by researchers and hobbyists who to try to find the solutions and limits of existing AI algorithms. In particular, Computer Games have become an area of interest when it comes to testing AI algorithms due to their nature of being customisable environments, going from a simple environment such as Ms PacMan, to a more complex and rich environment in modern games such as Spelunky.

Throughout this dissertation, there will be a detailed examination into the forward-facing AI algorithm known as Monte-Carlo Tree Search (MCTS). A forward-facing algorithm essentially looks into the future in order to generate a series of actions that would lead to the best outcome for that scenario. MCTS is one such algorithm which has proven to be effective and has already been utilized in a magnitude of different game types, such as Go (Google DeepMind, 2015) and StarCraft (Computational Intelligence in Games, 2015). Having an AI that is able to simulate a solution for a problem before acting on it is a tantalizing prospect for an AI researcher, which is one of the reasons why MCTS has been chosen for this project, amongst others to be later explored.

There will be a detailed look into what the principles of the algorithm are before attempting to integrate the algorithm into an existing AI framework. The game chosen for the integration is Spelunky created by Derek Yu at Mossmouth Studios (Mossmouth, 2009), which offers a complex range of problems that the AI will have to address. These problems, as well as what makes Spelunky an interesting target platform for the MCTS algorithm, will be discussed in both Chapter 2 and Chapter 3.
1.1 Project Aim

This project aims to investigate the feasibility of using forward-facing algorithms to control bots to play the game of Spelunky, whilst comparing the efficiency of a vanilla MCTS algorithm against a bot that has been written utilizing the SpelunkBots API, created by Daniel Scales and Tommy Thompson (Thompson, 2015) using rule sets in a decision tree format. A key objective is to integrate a vanilla form of the MCTS algorithm into the SpelunkBots API that will control a bot, which will be examined to understand how the algorithm and the bot can handle the complex problem space that Spelunky provides, in the form of both randomly generated and destructible terrain, as well as further game play features. These examinations will then be assessed to observe how the algorithm compares against the DLL bot and determine its final feasibility in such a problem space.

1.2 Project Objectives

- Integrate the MCTS algorithm into the SpelunkBots API.
- Create a MCTS-Driven bot that can be used within Spelunky.
- Compare a MCTS-Driven bot to Existing DLL-Based bots in Spelunky.
Chapter 2

2.0 Spelunky Overview

This chapter provides a brief overview of Spelunky (Mossmouth, 2009) to give an understanding of the relevant core elements of the game. It was originally released in 2009 by Derek Yu, founder of Mossmouth studios, and has gone on to see a high-definition remake in 2012.

The key element that make Spelunky interesting from an AI point of view is that it offers a great deal of variety. The levels for the game are randomly generated, containing any number of enemies and tools to use, whilst offering a variety of element tiles that the player has to navigate through. When this randomness is combined with the fact that the game is built as a platformer that can operated in three dimensions, it presents an excellent challenge to program an AI for.

Whilst comparing the gameplay of Spelunky to other popular platformers of a similar style, such as Super Mario Brothers (Nintendo, 1993), it is easy to see the similarity in different aspects. The unique feature of Spelunky within the genre how it approaches the death mechanic. If the player dies, they are forced to restart the entire game from the beginning, resulting in the loss of all the items they may have acquired. This presents a certain ‘roguelike’ challenge to the game and makes repetition through the game more interesting due to its random nature.

Figure 2.1. Spelunky Main Menu
2.1 ‘Spelunky Dude’ Actor

Figure 2.2. The Spelunky Dude Sprite

The player takes control of the ‘Spelunky Dude’ throughout the game. As expected with a platformer, there are the standard mechanics of running, jumping and attacking given to the player. On top of these, there are various items that a player can collect through the course of the game that can alter the abilities the player can use, such as utilizing a Jetpack to fly across a gap or using a gun to kill enemies. The limit to the range of abilities the player is able to perform is restricted to the items they can acquire.

Due to the nature of the levels, quite often the player encounters a natural obstacle blocking their path. This can be a fall that is too large to land safely… or the path is being blocked by the environment. In order to scale heights safely, a player has to utilize the ‘rope’ mechanic which allows them to use a rope to move down, demonstrated in Figure 2.3. A rope can also be thrown upwards to allow the player to scale upwards instead.

Figure 2.3. Example of a Rope in use.
As aforementioned in the previous paragraph, due to the random nature of the level generator, the player’s route to the end of the level or to a lucrative tool might become blocked by the terrain. Spelunky’s terrain is entirely destructible which allows the player to destroy sections to access new areas. This is primarily achieved by using the ‘Bomb’ mechanic which destroys blocks in its radius, as demonstrated in Figure 2.4.

Both of these mechanics are an essential part of the game, however they both come in limited supply and need to be resupplied constantly through the game. This forces the player to calculate when it is best to use these items, which accentuates the challenge of programming an AI to play this game.

Figure 2.4. Example of a situation requiring a bomb.
2.2 Level Environments

As the player begins a level, it will almost be entirely unique in terms of layout due to the random generation of the environment, enemies and items found within it. The player will always start at either a closed or open door, with the goal to find an exit door to progress to the end of the level. The Exit Door’s location is also completely random, so the player is unable to predict based on a pattern to find the door. This is a great challenge for an AI to solve due to the fact it has no basis for the direction it is meant to go.

As the player progresses through the game, they become exposed to more and more unique areas with their own environments. Some example of these include lava, thin ice and loose platforms. These are classed as ‘Traps’ alongside more traditional traps such as a ‘spike’ trap. With such diverse environments that can be randomly generated, it is difficult for the player to truly predict what they will find in each new level.

Figure 2.5. An Example Level

Figure 2.6. An example of another environment. (Spelunky Wikia, 2009)
2.3 Items and Enemies

Throughout the game, the player can encounter unique enemies, a few examples are given in Figure 8, which will try to kill the player. In earlier levels, these enemies are quite simple and are easily predictable as not to pose such a big threat to the player. However as the player delves deeper into the game new, more difficult, enemies are created to challenge the player. Some of these enemies are specific to certain areas, which means a player might not actually be exposed to some of the enemies until they progress to said area. This provides another challenge to the player, who not only has to deal with a randomised level but also progressively more complex enemies.

Figure 2.7. An enemy encountered during a level.

Figure 2.8. Example of some enemies that may be encountered
Aside from encountering more threats, Spelunky also provides a range of treasures and tools that the player can either find throughout the game or purchase from the ‘Shop Keepers’ present in some levels. Treasure can be collected through levels to award ‘cash’ to the player which they can then use to spend in or save to contribute to the score system.

Tools provide players with unique functionality that would otherwise be unavailable to them. These can be acquired by themselves throughout the level, found in supply crates or found in one of the various shops. While not necessary to the game itself, the tools can give bonuses that provide a significant advantage in gameplay, such as rescuing the ‘Damsel’ which grants an extra life to the player.

Utilizing both of these elements in the game are not in any form necessary, but they do have the possibility to significantly reduce the difficulty of the game which can become crucial towards the late stages. For programming an AI, this is a particular challenge in order to not only ensure they can navigate the game, but to weight the items by their usefulness enough so that they can be taken into consideration.

Figure 2.9. Treasure and Items that may be encountered.
Chapter 3

3.0 Literature Review

In this chapter we will begin to investigate what exactly MCTS is and how it has been implemented into other projects to control a ‘bot’, and to investigate how it can be incorporated into the ‘Spelunkbots’ AI Project for Spelunky. (Thompson, 2015)

We will take a detailed look into the design decisions between the various forms of MCTS and how we can construct a similar framework to allow MCTS to be incorporated into the Spelunkbots project. Whilst doing so, consideration will also be given to the difficulties caused by Spelunky itself that were previously discussed in Chapter 2, which may hinder the introduction of MCTS

3.1 Existing Game AI Implementations

The research performed in this dissertation is not the first time an algorithm has been utilized to control an AI within a game for research purposes. There have been plenty of cases of algorithms being integrated by game AI researchers. An example of a few of previously established research implementations are as follows:

<table>
<thead>
<tr>
<th>AI Implementation</th>
<th>Algorithm</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario AI</td>
<td>Learning Agent</td>
<td>Create a Bot that performs a certain amount of simulations on a ‘Learning Track’, before using the learned behaviours in the real game.</td>
</tr>
<tr>
<td>Mario AI (Togelius, 2009)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StarCraft AI Competition</td>
<td>Bayesian model, Potential Fields and Reinforcement Learning</td>
<td>Play against the StarCraft AI in a 1vs1 RTS Match with a max time of 60 minutes.</td>
</tr>
<tr>
<td>StarCraft AI Competition (Computational Intelligence in Games, 2015)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepMind Doom AI</td>
<td>Reinforcement Learning</td>
<td>Using Visual Inputs to control an AI</td>
</tr>
<tr>
<td>DeepMind Doom AI (Mnih, et al., 2016)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 1 Game AI Implementations

The majority of methods and algorithms used are based around static functions or algorithms that required the AI to learn behaviours through simulated play-through. None of these algorithms however include ‘Planning’, which is the process of creating a sequence of actions that will achieve a certain goal. (Russell & Norvig, 2010).

Yet why now? Why in particular have such complex approaches to AI come into fashion now? The answer revolves around to how far games have developed in the last few years. Coming from games such as Chess (Montoya, 2012) to games where entire armies of AI are co-ordinated together such as StarCraft (Computational Intelligence in Games, 2015) and other Real-Time Strategy games, the problem space in games has become increasingly more complex and dynamic.

As the problem space becomes more and more complex, it provides more opportunity for new algorithms to be tested on what limits they may have. An example would be to compare an AI playing chess, and an AI acting as a general of an army in a three-dimensional environment. Within Chess there is always a set number of moves an AI can make to try to win whilst inside of a 3D environment, the AI has multiple avenues and tactics it could use to try to win, which weren’t present before in the simpler Chess environment. As well as simply allowing more options for an AI to experiment, games can also introduce new problems for an AI to deal with such as destructible or procedurally generated terrain. It is these sort of features in games that make them an excellent platform for AI research.

This is where ‘Planning’ algorithms, such as MCTS, are becoming more popular in research to see how beneficial using a forward-modelling algorithm in a complex problem space can be, and to understand their limitations. The game Spelunky can be considered such a complex problem space, using MCTS is something to be considered with interest.
3.2 Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first search technique which uses stochastic simulations (Chaslot, et al., 2008). MCTS can be applied to any games of a finite length, where the basis is the simulation of games where both the AI-controlled player and its opponents play random moves or pseudo-random moves. Whilst using a single game can give little information to learn, simulating multiple random games can allow a good strategy to be inferred. The algorithm builds and uses a tree of future game states using the following process:

**Selection** While the state is found in the tree, the algorithm uses the currently stored statistics to select either to exploit or explore the tree which it balances. One hand, the task is often to select the game action to the best results (exploitation). On the other hand, less promising actions still have to be explored, due to the uncertainty of evaluation.

**Expansion** When the game reaches the first state that cannot be found in the tree currently, it is added as a new node. By doing so, the tree is expanded by one node per simulated game it performs.

**Simulation** For the rest of the game, random actions are selected until the game ends. Naturally, the adequate weighting of action selection probabilities has a significant effect on the level of play. If all actions are selected with equal probability, the strategy played is often weaker which the Monte-Carlo program to be suboptimal. We can use heuristic knowledge to give large weights to actions that look more promising.

**Backpropagation** After reaching the end of the simulated game, we update each tree node that was traversed during that game. The visit counts are increased and the win/loss ratio is modified according to the outcome.
The selection process is primarily driven by the use of Upper Confidence Bound applied to Trees (UCT), which takes the total value of the node’s weight against the times it has been visited before being checked with the number of times its parent has been visited against a tuneable bias parameter.

\[ \frac{v}{n} + B \times \sqrt{\frac{\ln N}{n}} \]

Figure 3.2. UCT equation

3.3 What does MCTS offer?

Firstly MCTS does not require the AI to have a fixed behaviour when it approaches a problem within a game. Whilst having fixed behaviours, such as a rule-based AI, is great for a static problem, games in nature are dynamic in problems for a range of possibilities. They may include having enemy AIs trying to stop the player, an environment space that is randomly generated or perhaps even both. By using a forward-facing algorithm, the AI can take into account all of these potential problems in the game space by taking the information from the ‘Game State’. By doing so and simulating the game multiple times, the AI can choose the best path that approaches these problems ensuring it is the most optimum path it can take, which is perfect for Real Time Strategy games. (Andruszkiewcz, 2015)

However, there are issues with forward-facing algorithms that can make them difficult to use. Without having full access to the game state, the algorithm is unable to be used with complete efficiency. This is problem when it comes to games where the game state may change, such as if a Fog of War system is used (Andruszkiewcz, 2015). Another issue is that the algorithm itself is very unstable, due to taking a great deal of time and performance in order to simulate the eventual actions of the AI. This is where having more static and pre-made behaviours can result in a faster and simpler way to integrate an AI bot.
3.4 Existing Implementations of MCTS

This section will discuss several existing forms of MCTS that have been implemented into video games as an AI Framework, as either a research tool or as a production AI framework, identifying where and how they implemented MCTS.

3.4.1. Monte Mario

A recent project investigating the use of MCTS in a similar nature to Spelunky was a MCTS implantation into Super Mario Brothers by Emil Jacobsen, Rasmus Greve and Julian Togelius (Jacobsen, et al., 2014). In order to introduce MCTS as an AI agent into the game, it was quickly realised that the default version of MCTS would not be compatible due to the nature of the action space. The ‘Mario’ would be unable to navigate a level efficiently, either running into enemies, falling down gaps or being unable to jump over objects. To solve this, they experimented with multiple modifications to the algorithm to improve its performance. These modifications are worth noting.

One modification performed was calculating a Mixmax reward for the level itself. This allowed the agent to calculate the best node which provided the ‘max’ reward path, instead of calculating the path based on the average value. By doing so, it helped the algorithm to quickly calculate the path which would be continuously adjusted as the game state changes i.e. a new enemy enters the level, new paths are discovered.

Another modification is the use of Macro Actions. Macro actions are a sequence of actions that can be executed in a sequence (Powley, et al., 2012). Using Macro-Actions to form nodes within the tree allowed MCTS to further expand the tree in Monte Mario, which allowed it to find more potential solutions before deciding the final path. However, an issue that was discovered was that using macro actions allowed the AI to effectively ‘get stuck’ due to the now coarse nature of the tree nodes and the repetition of actions. This issue was negated by micro-planning and specifying a distance threshold that would overrule some macros, such as when the AI is too close to an enemy it should try to avoid it.

A final modification worth noting from the study is use of Partial Expansion to assist MCTS in simulating the next action for Mario to take. The process itself is very time consuming. In order to reduce this, the tree is ‘progressively unpruned’ to reduce the branching factor of MCTS artificially, whilst allowing it to be unpruned when there is more time (Chaslot, et al.,
By doing so, the UCT factor of MCTS can now choose to exploit a promising path or expand another action that is unexplored. This is further controlled by fine-tuning the urgency of expanding a new node, which can help focus the algorithm without unnecessary branching.

3.4.2. Ms Pac-Man

A popular platform for AI frameworks is the Ms Pac-man game. Several studies have been performed into seeing how MCTS can be incorporated into the game, such as the work done by Ikehata (Ikehata & Ito, 2011) and Simon Lucas (Alhejali & Lucas, 2013). The implementation proposed by Ikehata was to help Pac-man survive the unpredictable pincer movements of the ghosts in the game. It utilized MCTS to evaluate the necessary path to avoid the ghosts without becoming trapped. Alongside this, MCTS took into account the ability to eat ‘pellets’ in order to defeat ghosts. The simulations performed using the proposed MCTS to solve the issue in fact proved to be so successful that it established a new world record at the Ms. Pac-Man Competition. However, despite its success, it was noted that the accuracy of the algorithm could be improved, including also autonomously collecting parameters to allow the AI agent to play by itself through real-time learning, which would improve performance.

3.4.3. Total War: Attila

While the previous two implementations have been focused on a 2D environment, one excellent demonstration of MCTS in a production environment is the implementation of MCTS used within Total War: Attila by Creative Assembly. (Andruszkiewicz, 2015) Here MCTS was used to control the tactical co-ordination of the various NPC agents used within Total War. The reasons behind the use of MCTS highlighted by Creative Assembly were that the algorithm allowed fine-grained control of performance, making it easily extensible and suitable for a large search space.

However, the vanilla form of MCTS still was not a perfect solution for the scope of the game. The algorithm was improved through multiple steps such as pruning nodes that were either impossible in nature and actions that were unlikely to succeed, enforcing a node and tree structure to improve efficiency and Lazy evaluation of paths to find the best possible sequence of actions against the current action sequence.
3.4.4. **AlphaGo Google DeepMind Project**

A recent project completed by the Google DeepMind Project was the development of the AlphaGo AI to play the game ‘Go’. The project achieved critical success due to being the first AI to defeat a professional human player at the game, which was feat considered impossible with previous AI only achieving an ‘amateur’ level of play. (Google DeepMind, 2015).

To achieve this feat, the AlphaGo AI incorporates a great deal of sophisticated AI algorithms such as Deep Neural Networking and Reinforcement Learning. However what makes this AI interesting to this project is that it utilizes MCTS purely for its ability to select the best valued ‘action’. (Silver, et al., 2016) The algorithm itself was only utilized for the move selection of the AlphaGo AI, with the evaluation being handled by the Neural Networking.

By using MCTS alone it likely would not have been successful, however the AlphaGo AI demonstrated how efficient the algorithm is, which in combination with the other elements it allowed the AlphaGo AI to achieve the rare feat of not only defeating a professional human player, but also to later defeat a player considered to be one of the world’s best players of Go. (Google DeepMind, 2015)

3.4.5. **The GVG-AI Competition**

Another area of interest is the GVG-AI Competition, run by Diego Perez et al., which is an AI competition dedicated to exploring the problem of creating controllers for general video game playing. (Perez, 2016)

Last year’s competition ran only with the ‘Planning’ track being open for competitors to compete in. This meant that by the end of the competition, the top-ranked AI would effectively be the best Planning AI put forward that year. Throughout the competition, AI Bots that incorporated MCTS were found towards the winning half of the entries, even winning in the case of the ‘Yolobot’ which consisted of a combined approach between MCTS, Breadth First Search and targeting heuristics. (Perez, et al., 2016)

With multiple bots using MCTS either fully or partially, the fact that these MCTS-Driven bots were dominating the field implies how effective the algorithm is as a ‘Planning’ algorithm, beating other algorithms in its field.
3.5 Evaluation of MCTS Implementations

When we look at all three cases of implementing MCTS in different scopes, we can easily identify a theme in the way that they were implemented.

The initial three implementations were used for its ability to handle complex search spaces efficiently whilst not being too expensive to use, however in its basic state MCTS proves to be far less efficient compared another algorithm due to the complexity of the algorithm. Yet with each case, when the MCTS tweaked alongside common improvements such as Minimax, Tree pruning and structuring the tree, the algorithm has shown to be far more efficient than a common algorithm.

Yet even while showing promise in controlling AIs, MCTS has demonstrated its efficiency in being incorporated partially as being part of a larger system. Being able to perform as part of a greater system demonstrates its flexibility and versatility as an algorithm, breaking down its core elements and thriving. This adaptability in its use and scale of environments in a wide variety of games, from the simpler Ms Pac-Man to controlling multiple AI agents inside of Total War: Attila, the MCTS algorithm showcases its ability to be integrated into differing game problem spaces.

It is this feedback and case points that highlight MCTS as an excellent candidate for controlling an AI agent in Spelunky.

3.6 Considering Spelunky

Spelunky offers an interesting problem for an AI. The very nature of the game is a far more complex environment compared to static environments such as a Mario level or a map in StarCraft. While the level size is limited in Spelunky, the entire content within it is randomly generated each time through the game, ensuring that no matter how many times the game is played, you will be unable to play the same level twice. When you take into account that this can result in a range of different enemies with a range of different behaviours also being generated, alongside traps and obstacles such as a wall blocking the path, the problem space that an AI would have to face is incredibly complex.
Also as mentioned previously, the game rogue-like conditions not only provide a difficult challenge for players, but it also provides a source of trouble for an AI. Combining this alongside the three dimensional movement and ‘fog of war’ state, it is crucial that any AI agent can be able to navigate and calculate the best action sequence it can quickly depending on the action space of the game state.

Using pre-determined rules and behaviours in an AI bot within Spelunky is almost certainly guaranteed to encounter a situation to which its’ fixed behaviour cannot adjust to the problem space that is generated. This is where MCTS shows promise, having already proven to be able to handle a diverse range of state problems, from a small scope to a very intensive production setting, which makes it an excellent candidate to run the most efficient AI agent.

For example, an algorithm such as A* (Patel, 2009) would be sufficient for allowing the AI to navigate across the generated dungeon provided. However it would not be able to handle actions such as picking up a gun and using it against an enemy, whilst also using a rope to scale down a cliff. This is not a fault of the algorithm itself, but the handling of the environment and action space. MCTS would allow the agent to calculate the best sequence of actions which may include all of the mentioned ones from the example, which would save having to add more complication to a simple algorithm. Further areas are identified in research performed by Dr Tommy Thompson (Thompson, 2015) which further iterates not only the physical challenges present in the game but also the logic issues an AI must deal with such as utilizing an item to skip past a certain section of the game or to use a rope or bomb to access a ‘locked’ area by terrain to find certain resources that can potentially aid the AI or harm it if the detour is not fruitful.

In order to test this application, it is necessary to be able to compare the performance of an MCTS agent against another AI agent within Spelunky. Fortunately there has already been work on introducing an AI agent into Spelunky which will be used to provide the comparison.
3.6.1. **SpelunkBots Framework**

SpelunkBots is a C++ DLL and GML API that allows individuals to create an AI Bot to play through Spelunky developed by Daniel Scales and Dr Tommy Thompson (Scales & Thompson, 2014). The concept of the API is to allow the information given to the Bot to be as accurate to that of a human player as possible. Currently the Framework allows the developers to create a set list of pre-determined commands using Booleans to allow the AI Bot to interact with the level. The control variables are reset after each frame once the script provided by the player is applied. This allows the developer to experiment with a range of actions to see which sequence provides the best behaviour.

Also, the framework has an inbuilt A* search algorithm to control the path finding of the bot. It utilizes a grid system where each tile is 16x16 pixels and is mapped directly to the actual tile method used in Spelunky. This has been shown to provide good results when used with a Bot, however an element that causes issues is the ‘Fog of War’ system used in Spelunky. It causes a section of the level to be hidden from the player, which prevents both them and an algorithm from knowing what lies ahead. The A* algorithm is easily able to expand the navigation system, using a simple ‘0-1’ scale for traversable terrain, however the case for repetitive routes is likely due to the new nodes being unlocked as the Bot travels through the map. The Framework also provides several useful testing and debug scripts that allow developers to easily produce experimental test conditions for the bots, especially when combined with built-in level editor for Spelunky. This makes the API not only an excellent place to begin developing a MCTS based Bot, but it also provides an excellent source of testing against existing and experimental AI Bots created through the API.

However one area the framework does address is that an algorithm such as MCTS may be hindered by Spelunky’s nature due to MCTS being forward-facing. Being unable to access all the information of the game state and action space may hinder the algorithm significantly to the point it is inefficient, yet as the implementation by Creative Assembly has shown, the algorithm can be able to deal with limited information as long as precautions and modifications are made to the vanilla algorithm.
Chapter 4

4.0 Methodology and Implementations

Having looked into both MCTS and its possible application within Spelunky, this Chapter will address the considerations that were taken both before and while implementing a MCTS framework into Spelunky, whilst detailing the process of introducing the algorithm into the game.

4.1 Considerations

Before discussing how the MCTS driven AI Bot will be implemented, there are several factors that need to be taken into consideration that may affect the development of the Bot, either directly or discretely.

4.1.1. GameMaker

One core consideration that needs to be addressed is the development tool that was used to create Spelunky. The tool used is known as GameMaker and is quite efficient to make small scale games. However, the most important part of GameMaker concerning the development of the MCTS Bot is the treatment of extension files to GameMaker. GameMaker utilizes its own scripting language known as 'GameMaker Language (GML)' which is the only language available to use within the engine. GameMaker allows .DLL files to be included with the project to be used, but they can only return either a double or string variable. This causes issues regarding how we can approach developing a fairly complex MCTS algorithm, but fortunately this stepping stone has been partially fixed by the work done on the SpelunkBots API.
4.1.2. Utilizing the Spelunkbots API

The SpelunkBots API as previously mentioned in the last section allows users to create their own bots by using the .DLL provided. The API relies on the use of Global variables within Spelunky, which it accesses and then returns to GameMaker to drive the AI Bot. The user can create their own scripts by either using GML or the C++ DLL. While these two options provide not very much gains in terms of flexibility, users are also able to create their own Bots using C++ alone. This is caused by the user inheriting from the provided interface class of the API and allows users to override the internal 'Update' tick function of Spelunky. By doing so, it allows users complete control of the Bot without having to use an interface between the .DLL and GML. This is perfect for the development of the MCTS Bot as it allows us complete control of its functions within C++ alone, meaning we can create and run the algorithm separately within the project as long as the Interface 'IBot.h' class is inherited.

4.1.3. The Problem Space

Another important consideration is our approach to the Problem Space of Spelunky. Unlike previous AI projects which have a fixed environment, Spelunky's environment is always randomly generated and is also partially hidden by a 'Fog of War'. Previous MCTS implementations have almost entirely relied either knowing the entire problem space before using the algorithm or knowing there is a case of bias in the level design, such as the right-side bias discovered in Jacobsen’s Mario MCTS Bot which takes advantage of the fact that the end goal will always be to the right of the bot. (Jacobsen, et al., 2014)

In order for the MCTS Bot of Spelunky to work, the problem space of Spelunky needs to be categorised in a way that the MCTS algorithm can calculate the best route. A way of exploring this answer is to use a driving 'End Goal' behind the Algorithm, which either provides a location based upon the Exit Door within Spelunky being within sight or returning the largest section of concealed space that has yet to be explored. In order to achieve this, the weighting of the nodes also needs to be adjusted accordingly.
4.1.4. Implementing the Forward-Model

A large part of the ethos of MCTS is its ability to simulate actions, essentially predict the best path in future gameplay as the game progresses. In order to fully achieve this part of the algorithm, the ‘GameState’ that is used to control the Bot must be kept every time the algorithm is cycled and cloned.

In order to keep the GameState as simple as it can possibly be to achieve an entry point framework, I decided to create a nested class within the MCTS bot called ‘TreeNode’. The logic behind this decision is that by driving the algorithm separately to the main class and using a ‘Node’ object, the transition between states should be kept simple without compromising the algorithm. By taking account for the ‘GameState’ in a separate node used in the bot algorithm, the GameState is effectively cloned with each iteration of the algorithm in order to produce the optimum action based on the current state of the bot, which is demonstrated in Appendix A.

For my own implementation of the algorithm, I treated the search tree as searching through an expanded Linked List, where a State transitions into another based upon performing an action, encompassing the graph based design.

![Figure 4.1 Example of a Search Tree](image)

![Figure 4.2 Action-based State Transition](image)
Whilst simplistic in nature, treating the state transitions as such fits within the parameters of Spelunky’s design. By taking account for the individual node positions throughout the level, the level itself can be turned into a navigation grid which the algorithm can utilize as it runs to help control the bot.

4.1.5. Goal of the AI

Another aspect to consider is what will drive the AI. It could range on multiple conditions such as achieving the best time, collecting the largest amount of treasure it can or staying alive as long as possible.

For this implementation of the algorithm, the goal of the AI is simply Navigation. The only problem the AI will consider is navigating from its starting point and travelling to the end point. This will allow the simplest implementation, which offers the greatest flexibility when it comes to testing the capabilities of the algorithm.

4.2 Implementation

In the following section, the implementation of the MCTS algorithm into the 'Bot' DLL provided with SpelunkBots API will be explained in depth, as well as the modifications and other features that have been added in order to run the algorithm.

4.2.1. MCTS Algorithm

The MCTS implementation of algorithm is inspired by Simon Lucas' Java implementation. (Lucas, 2015) The implementation operates under the same assumption that the Game State drives the processing of the AI, which moves to a new Game State with a series of examples.

In order for the MCTS algorithm to work efficiently, the implementation of the representation of the Game State was aimed at being as simple as possible, but could easily be expanded by users. To this end, the Game State is represented solely as the current position occupied by the Bot (The Node Position of the player) and whether the Bot has an End Goal in sight.

```c
//GameState Values
double positionX;
double positionY;
double endGoal;
Action action;
```

Figure 4.3 GameState values
The algorithm itself is written in C++, utilizing the principles described in Chapter 3 alongside UCT. This implementation is a raw take of the algorithm, essentially acting as a 'Vanilla' model for future investigation. By using a form of UCT, the MCTS algorithm should conform at least to the best weighting when it moves through the selection phase.

\[
\frac{S_{n+1}}{S_n} = M \cdot \frac{S_n}{S_n}
\]

Equation 1 Implemented State Selection based on the calculated UCT

By using this basis, the ‘Best Value’ that would compare to the UCT value calculated through the selection phase can be adjusted to suit the algorithm with refinement. In this case, the best possible outcome the Bot can achieve is finding the ‘Exit Door’ within a level. Any action that allows great exploration (such as Moving and Jumping) would have more contribution than an action that was not based on navigation, so taking into account this fact there is a weighting bias towards movement alone.

4.2.2. 'Action' Object

Firstly the Action object is a simple enum value that dictates what action the Bot should be performing. This is automatically assigned to the node during the expansion phase of an existing node, which is based against the position of the previous node.

```cpp
enum Action {
    Standing,
    MoveLeft,
    MoveRight,
    JumpLeft,
    JumpRight,
    Duck,
    Attack
};
```

Figure 4.4 Action Enum Object

For the MCTS Bot, currently the existing Action represents an idle stance, x-axis movement (Left and Right), y-axis (Jump and Fall) and Attack action. These actions summarise the primitive AI behaviours that we can expect from the basic Bot.
4.2.3. Action Sequence

After running the MCTS algorithm to simulate its play-through, the tree returns its’ optimum action in order to be run through the ‘Update()’ function to play the bot, which can be seen in Appendix A. By creating a sequence based from the action, it allows better visualisation whilst also allowing the API to better handle the logic provided with the algorithm.
Chapter 5

5.0 Testing Conditions

In order to test the purpose of the MCTS bot, as described in Chapter 1, and collect relevant test data to demonstrate if the MCTS bot can successfully utilize MCTS to drive the bot, whilst also comparing its performance to an existing DLL bot. Several test environments were created to test the depth of ability the bot possesses.

5.1.1. Test Levels

Designing the test levels was based around the goal to identify the limitations in the current implementation of the algorithm, whilst comparing the performance of the bot to an existing DLL bot.

<table>
<thead>
<tr>
<th>Level Name</th>
<th>Image</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test One</td>
<td><img src="image1.png" alt="Image" /></td>
<td>Testing if the bot can perform a simple movement</td>
</tr>
<tr>
<td>Test Two</td>
<td><img src="image2.png" alt="Image" /></td>
<td>Testing if the Bot can cope with a complex movement</td>
</tr>
<tr>
<td>Testing Level</td>
<td><img src="image3.png" alt="Image" /></td>
<td>Testing the Bot in a complex environment</td>
</tr>
</tbody>
</table>

Table 2 Testing Level Examples and Conditions
1. **Test One**: This level is simple in design consisting of only three tiles to see if the bot can perform basic movement. From a behavioural point of view, we expect to see the bot simply move from its’ starting state to the exit door.

2. **Test Two**: Like ‘Test One’, this level relies on a simple design but includes a sudden change in pattern, i.e. a fall beyond the calculation range, to test whether the sudden change in environment will cause the bot to break or if it will be able to handle the complexity. Again the expected behaviour to see from the AI is for it to run smoothly across the level, drop down, before continuing to reach the end.

3. **Testing Level**: Unlike the previous two tests, this level was pre-built to test the functionality of the various DLL bots before entering the live game. It features the minimal actions for a bot to essentially complete a level of Spelunky, including jumping, climbing and attacking. For a successful bot, the behaviour we are looking for is for it to cleanly navigate across the plane, jumping and climbing the obstacle, before navigating down and attacking a few enemies in front of the exit.

With each iteration the bot will be tracked against ‘Tile Distance’ achieved, checking if the bot is able to complete the level, alongside the time taken to complete a run of the level. Existing DLL bots will also be used to check the level and will also be recorded for comparative purposes with the results of the MCTS bot. On top of this, statistics will be collected relevant to how the application handles the usage of the algorithm whilst the bot runs.

Based upon these results, any weaknesses or strengths of the bot can be observed and compared against the statistics retrieved, identifying both the perceived intelligence of the bot and how efficient the algorithm is.
5.1.2. Existing DLL Bots

In order to test and compare the performance of the MCTS bot, part of the testing will also include a run through of the testing conditions with existing bots created with the SpelunkBots API, which can be seen in Appendix B.

<table>
<thead>
<tr>
<th>Bots Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoldDigger</td>
</tr>
<tr>
<td>Indie</td>
</tr>
<tr>
<td>JordanBot</td>
</tr>
<tr>
<td>SeanBot</td>
</tr>
<tr>
<td>NotSoSolidSnakeBot</td>
</tr>
<tr>
<td>FishBot</td>
</tr>
<tr>
<td>DiscoveryDanBot</td>
</tr>
</tbody>
</table>

Table 3 List of Existing DLL Bots

By comparing the performance of these bots with that of the MCTS bot, the results of both bots can be analysed to see not only which script performs better in the game, but what advantages and disadvantages each approach brings to the framework.
5.2 Results

With the explanation of the testing conditions, testing was conducted on the bots in the test environments to analyse both their ability to ‘Play’ the game, as well as their performance.

5.2.1. Test One

![Bot Completion](image)

**Figure 5.1 Bot Completion of Test One**

To begin with the initial test, the only sole bot to function in the level as expected was in fact the MCTS Bot, completing the level within the time of a second. The other DLL Bots either did not move at all or were stuck in an opposite movement away from the door. This was an unexpected result due to the nature of the test, seeing that the others all failed completely. However whilst able to navigate through the level, the MCTS Bot was unable to complete the level by not entering the ‘Exit Door’. This likely is due to a lack of catching during the states transitions or it was not taken account for in the implementation as originally planned.
The MCTS bot itself had little effect on the application, demonstrating an increase in CPU usage of only 5% whereas some of the failed DLL bots demonstrated increases of 5% and more. Between the bots that failed there was an identifiable pattern of behaviours that were demonstrated.

![Failure Behaviours Diagram](image)

**Figure 5.2 Ratio of Demonstrated Behaviours during bot failure.**

In the majority of cases, when the DLL bot failed it displayed a static behaviour and remained idle. What this suggests is that the AI navigation for the bot for some issue was broken. A likely candidate for the failure was that the bot was designed with a specific rule-set that did not accommodate for the structure of the level. Another potential candidate is that the navigation was not assigned correctly for the AI to react to, such as the GoldDigger bot which primarily seeks out treasure items. It was expected for it to react different to a navigation-focused Bot, but for all to remain broken was a true surprise.

Observing the feedback from the bot in the Debug Console Window, evidence supported the breaking of the navigation with every bot, except for the JordanBot and DiscoveryDan bot, due to the feedback remaining unchanged. The MCTS bot demonstrated continual feedback to the console implying it was able to take account for the environmental factors and act accordingly.
5.2.2. Test Two

When it came to the second test, every bot failed to complete the level. The only one that seemed to show any progress however was the MCTS bot, which was able to make it halfway through the level until shortly after the fall section to which the bot and application became unresponsive. Whilst demonstrating the only amount of success in the game, the MCTS bot failed to be continually run when presented with the scenario whereas its failed counterparts did not cause the application to crash.

From a systems point of view, the only true failure for this test was the MCTS bot due to the program becoming unresponsive, with the others demonstrating only a slightly higher CPU Usage than the previous test. There were no previous indicators of a memory leak or overuse of the CPU, which implies that the algorithm itself must have encountered an error.

In order to isolate what the cause of the application crash was, the bot was examined by the amount of nodes available during the duration of the section the bot was able to run. For the initial second, the bot only has to deal with a maximum of two states. However when the bot encounters three available nodes, the bot pauses and the application becomes unresponsive which matches the duration of the working bot.
Figure 5.4 Number of nodes available to MCTS before failure.

This heavily implies that the bot is unable to successfully prune the search tree efficiently to return an action to the bot’s update function. Another possibility is that the tree is continually expanding nodes, which would cause an infinite loop, due to not having sufficient rule-sets to exit the tree.

Despite the issue with the node management, the MCTS bot’s performance prior to the unresponsiveness performs as expected. It is able to successfully proceed onward towards the falling point, even successfully falling and continuing onwards for a few moments. The DLL bots again displayed similar behaviours as within the first test.
5.2.3. Testing Level

![Bot Completion](image)

**Figure 5.5 Bot Completion of Testing Level**

With the final test, again there were repeated results in the case of the DLL bots, but now with the exception that one DLL bot, JordanBot, also partially managed to succeed in playing the level.

Both the MCTS bot and JordanBot were able to initially move successfully. The MCTS bot however, like in Test Two, quickly became unresponsive in similar conditions as before. The JordanBot however managed to proceed further than the MCTS bot before becoming stuck on geometry without becoming unresponsive.

Yet, whilst the JordanBot was able to travel further, it did not display the same behaviour as the MCTS bot did before it became unresponsive. The JordanBot travelled by ‘Jumping’ right repeatedly. Despite being a successful behaviour to perform, it was not the realistic behaviour that was expected from the bot. In this viewpoint, the MCTS bot performed a more realistic response behaviour by running across the flat ground.

In terms of bot performance affecting the system, all bots had significantly more impact on the program with this particular level but the effects remained consistent throughout testing.
5.3 Analysis

After finalising the test results, it was important they were critically analysed to understand the strengths and weaknesses of the implementation to understand if it truly has met the requirements that were aimed for the implementation. As well as finding out more in-depth information, it is a good opportunity to see how well the implementation compares to the existing Bot DLLs.

5.3.1. Bot Intelligence

Whilst implementing the MCTS algorithm, there was concern for how the algorithm would affect the performance of the game whilst running. Initially, this was the case with the algorithm being difficult when it came to implementing it into the API framework of SpelunkBots. Having to rely on the Game Tick using the Update function, the concern was that the speed of the algorithm would have an adverse effect on both the Game itself and the Bot’s ability to predict its future movement.

However when it comes to ‘controlling’ the bot, its behaviour is very simplistic. Playing through the levels multiple times, the bot becomes stuck on the first obstacle it comes across. The reasons behind this likely revolve around the lack of complex searching when it comes to the ‘Expanding’ phase of the MCTS algorithm. Making the framework as simplistic as possible may have inadvertently debilitated the bot’s ability to compute the state it is in. Another likely reason is that the handling of the GameState that was implemented was both inefficient or is not being handled correctly. With this limited functionality, the algorithm suffers in its inability to simulate in greater detail.

When this is compared to the C++ DLL bots however, is there a noticeable difference? In some cases there is an arguable difference. Many of the bots that were tested alongside the bot with MCTS have a specific purpose that doesn’t revolve around pure navigation. Each bot has its own set of hard-coded rules that govern its decisions. Because one does not work in test environments provided does not necessarily mean it would not work with its functionality in another. The problem however lies with the nature of the rule-based system. It is a fixed structure that only works inside of the rules it is aware of. If the bot encounters or escapes this rule set, its entire logic will break. This was demonstrated with the bots that continued to move but were stuck during testing.
In this situation, MCTS demonstrated its main advantage over the rule-based system. The continuous feedback to the debug console supplied in Test One shows that the logic flow of the MCTS bot doesn’t break when it becomes stuck. It continues to simulate based on the current logic it adheres to. This gives the bot the chance to work its way out of the situation, which combined with the nature of forward-modelling, is a provocative solution to a bot that could play ‘Spelunky’ comfortably.

5.3.2. Game Performance

Another important part of the testing phase was to examine how much the algorithm affected the application. Whilst conducting the tests, the CPU usage and FPS of the ‘SpelunkBots’ application were monitored, utilizing the GameMaker Debug Tool and Performance Monitor Tool, to identify any significant changes caused by the individual Bots.

![Average CPU Used](image)

After each level was tested by each bot, the average CPU usage of each bot was calculated by the increase in CPU requirements whilst running the bot. All the bots were able to keep the system running at 30 FPS. Looking specifically at the CPU usage as shown in Figure 5.6, the information provided conforms to the implementation detail that was considered and expected. While running the MCTS algorithm did provide one of the largest CPU increases compared to the other bots, the difference between the algorithm and the C++ DLL bots was marginal.
This observation implies the implemented MCTS algorithm can be run quite comfortably at runtime, which is a positive outlook for any future research into the algorithm. But what must be considered is that while having a manageable system usage is a positive sign, keeping the system usage too low might imply that the algorithm isn’t able to use its full potential. This is particularly evident when the algorithm is unable to deal with more than three node states at a time. If the calculation was more detailed, perhaps the problem would no longer exist at the cost of a bit more CPU usage.

![CPU Usage Per Section](image)

**Figure 5.7 CPU Usage by the MCTS Bot and the ‘JordanBot’ DLL Bot**

To illustrate this fact, when we compare the best two performing bots from these tests, the MCTS bot and the JordanBot, we are presented with an interesting image as shown in Figure 5.7. In Test One, the MCTS bot outperforms the JordanBot marginally, whereas in both Test Two and the Testing Level the results indicate the JordanBot outperforms the MCTS bot. When comparing the pair when they were both operational, the JordanBot actually performed better whilst using a simpler design and lower CPU usage. From a system’s point of view the JordanBot is a more viable choice to use in terms of processing power. But if we take into consideration what MCTS offers in comparison to a simple decision tree, the slight difference in CPU usage could be considered as the cost for a more powerful algorithm. Fortunately the algorithm after a few edits and modifications, to take account for the various variables available to the algorithm from the API, is able to return a value that allows the bot to engage with the game.
5.3.3. Behaviour Analysis

The Behaviours demonstrated by an AI can be a useful insight into how the AI works. During the tests, the predominant behaviour of the bots was recorded in order for their behaviour to be compared against the rest.

The observed behaviours showed that across the tests, the most common behaviours observed were either the bot remained stationary looking in a specific direction or it was stuck performing a particular movement. For the lack of movement, it can be inferred that the Botscript could not begin due to a lack of ‘Trigger Condition’ that would allow the bot to work, thus it demonstrated a default behaviour. This makes sense for the bots that were designed around collecting items, which were the majority of broken individuals in the test phase. For those that continuously moved, an assumption we can make is that the logic circuit for the Bot was entered correctly, however it was unable to operate fully due to condition failing which caused the looping animation of the Spelunky Actor.

Figure 5.8 All Failure Behaviours displayed by Bots
Approaching the two bots that demonstrated some success in the tests, the JordanBot and the MCTS Bot, an analytical look into how they compared to one another was taken. An important part of the MCTS algorithm is that it simulates the ‘Best’ path of actions it can take, which is determined in how it is implemented. A good indicator from a behavioural perspective for the algorithm to show this would be for it to not perform any unnecessary actions. Taking a closer look into where both bots were most successful, the behaviour of both were compared against the distance they took in a straight line against the height they took. For a flat level, the optimum behaviour would be for the bot to continue running against the ground.

As Figure 5.9 describes the JordanBot overall travelled further than the MCTS bot, yet whilst it was more successful, it demonstrated a divergent behaviour from what was expected. In truth the JordanBot demonstrated a continuous jumping behaviour to the direction of its right, whereas the MCTS bot that travelled shorter did not diverge from this behaviour.
So, from a behavioural point of view, which bot was the best? The optimum behaviour from the more successful JordanBot shows that whilst flawed its behaviour led to it being more successful compared to the MCTS bot. The MCTS bot on the other hand demonstrated a more realistic behaviour, which did not perform as greatly as the JordanBot. The reasons behind the MCTS bot behaving more realistically could also be biased due to a poor implementation of the algorithm and likewise the behaviour of the JordanBot also could be more successful in other test conditions. It is a matter of perspective on what the viewer deems as its required goals.

5.4 Summary of the Results

Through observing and repetitive testing of both the DLL and the bots, the analysis provides confidence in that the framework of MCTS algorithm is able to interact with Spelunky via the SpelunkBots API. Whilst the bot has demonstrated a limited and simplistic behaviour prone to issues identified with its implementation, the times of partial success have shown the potential of the algorithm showing comparable performance to the existing C++ DLL bots, which in certain cases the MCTS bot was succeed where the others could not.
Chapter 6

6.0 Conclusion

When looking at the aims of the project given in Chapter 2.3, the primary goal of the project was to investigate if the forward-facing MCTS algorithm could be implemented into Spelunky to control an AI bot that would be able to respond to the level in some manner. Whilst the implementation is simplistic in nature, it both provides an easy to understand presentation of the algorithm and provides an example of how someone could go about creating a MCTS bot in a tutorial manner. In this aspect, the aim was achieved for the implementation.

A secondary aim for the project was to also compare the performance of an existing Botscript written for the SpelunkBots API against the MCTS bot, to see the differences between the two approaches and to verify any strengths the algorithm can offer.

Throughout the testing, as described in Chapter 5, the implementation of the algorithm was exposed and could be analysed thoroughly. It was clear from the results that while the algorithm had showcased some success, the implementation was still flawed when it came to the expansion phase resulting in it becoming unresponsive. This was the major weakness of the implementation, but despite being limited during this phase, the algorithm was still able to provide input to towards the Bot in some manner.

The MCTS bot’s performance at first was both a disappointment in its ability to play the game whilst showing promise in system performance of the algorithm. Seeing the MCTS bot perform against the C++ DLL bots was enlightening to see it demonstrate both a similar performance in both the game and their play-through, but what caught attention the most was that the MCTS bot did not stop running the algorithm when it got caught out by the game whereas the C++ DLL bot did.

The fact that the MCTS bot continued demonstrated the true potential of the algorithm, which is the fact it is autonomous in nature. With the problems of procedural generation in Spelunky’s environment, it is a safe assumption to say that while a rule-based DLL bot can run the game successfully, there will always be one point where the conditions of the game are outside the scope of the DLL bot, causing it to break and fail playing the game.
In the case of MCTS, this would be entirely avoided. Being able to analyse the problem space and the current GameState to simulate potential paths through the game allows the bot to not be governed by static rules, it can take into account of the changing conditions and address them accordingly. While there will be cases where the algorithm will fail and the bot will die, these are the cases that would identify areas the algorithm could not predict highlighting potential research to further improve the algorithm.

In hindsight, the project highlights the potential of MCTS within the problem space that is Spelunky, whilst also identifying its weaknesses in comparison. With further experience and a better understanding of both the algorithm and AI, MCTS would quite comfortably fit inside of Spelunky as an AI Controller.

7.0 Recommendations and Improvements

Throughout the implementation of the forward-facing model into SpelunkBots, it was clear that the vanilla form of the algorithm had some limitations when introduced into the SpelunkBots API. Whilst causing issues with the implementation of the algorithm, there are several recommendations for further research that would help the performance of the algorithm within Spelunky.

7.1. GameState Modification

An important part of the MCTS algorithm is the GameState. Being able to offer a greater set of information for the algorithm to utilize would provide a more accurate simulation of the game space. Currently the GameState implementation is very simplistic, only relying on both the position of the Bot and its chosen ‘End Goal’. Further investigation into the vast range of variables provided by the SpelunkBots API into their use with the algorithm could provide a more accurate and powerful implementation of the algorithm.
7.2. Greedy MCTS

One recommendation I would make is an alternative form of MCTS known as ‘Greedy’ MCTS. Unlike the Vanilla algorithm, ‘Greedy’ MCTS utilizes heuristics and an algorithm known as Rapid Action Value Estimation (RAVE). (Gelly & Silver, 2011) The idea behind this extended MCTS algorithm is to replace the standard selection phase with an adapted RAVE algorithm, which provides a much faster estimation at the cost of accuracy. Combined with Heuristics, the algorithm offers a faster alternative to the vanilla form of MCTS which might be more beneficial in the dynamic search space of Spelunky.

7.3. Better Pruning

A problem that exists in multiple implementations of the MCTS algorithm as well as within this project is the pruning of nodes from the simulated tree. It is both a time consuming and expensive phase of the algorithm that can affect its performance, which is why it should proactively reduce as much as possible. A few starting points for improving the implementation are as follows:

- **Introduce Mixmax**: Mixmax is the idea of utilizing a constant value when calculating the total reward of nodes by using a ‘mix’ between the average value and maximum value between the children nodes. This was demonstrated in a MCTS Implementation into Super Mario Brothers by Jacobsen et al. (Jacobsen, et al., 2014) It proved to not only help offset the interpretation of riskier nodes against safer nodes, but also helped to increase the time a path could be calculated.

- **Macro Actions**: While in Spelunky we can slow down the speed a room is passed through to allow more computational time, the process of updating the bot’s action still is fast enough that it barely has time to search through the generated tree to find the best course of action. Utilizing Macro Actions has proven to help enhance search trees with a trade-off between precision and strategic quality. (Powley, et al., 2012) The incorporation of the algorithm into SpelunkBots should enable the Bot to handle the domain more efficiently whilst being able to perform more complex moves.
• **Partial Expansion:** Saving time searching the tree can help increase the performance of the algorithm dramatically, allowing more significant nodes to be explored while ignoring ones that aren’t beneficial to the bot (Chaslot, et al., 2007). This would be a great addition to the SpelunkBots MCTS implementation because of the complex problem space. Using Partial Expansion would help reduce the branching factor of the tree to help increase the performance of the algorithm, with minimal risk of missing promising paths through the reduction.

**7.4. Detailed Debugging Statistics**

To better analyse and improve the algorithm, one contribution that would be highly recommended is the introduction of more detailed statistics into the existing debug framework. Receiving information about the system effects of any iteration is sorely missed and currently must be committed outside the program. If such information could be included and recorded during the bot iterations, it would provide a much better look at the operation of the AI and would allow developers a closer inspection of their implemented bots without a debugger.

**7.5. Summary of Conclusion**

Throughout the testing of both the implementation of the MCTS algorithm into the SpelunkBots API and its ability to control a bot in Spelunky, it was never intended for the algorithm to be the perfect solution to controlling a SpelunkBot, but to assess the potential the algorithm has to offer with the game.

Overall the implementation can be considered a success, having provided a framework that is easy for future researchers both to understand and use as starting point for their own implementations, and the bot be able to utilize the algorithm’s provided data to control it. Whilst the current state of the bot is limited both by its’ very basic form and my own knowledge, with what has been provided researchers with an even greater knowledge of the algorithm will be able to make significant progress with research into the MCTS Algorithm, whilst also testing the limits of Spelunky with an algorithm that has shown the potential to be able to address the problems the game can challenge it with.
References


Available at: http://spelunky.wikia.com/wiki/Places/Classic

Thompson, T., 2015. “With Fate Guiding My Every Move” The Challenge of Spelunky,
Derby, UK: University of Derby.

Available at: http://spelunkbots.com/about-spelunkbots/
[Accessed 30 September 2015].

Available at: https://dl.dropboxusercontent.com/u/2353/learningagentguide.pdf
[Accessed 22 March 2016].
Bibliography


Appendices

Appendix A: MCTS Bot Implementation

Figure 0.1 MCTS Bot C++ Code

```cpp
void MCTSBot::Update()
{
    std::cout << "Simulating ..." << std::endl;
    gameState.SelectAction(_playerPositionXNode, _playerPositionYNode);
    std::cout << "Simulated" << std::endl;
    currentAction = gameState.action;
    std::cout << "Play: " << currentAction << std::endl;
    if (currentAction < 0)
    {
        currentAction = Standing;
    }
    if (currentAction = Standing)
    {
        currentAction = MoveRight;
    }
    std::cout << "Execute: " << currentAction << std::endl;
    if (currentAction = Standing)
    {
        _goRight = false;
        _goLeft = false;
        _headingLeft = false;
        _headingRight = false;
    }
    else if (currentAction = MoveRight)
    {
        _goRight = true;
        _headingLeft = false;
        _headingRight = true;
    }
    else if (currentAction = MoveLeft)
    {
        _goLeft = true;
        _headingLeft = true;
        _headingRight = false;
    }
    else if (currentAction = JumpRight)
    {
        _goRight = true;
        _jump = true;
        _headingLeft = false;
        _headingRight = true;
    }
    else if (currentAction = JumpLeft)
    {
        _goLeft = true;
        _jump = true;
        _headingLeft = true;
        _headingRight = false;
    }
}
```
void JordanBot::Update()
{
    _canGoRight = IsNodePassable(_playerPositionXNode + 1, _playerPositionYNode, NODE_COORDS);
    _canGoLeft = IsNodePassable(_playerPositionXNode - 1, _playerPositionYNode, NODE_COORDS);

    _canJumpRight = IsNodePassable(_playerPositionXNode + 1, _playerPositionYNode - 1, NODE_COORDS);
    _canJumpLeft = IsNodePassable(_playerPositionXNode - 1, _playerPositionYNode - 1, NODE_COORDS);

    _canJumpGrabRight = IsNodePassable(_playerPositionXNode + 1, _playerPositionYNode - 2, NODE_COORDS);
    _canJumpGrabLeft = IsNodePassable(_playerPositionXNode - 1, _playerPositionYNode - 2, NODE_COORDS);

    if (!_hasGoal)
    {
        for (unsigned nodeY = 0; nodeY < Y_NODES; nodeY++)
        {
            for (unsigned nodeX = 0; nodeX < X_NODES; nodeX++)
            {
                if (GetNodeState(nodeX, nodeY, NODE_COORDS) == spExit)
                {
                    _hasGoal = true;
                    _targetX = nodeX * PIXELS_IN_NODES;
                    _targetY = nodeY * PIXELS_IN_NODES;
                    CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
                    return;
                }
            }
        }

        if (_headingRight && (_canGoRight || _canJumpRight || _canJumpGrabRight))
        {
            if (!_canGoRight)
            {
                _jump = true;
            }

            _goRight = true;
            _headingRight = true;
            _headingLeft = false;
        }
        else if (_headingLeft && (_canGoLeft || _canJumpLeft || _canJumpGrabLeft))
        {
            if (!_canGoLeft)
            {
                _jump = true;
            }

            _goLeft = true;
            _headingLeft = true;
            _headingRight = false;
        }
        else if (_headingRight && (!_canGoRight && !_canJumpRight && !_canJumpGrabRight))
        {
            _goLeft = true;
        }
    }
}
_headingLeft = true;
_headingRight = false;
}
else
{
  _goRight = true;
  _headingRight = true;
  _headingLeft = false;
}
else
{
  if (_pathCount > 60)
  {
    _pathCount = 0;
    CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX,
    _targetY, PIXEL_COORDS);
  }
  _pathCount++;

  if (_playerPositionXNode < GetNextPathXPos(_playerPositionXNode,
  _playerPositionYNode, NODE_COORDS))
  {
    _goRight = true;
  }
  else
  {
    _goLeft = true;
  }

  if ((_playerPositionYNode - 1) > GetNextPathYPos(_playerPositionXNode,
  _playerPositionYNode, NODE_COORDS))
  {
    if (_goRight && (_canJumpRight || _canJumpGrabRight))
    {
      _jump = true;
    }
    else if (_canJumpRight || _canJumpGrabLeft)
    {
      _jump = true;
    }
  }
}
if (_headingRight)
{
  _attack = IsEnemyInNode(_playerPositionXNode + 1, _playerPositionYNode,
  NODE_COORDS);
}
else
{
  _attack = IsEnemyInNode(_playerPositionX - 1, _playerPositionYNode,
  NODE_COORDS);
}
if (_attack)
{
  _goLeft = false;
  _goRight = false;
}
void DiscoveryDan::Update()
{
    if (!_hasGoal)
    {
        for (int nodeX = 0; nodeX < X_NODES; nodeX += 1)
        {
            if (GetFogState(nodeX, _playerPositionYNode, NODE_COORDS) == 1)
            {
                _targetX = nodeX * PIXELS_IN_NODES;
                _targetY = _playerPositionY;
                _hasGoal = true;
                CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY,
                                      _targetX, _targetY, PIXEL_COORDS);
                return;
            }
        }
        for (int nodeX = 0; nodeX < X_NODES; nodeX += 1)
        {
            for (int nodeY = 0; nodeY < Y_NODES; nodeY += 1)
            {
                if (GetNodeState(nodeX, nodeY, NODE_COORDS) == spExit)
                {
                    _targetX = nodeX * PIXELS_IN_NODES;
                    _targetY = nodeY * PIXELS_IN_NODES;
                    _hasGoal = true;
                    CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY,
                                             _targetX, _targetY, PIXEL_COORDS);
                    return;
                }
            }
        }
    }
    else
    {
        if (_pathCount > GetPathCount())
        {
            _pathCount = 0;
            _hasGoal = false;
        }
        _pathCount += 1;
        if (_playerPositionXNode < GetNextPathXPos(_playerPositionXNode,
                                                    _playerPositionYNode, NODE_COORDS))
        {
            _goRight = true;
        }
        else
        {
            _goLeft = true;
        }
        if ((_playerPositionYNode - 1) > GetNextPosY(_playerPositionXNode,
                                                     _playerPositionYNode, NODE_COORDS))
        {
            _jump = true;
        }
    }
}
Figure 0.4 FishBot script

```cpp
void FishBot::Update()
{
    // Evaluate the surrounding blocks
    double block = GetNodeState(_playerPositionXNode + 1, _playerPositionYNode, NODE_COORDS);
    bool canMoveRight = (block == spEmptyNode);
    
    block = GetNodeState(_playerPositionXNode - 1, _playerPositionYNode, NODE_COORDS);
    bool canMoveLeft = (block == spEmptyNode);
    
    if (_headingRight)
    {
        _facing = 1;
    }
    else
    {
        _facing = -1;
    }
    
    bool canJump;
    bool canFall;
    canJump = (GetNodeState(_playerPositionXNode + _facing, _playerPositionYNode - 1, NODE_COORDS) == spEmptyNode);
    canFall = (GetNodeState(_playerPositionXNode + _facing, _playerPositionYNode + 1, NODE_COORDS) == spEmptyNode);
    
    // Detect creature in front
    for (int creature = spGhost; creature <= spSpiderHang; creature += 1)
    {
        _numberOfCreatures = NumberOfEnemyTypeInNode(creature, _playerPositionXNode + _facing, _playerPositionYNode, NODE_COORDS);
        int creatureType = creature;
        if (_numberOfCreatures > 0)
        {
            break;
        }
    }
    
    // Act on evaluations
    if (_isHanging)
    {
        _duck = true;
        _jump = true;
        _isHanging = false;
    }
    else if (_numberOfCreatures > 0)
    {
        _attack = true;
    }
    else if (_headingRight)
    {
        if (canFall && canMoveRight)
        {
            int heightCount;
            heightCount = 1;
            bool searchingForGround = true;
            
            do
            {
                heightCount += 1;
            }
```
searchingForGround = GetNodeState(_playerPositionXNode + _facing, _playerPositionYNode + heightCount, NODE_COORDS) == spEmptyNode;
} while (searchingForGround == false);

if (heightCount > 8)
{
   _duck = true;
   _ropep = true;
   std::cout << "place rope?" << std::endl;
   _goRight = true;
}
else if (heightCount > 4)
{
   _duck = true;
   _goRight = true;
   _isHanging = GetNodeState(_playerPositionXNode, _playerPositionYNode + 1, NODE_COORDS) == spEmptyNode;
}
else
{
   _goRight = true;
}
else if (canMoveRight)
{
   _goRight = true;
}
else if (canJump)
{
   _jump = true;
}
else
{
   _headingRight = false;
}
else
{
   if (canFall && canMoveLeft)
   {
      int heightCount = 1;
      bool searchingForGround = true;

      do
      {
         heightCount += 1;
         searchingForGround = (GetNodeState(_playerPositionXNode + _facing, _playerPositionYNode + heightCount, NODE_COORDS) == spEmptyNode);
      } while (searchingForGround == false);

      if (heightCount >= 8)
      {
         _duck = true;
         _ropep = true;
         std::cout << "place rope?" << std::endl;
      }
      else if (heightCount > 4)
      {
         _duck = true;
         _goLeft = true;
         _isHanging = GetNodeState(_playerPositionXNode, _playerPositionYNode + 1, NODE_COORDS) == spEmptyNode;
      }
      else
      {
         _goLeft = true;
      }
   }
} else if (canMoveLeft)
  {_goLeft = true;}
else if (canJump)
  {_jump = true;}
else
  {_headingRight = true;}
}
void GoldDigger::Update()
{
    if (!_hasGoal)
    {
        for (int nodeY = 0; nodeY < Y_NODES; nodeY += 1)
        {
            for (int nodeX = 0; nodeX < X_NODES; nodeX += 1)
            {
                if (NumberOfCollectableTypeInNode(spGoldBar, nodeX, nodeY, NODE_COORDS) > 0)
                {
                    _targetX = nodeX * PIXELS_IN_NODES;
                    _targetY = nodeY * PIXELS_IN_NODES;
                    _hasGoal = true;
                    CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
                    return;
                }
                else if (NumberOfCollectableTypeInNode(spGoldBars, nodeX, nodeY, NODE_COORDS) > 0)
                {
                    _targetX = nodeX * PIXELS_IN_NODES;
                    _targetY = nodeY * PIXELS_IN_NODES;
                    _hasGoal = true;
                    CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
                    return;
                }
            }
        }
    }
    else
    {
        if (_pathCount > GetPathCount())
        {
            _pathCount = 0;
            CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
        }
        _pathCount += 1;
        // Check if the item we're looking for is still there - or did we collect it?
        if (_hasGoal && NumberOfCollectableTypeInNode(spGoldBar, _targetX, _targetY, PIXEL_COORDS) == 0 &&
            NumberOfCollectableTypeInNode(spGoldBars, _targetX, _targetY, PIXEL_COORDS) == 0)
        {
            _hasGoal = false;
        }
        if (_playerPositionXNode < GetNextPathXPos(_playerPositionXNode, _playerPositionYNode, NODE_COORDS))
        {
            _goRight = true;
        }
        else
        {
            _goLeft = true;
        }
        if ((_playerPositionYNode - 1) > GetNextPathYPos(_playerPositionXNode, _playerPositionYNode, NODE_COORDS))
        {
            _jump = true;
        }
    }
}
Figure 0.6 Indie script

```cpp
void Indie::Update()
{
    if (_holdingItem)
    {
        _goRight = true;
        _lookUp = true;
    }
    else
    {
        if (!hasGoal)
        {
            for (int nodeX = 0; nodeX < X_NODES; nodeX += 1)
            {
                for (int nodeY = 0; nodeY < Y_NODES; nodeY += 1)
                {
                    if (NumberOfCollectableTypeInNode(spGoldBar, nodeX, nodeY, NODE_COORDS))
                    {
                        _targetX = nodeX * PIXELS_IN_NODES;
                        _targetY = nodeY * PIXELS_IN_NODES;
                        hasGoal = true;
                        itemGoal = false;
                        CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
                        return;
                    }
                }
            }
        }
        else
        {
            if (_pathCount > GetPathCount() && !itemGoal)
            {
                _pathCount = 0;
                _hasGoal = false;
                _itemGoal = false;
            }
            _pathCount += 1;
            if (_playerPositionXNode < GetNextPathXPos(_playerPositionXNode, _playerPositionYNode, NODE_COORDS))
            {
```

64 | Page  Appendix B: Botscripts
_goRight = true;
}
else{
    _goLeft = true;
}

// Jump if below the nearest y point.
if ((playerPositionYNode - 1) > GetNextPathYPos(playerPositionXNode, playerPositionYNode, NODE_COORDS)) {
    _jump = true;
}

if (playerPositionX < targetX + 16 && playerPositionX > targetX - 16 && playerPositionY < targetY + 16 && playerPositionY > targetY - 16) {
    if (itemGoal) {
        _duck = true;
        _attack = true;
        _holdingItem = true;
    }
}
}
Figure 0.7 NotSoSolidSnakeBot script

```cpp
void NotSoSolidSnake::Update()
{
    if (!_hasGoal)
    {
        for (int nodeX = 0; nodeX < X_NODES; nodeX += 1)
        {
            for (int nodeY = 0; nodeY < Y_NODES; nodeY += 1)
            {
                if (NumberOfEnemyTypeInNode(spSnake, nodeX, nodeY,
PIXEL_COORDS))
                {
                    _targetX = nodeX * PIXELS_IN_NODES;
                    _targetY = (nodeY - 1) * PIXELS_IN_NODES;
                    _hasGoal = true;
                    CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
                    return;
                }
            }
        }
    }
    else
    {
        if (_pathCount > GetPathCount())
        {
            _pathCount = 0;
            _hasGoal = false;
        }
        _pathCount += 1;
        // go towards the x point of the closest node on the path
        if (_playerPositionXNode < GetNextPathXPos(_playerPositionXNode, _playerPositionYNode, NODE_COORDS))
        {
            _goRight = true;
        }
        else
        {
            _goLeft = true;
        }
        // Jump if below the nearest y point.
        if (_playerPositionYNode > GetNextPathYPos(_playerPositionXNode, _playerPositionYNode, NODE_COORDS))
        {
            _jump = true;
        }
    }
}
```
void SeanBean::Update()
{
    if (!_hasGoal)
    {
        // Search for the spike!
        for (int nodeY = 0; nodeY < Y_NODES; nodeY += 1)
        {
            for (int nodeX = 0; nodeX < X_NODES; nodeX += 1)
            {
                if (GetNodeState(nodeX, nodeY, NODE_COORDS) == spSpike)
                {
                    _hasGoal = true;
                    _targetX = nodeX * PIXELS_IN_NODES;
                    _targetY = nodeY * PIXELS_IN_NODES;
                    CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
                    std::cout << "FOUND: X: " << (_targetX / 16) << " Y: " << (_targetY / 16) << std::endl;
                    return;
                }
            }
        }
    }
    else
    {
        if (_pathCount > GetPathCount())
        {
            _pathCount = 0;
            CalculatePathFromXYtoXY(_playerPositionX, _playerPositionY, _targetX, _targetY, PIXEL_COORDS);
            _pathCount += 1;
        }
        _playerPositionXNode < GetNextPathXPos(_playerPositionXNode, _playerPositionYNode, NODE_COORDS)
        {
            _goRight = true;
        }
        else
        {
            _goLeft = true;
        }
        _playerPositionYNode - 1 > GetNextPathYPos(_playerPositionXNode, _playerPositionYNode, NODE_COORDS)
        {
            _jump = true;
        }
    }
}
Appendix C: How to run the MCTS Bot in SpelunkBots

The Spelunkbots project is also included in this project, this includes the GameMaker project, Spelunkbots DLL and Source, and GameMaker 8.1 to run the project.

In order to run the Spelunkbots project and the Bot AI, open Game_Maker.exe in the “GameMaker_8” folder, and click yes when it prompts to enter Advanced Mode. Finally, click file and open, and select the file titled “Spelunkbots.gmk” in the folder Source\Spelunky_1_1.

Inside the GameMaker application, Navigate to the ‘Scripts’ Folder in the left content browser window, and go inside the ‘AIToolSet’ folder, the ‘BOTSCRIPTS’ folder and open the ‘PlayerChoice’ script.

Figure 0.9 PlayerChoice Location

In order to change which bot plays, you will need to change the value found in the following expression on line 18:

‘Bot = 8.’

In order to play the MCTS Bot, set the value to ‘1’.

In order to play the JordanBot, set the value to ‘8’.

To run the various test levels as mentioned in Chapter 5, press ‘F3’ on the main menu screen and enter the relevant level names to play-through them. Likewise, to see how well the Bots perform inside the Game, allow the Bot to enter the ‘Start’ door as shown on the main menu.