University of Derby

School of Computing & Mathematics

A project completed as part of the requirements for the

BSc. (Hons) Computer Games Programming

entitled

Implementation of High-level Strategy Formulating AI in Ms. Pac-Man

Luc Shelton

lucshelton@gmail.com

2012-2013

Abstract

The game of Ms. Pac-Man was first released in 1981. It has since been recognised by academics as an interesting test-bed for developing AI that is capable of playing against a stochastic opponent. As such, numerous journals have been published to the World Congress of Computational Intelligence in Games demonstrating various methods of producing competitive AI. Within our paper we aim to determine whether an agent with a hard-coded strategy applied with heuristic simulations, is capable of producing a high-scoring AI than those previously demonstrated.

Contents

University of Derby 1
School of Computing & Mathematics 1
A project completed as part of the requirements for the 1
BSc. (Hons) Computer Games Programming 1
entitled 1
Abstract 2
Acknowledgements 6
1. Introduction 9
1.1 Aims & Objectives 10
2. Literature Review 11
2.1 Methods of agent implementation 11
2.1.1 Screen Capturing 11
2.1.2 Simulator 12
2.2 Monte Carlo Tree-Search 14
2.2.1 Multi-armed bandit problem 15
2.2.3 Ms. Pac-Man MCTS implementations 17
2.2.4 Endgame approaches 19
2.2.5 Other approaches 20
2.3 Decision weighting and Finite State Machines 22
2.4 Ghost Avoidance and Detection 25
2.4.1 Pincer moves 25
3. Implementation 27
4. Agent Design 28
4.1 Tree Search Implementation (MCTS) 29
4.1.1 Structure 29
4.1.2 Usage 29
4.1.3 Direction Selection 30

4.2 Considerations 31
4.2.1 Simulation Cycles 31
4.2.2 Tree Depth 32
4.3 Finite States 33
4.3.1 (Default) Wander 33
4.3.2 Flee 34
4.3.3 Ambush 34
4.3.4 Hunt 35
4.3.5 End Game 35
4.4 Development 36
4.5 Modifications 37
4.5.1 External File Management 37
4.5.2 Cloning Game States 37
4.5.3 Capturing screen buffer and saving to images 38
4.5.4 Logging information 39
4.5.5 Simulator 40
5. Research & Analysis 42
5.1 Setup 42
5.1.1 Monte-Carlo Tree Search Parameters 42
5.1.2 Finite State Machine Parameters 43
5.1.3 Testing Machine 44
5.1.4 Controller 45
5.1.5 Ghost Behaviour 48
5.2 Data Collected and results 49
5.2.1 Folder Structure 49
5.2.2 Main Implementation 51
5.2.3 Scripted behaviour and Finite State Machine 59
5.2.4 Pure MCTS approach 61

5.3 Analysis and Critical Evaluation 65
5.3.1 Finite State Machine and Scripted Behaviour 65
5.3.2 Pure MCTS Approach 66
5.3.3 Main Implementation 68
6.1 Conclusion 71
6.1.1 AI Performance 71
7. Future Work 76
7.1 Agent Improvements 76
7.1.1 Application of MCTS 76
7.1.2 Counter-acting Pincer Moves 78
7.2 Re-evaluating our tools 79
7.3 Usage of MCTS with other games 79
8. Bibliography & References 81
9. Appendices 83

Table of Figures

Figure 1 - A screenshot of the game on the first level.....9
Figure 2 - The simulation of the play out and the selection of a child node. (Pepels and Winands, 2012).....14
Figure 3 - Branching decision points within the maze used for MCTS evaluation. The blue circle represents the position of Ms. Pac-Man. (Ikehata and Ito, 2011).....15
Figure 4 – The UCB1 formula used for evaluating tree nodes as demonstrated by (Auer et al., 2002).....16
Figure 5 – The UCB-tuned algorithm as demonstrated by (Samothrakis et al., 2011).....17
Figure 6 - Danger map of the first maze within Ms. Pac-Man (Ikehata and Ito, 2011).....18
Figure 7 - Ms. Pacman can't find a suitable path to take as the paths are returning with a 0 reward due to lack of adjacent pills.....19
Figure 8 - The abstract method stub that is provided to us through the simulators API.....27
Figure 9 - The direction that has to be returned to the simulator at each tick.....27
Figure 10 - Simple function for returning the next possible move when traversing irregular C-Paths.....30
Figure 11 – Random-No-Inverse: The logic used for the simulation cycles in the tree.....32
Figure 12 - Diagram displaying the finite state machine layout of our agent.....36
Figure 13 - An example image that is captured from the simulator upon arriving at a junction.....38
Figure 14 - The TestStats object that is serialized when the testing is complete.....40
Figure 15 - The appearance of the simulator during runtime. The debug console (left) and the visual interface of the game state (right).....42
Figure 16 - Scoring output from the scripted behaviour agent.....65
Figure 17 - Pills eaten from the scripted behaviour agent.....65
Figure 18 - Scoring output for our Pure MCTS agent.....66
Figure 19 - The pill consumption scoring from our Pure MCTS agent.....67
Figure 20 - Scoring output for our Main Implementation.....68
Figure 21 - Pills eaten by our Main Implementation.....69
Figure 22 - Chart displaying the total survival time of all our agents and their respective configurations.....71
Figure 23 - Graph demonstrating the total sum of ghosts that were consumed by each agent.71
Figure 24 - Pills eaten out of all our agent implementations.....72

Figure 25 - Ms. Pac-Man being caught out by a pincer move in our Configuration 2 of Main Implementation 78

Acknowledgements

I would like to take a moment to thank dissertation supervisor Dr. Tommy Thompson for his patience, and providing me with highly valued guidance as I embarked upon this challenging project.

In addition, I would like to acknowledge the loving support of my peers on the degree as well as that of my mother, Françoise's kind words during tough times when I needed it most.

Lastly, I would like to extend my thanks to Tim Leonard, Matt Lowe, Hassan Ghoncheh and Darren O'Connor for giving me critical feedback and consequently enabling me to improve my work at every step of the way.

1. Introduction

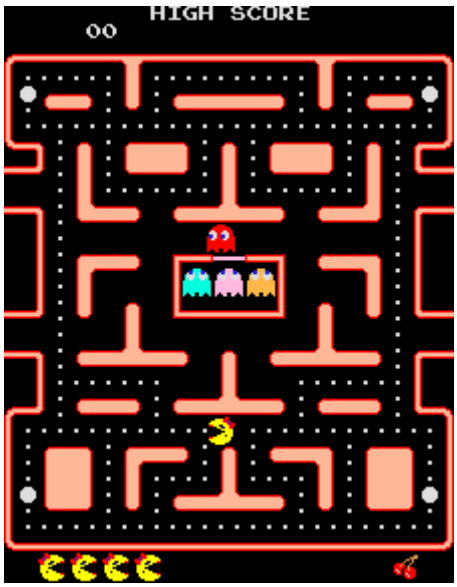

Figure 1 - A screenshot of the game on the first level

The game of *Pac-Man* was first released in arcades during the 1980s and featured Pac-Man himself, along with 4 enemy ghosts called; Blinky, Pinky, Blue and Clyde. The objective of the game was simple in that the player had to avoid (or consume) the 4 ghosts whilst also collecting all the pills within the maze. Should the player have consumed a power-pill found inside the maze, the ghosts would become edible for a short period of time. Upon consumption of a ghost the player would score a larger bonus than they would from a normal pill and the ghost would then return to the maze in an immortal state. Once the player consumed all the pills within the maze, the level would change layout and the game would then become progressively harder, increasing the speed of the ghosts and decreasing the time in which the ghost were edible (Galván-López et al., 2010). Shortly after the release of *Pac-Man*, *Ms. Pac-Man* was released as a sequel to the game and contained identical gameplay mechanics however the enemy ghosts portrayed distinct behavioural differences from the previous version. The main difference was that the ghosts within the first game were deterministic and for the most part considered predictable (Pittman, 2011). Instead, in the game of Ms Pac-Man not only was Clyde renamed to Sue, but the artificial intelligence (AI) behind the ghosts became unpredictable and for the most part demonstrated stochastic (random) behaviour (Galván-López et al., 2010).

The stochastic nature of the ghosts soon became an interesting problem space for academics when the first publications emerged demonstrating this was by (Koza, 1992). They detailed that Ms. Pac-Man was one of the possible test beds when it came to applying genetic programming within games. Since then, the conference on *Computer Intelligence of Games* (CIG) has been conducting a competition to apply various methods of agent behaviour and implementation to determine which outputs the most optimal score.

Recent papers on the topic of developing high-scoring agents for Ms. Pac-Man mention the usage of a new method of predicting future game states based on recursive random simulations using a tree-based structure such as the Monte-Carlo Tree Search algorithm. One of the first academic journals presenting the idea was (Robles and Lucas, 2009) , in which

they demonstrate a simple tree search structure for determining the reward from future game states at each junction within the maze. From the results of their research they demonstrate positive results and a good premise for extending such an idea. Furthermore we have begun to see promising results from the likes of rule-based fixed-strategy agents such as ICE Pambush 3 (Thawonmas and Ashida, 2010), of which won the CIG Ms. Pac-Man competition in 2009.

In this paper, we aim to determine whether it is considered possible to achieve a high scoring agent (AI) when applying the best-first search heuristics of the Monte-Carlo Tree Search (MCTS) algorithm along with fixed state-based strategies. Considering the research that has been conducted previously regarding the two respective areas of Ms. Pac-Man AI, we feel that by developing a finely tuned hybrid can produce competitive results in comparison to the other entrants of the same competition. As of yet, we have yet to see such an agent that makes use of a heuristic search method while implementing fixed hand-coded strategies. We believe that by directing the usage of such methods by applying certain tactics means that we can create an optimal agent within the game of Ms. Pac-Man.

1.1 Aims & Objectives

After reviewing an array of relevant academic and non-academic articles, we have come to decide that the aims of our research are;

- Apply an optimized form of a heuristic based best-first tree search and combine the return values with hand-coded rules (Finite State Machine).
- Conclude whether the utilisation of hand-coded conditions through the means of a finite state machine is preferable to high-level heuristic decision making.
- Determine if the computational cost of implementing such an algorithm can be reduced, enabling for more competent AI within games of high branching moves.
- Produce a high scoring agent within the game of Ms. Pac-Man
    - The agent should be capable of at least proceeding onto the next level.
    - The agent should also, on average, produce better results than other rule-based look-ahead agents that have entered the IEEE CIG conference in previous years (refer to Literature Review).

2. Literature Review

In recent years, academics have been coming together annually for the competition known as the Ms. Pac-Man AI Competition held at the IEEE Computer Intelligence and Games Conference (CIG). The competition organised and moderated by *Philipp Rohlfshagen*, *David Robles* and *Simon Lucas* from the *University of Essex* involves contestants to designing AI controllers to play the classic arcade game of Ms. Pacman. Consequently due to the unpredictable nature of the enemy ghosts, it has proven to be a challenging test-bed for agent behaviour, and as such has become the basis of the competition.

The goal of the competition is develop a high-score agent using any AI methodologies necessary (Robles and Lucas, 2009). Most noticeably there have been attempts to use methods such as neural networks (Gallagher and Ledwich, 2007) with temporal difference learning (De Bonet, 2006), fuzzy systems (Handa and Isozaki, 2008) and population based incremental learning (Gallagher and Ryan, 2003). This has meant that numerous publications have been made in regards to how to approach the random behaviour of the opponent and when to prioritise the rewards within the game to gain the highest possible score.

Within our literature review we aim to demonstrate the current research in regards to agents that use finite state machines and event driven behaviour, as well as heuristic based best-first search algorithms such as Monte-Carlo Tree Search. Additionally we aim to determine the most effective methods of applying these kinds of behaviour, and what has been proven as successful in previous conferences and academic journals.

2.1 Methods of agent implementation

To date, there have been various methods of agent implementation that use certain methods of game simulation for the purpose of executing AI behaviour. At the IEEE CIG 2009 conference, it was displayed that most agents utilized the likes of screen-scraping interfaces for enabling agent controllers to behave. In addition academics have used simulators to gain further control over the game to better determine how agent behaviour alternates when certain factors are removed from the game.

2.1.1 Screen Capturing

Utilising the original *Microsoft Ms. Pacman Revenge of the Arcade* game software, screen capturing involves the intercepting the graphics buffer of the computer's screen. Afterwards it is then analysed remotely by another piece of software controlling the AI controller for the player. A grid based layout is typically generated from the image that is taken from the buffer,

and then further used to determine the next moves by the artificial controller (Fitzgerald and Congdon, 2009). The reason for this is so that the game state can be discretised by the software so that the agent controller can observe a set of perfect information for the given state.

(Robles and Lucas, 2009) discuss this with their agent implementation and determine that on average there is an 80 millisecond delay between image processing and game state recognition. A timeframe that, considering the movement speed at which Pacman is going is quite considerable as the agent requires time to compute a decision. The agent in this case is a separate process operating on top of the original software which means that it's running in a completely different thread. This means that providing that the screen capture adapter is accurate or even fast enough, it could cause for the agent to miss junctions within the levels maze environment (Tong and Sung, 2010). Within the game of Ms. Pac-Man we consider a junction to be a point in the maze in which Ms. Pac-Man is able to go in more than or equal to 3 possible directions.

(Fitzgerald and Congdon, 2009) refactored the original code by Simon Lucas in an effort to reduce this latency. This was achieved by ignoring certain features of the maze such as walls during the processing of the graphics buffer. Considering that their implementation simply requires that the controller return the intended direction, it's more efficient to discretise the game as a graph of nodes. They do go on to mention that there are some limitations with the usage of screen capturing. For example; should a ghost be hovering on top of a pill within the game state, then during the period in which the image is parsed, it will be close to impossible to recognise that the ghost is on a pill. This is troublesome, however it is possible to prevent this by storing information from previous screen captures and aggregating them into a single set of data (Fitzgerald and Congdon, 2009). Pursuing this option does mean that agent is able to obtain the most accurate score within the game; however it loses valuable time that could otherwise be used for expensive algorithm calls on the CPU required to determine the decisions of the AI agent.

2.1.2 Simulator

The usage of a simulator enables for larger control of how AI agents can be implemented, however it causes there to be larger inaccuracies in how the actual game is represented as well. Notably (Fitzgerald and Congdon, 2009) mention that using a simulator simplifies the task somewhat as it offers the AI agent direct access to the game state include the behaviour of the enemy ghosts. This is concluded through the demonstration of using the *Monte Carlo*

Tree Search (MCTS) algorithm as referred to in section 2.2, due to the probabilities required in determining the behaviour of the ghosts and additionally if we were to refer back to the problems noted in the previous section. If the AI agent has direct access to the values that are being used to determine the ghost's next moves, it means that it will be able to generate a much more accurate simulation and measurement of the danger of adjacent ghosts.

On the other hand, using a simulator in this regard can enable for further analysis of certain agent states, considering we will be able to get a data set for the game state that is recognised to be perfect information. Through this we can gain a further understanding as to how the agent performs when interacting solely with a configurable amount of ghosts within the game environment (Gallagher and Ryan, 2003).

The usage of a simulator-based test bed additionally causes lack of accuracy from the scoring output considering that the rules can be modified in accordance to however the authors feel that it's best. For instance, the speed at which Ms. Pac-Man travels through the maze when energised on a power-pill, and for how long the player remains energised for can be modified freely causing for variations on the game. It was noted by (Samothrakis et al., 2011) that the speed in which Ms. Pac-Man traverses was altered slightly to match that of the ghosts in the maze, and no life was given at 10,000 points. In essence, it simply means that it would be hard to compare the results generated from our own implementation directly with similar agents due to the variation in environments.

From a performance perspective however, utilising a simulator means that it doesn't suffer from the same problems that screen scraping would, in that turnings at intersections within the maze could never be missed by the functionality of the controller. However, considering that the agent would be operating on the same thread as the simulator, it would then mean that instead the MCTS algorithm would block the updating and rendering of the game due to its CPU expensive nature.

2.2 Monte Carlo Tree-Search

"Monte Carlo Tree Search MCTS is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results."

(Browne et al., 2012)

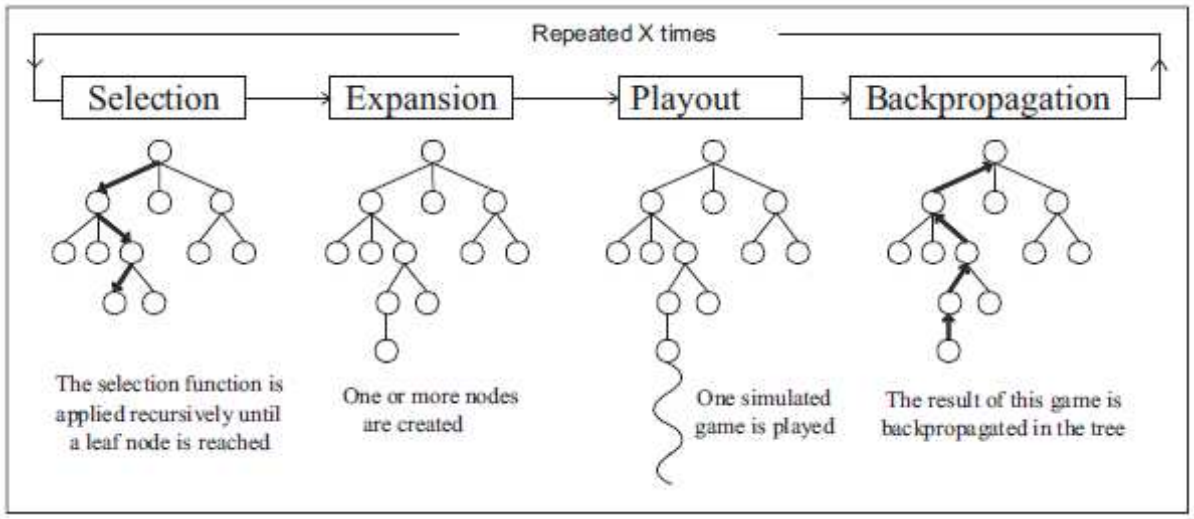

Figure 2 - The simulation of the play out and the selection of a child node. (Pepels and Winands, 2012)

Originating from a collection of methods noted as the "Monte-Carlo methods", of which were typically used in statistical physics, they have also began to be used within the likes of games too. Demonstrating world-class levels of play in the likes of Scrabble and Bridge (Browne et al., 2012), the Monte-Carlo methods rely on repeated random sampling to compute the result of a given game state. Deriving from this idea, the Monte-Carlo Tree Search (MCTS) algorithm operates under the idea of two main concepts in that the true value of an action (a node within the tree) can be approximated by using random simulation recursively. The *X* amount of times in which the simulations will recursively cycle can vary on the implementation. Based on previous publications, it can be concluded that the simulation of the generated tree will finish based by a computational budget (typically memory or time constraint) or a maximum iteration constraint (Browne et al., 2012). Considering the large range in power between modern CPUs, it would be preferable to adjust the simulation time based on a computational budget. This form of constraint is more flexible, as the controller will determine how long it has taken to generate the MCTS search tree. From there, the controller will determine whether the computer is capable of doing more simulations within the initial provided time constraint (Browne et al., 2012). It is stated by (Samothrakis et al., 2011) that it takes approximately half a second (500 milliseconds) to perform 400

simulations, however (Pepels and Winands, 2012) demonstrate that it's possible to produce an even higher amount within less than 100 milliseconds.

The generated tree is then used to estimate future game states at different parts of the nodes, whilst continuing on the game state from the parent node in the tree (Browne et al., 2012). To date, this algorithm has set a precedent in enabling computers to compete with players within games that have high branching decision making factors. Examples of this are demonstrated by games such as *Go* and *Scrabble*, of which have imperfect information, instances where information is partially visible to the agent that is generated a decision.

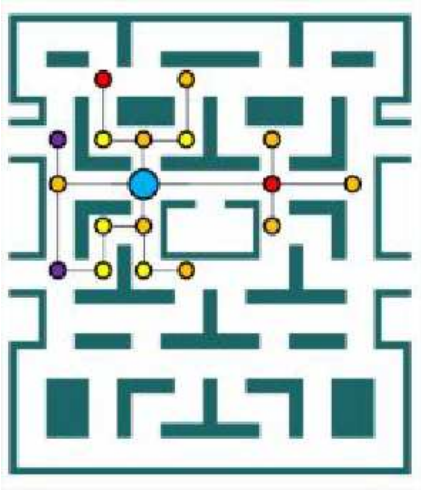

Figure 3 - Branching decision points within the maze used for MCTS evaluation. The blue circle represents the position of Ms. Pac-Man. (Ikehata and Ito, 2011)

Similarly we can see this being applied to Ms. Pac-Man considering the previously mentioned stochastic behaviour of the ghosts in the game. The random simulation and reward evaluation enables for us to better determine what would be the optimal path to take when the agent arrives at a junction within the maze. We consider each junction within the maze to be a branching point for decision evaluation with the tree of the MCTS algorithm as demonstrated in Fig. 3. Through this algorithm, the Ms. Pac-Man agent is able to simulate future game states without being required to determine the moves of the ghosts based on its previous actions. Instead, through random simulation and an appropriate evaluation of a generated tree, we can determine the safest route based on the current position of the agent. The evaluation however of which generated branch of the tree to take can vary based on the goal that we are trying to aim for and furthermore the structure of the tree that we are using.

2.2.1 Multi-armed bandit problem

The MCTS algorithm has been combined with the usage of bandit-based methods to best evaluate the optimal set of actions to use within a game based on the generated tree. Considered as the multi-armed bandit problem, the MCTS algorithm is presented as having k set of arms in possible decisions that can be made by the controller (Browne et al., 2012).

As demonstrated by (Auer et al., 2002), the UCB1 (Upper Confidence Bound) formula can be applied to a node within a tree to generate a random score that is then recursively back-propagated from the leaf of the tree (i.e. a node with no children) to its predecessor. This

score is then utilised to determine the reward of the branch, which within context to Ms. Pac-Man would be considered as a path within the maze.

$$UCB(i) = \begin{cases} n & (T_i = 0) \\ X_i + C\sqrt{\frac{\ln T}{T_i}} & (T_i > 0) \end{cases}$$

Figure 4 – The UCB1 formula used for evaluating tree nodes as demonstrated by (Auer et al., 2002)

Referring to Fig. 4, *T* is considered the visit or "sample count" in which determines how many samples are to be generated before the node is considered as "exhausted". Once a node is considered as exhausted, it then means that it is ready to be expanded and then extend the tree at that given point. The log natural is then applied on the right part of the algorithm to enable the exploration of less visited nodes within the tree's structure during the evaluation period of the tree (Pepels and Winands, 2012).

This formula is used within bandit-based sequential decision-making problems in which the choice between exploitation moves we know already to be profitable, versus unexplored moves has to be balanced appropriately when a decision has to be made across *k* certain amount of nodes. This is typically called the exploitation and exploration dilemma (Browne et al., 2012). In context to the method of our implementation, the sample count is used to determine whether or not we choose to expand the children of the node that we are currently looking at.

(Samothrakis et al., 2011) mention within with their research that using the UCB-Tuned formula (Auer et al., 2002) for evaluating nodes displays optimal results with their chosen structure of the MCTS algorithm. They indicate that by applying the UCB-tuned formula with a min-max based MCTS tree, a structure that is preferable to games with perfect information, that it displays results which were preferable in comparison to the previously introduced UCB1. The results provided in (Samothrakis et al., 2011) demonstrate that when utilising the UCB-tuned formula, the game score consistently tends to be higher when the depth of the search tree is higher and the simulation threshold (i.e. amount of times a child node is simulated) is anywhere between 200 and 350. Whereas in comparison to when the UCB1 node evaluation formula is used, it's recognised that there is a large inconsistency in performance when both simulation and tree depth is higher. Consequently it would be worthwhile running our experiments with either formula to determine if they affect score

outputs when combined with the idea of utilising state-based strategies mentioned in section 2.6.

Refer to Fig. 5 to understand the algorithm in question.

$$\bar{x}_j + \sqrt{\frac{\ln(n)}{n_j} \min\left\{1/4, \bar{x}_j^2 - \bar{x}_j^{\,2} + \sqrt{\frac{2\ln(n)}{n_j}}\right\}}.$$

Figure 5 – The UCB-tuned algorithm as demonstrated by (Samothrakis et al., 2011)

Within the diagram; *n* is the sample size of parent node, *Xi* is the sample score of the child node of the one that we are looking at within the tree, *nj* is the sample size (or visit count) of the parent node the one that we are observing within the tree.

2.2.3 Ms. Pac-Man MCTS implementations

Rather than observing the current game state the agent is operating within (i.e. rule based look-ahead), MCTS approaches the problem of determining optimal future game states. This is achieved by simulating them based on a set of probabilities used to approximate ghost behaviour during the game (Ikehata and Ito, 2011). From this we can then determine the rewarding output for the newly simulated game state.

This is demonstrated in (Ikehata and Ito, 2011) by simplifying the rules of movement that are used for the ghosts. A level of probability is used for determining the aggressiveness of each of the ghosts within the game to determining their actions for each simulation. They note also that due to the expensive nature of the heuristics, they calculate path simulations at each intersection or turn within the maze environment. This is done instead of computing at each grid space on the tunnels between intersections as it would otherwise be too computationally expensive.

Applying a more static and strategic approach, (Ikehata and Ito, 2011) utilize certain tactics to determine how their UCT (Upper Confidence Tree) generated from algorithm would be best suited to avoidance or eating of ghosts (Ikehata and Ito, 2011). There is, however, no mention of implementing an ambush strategy for the ghosts, a strategy that consists of luring the ghosts adjacent to the pill so that when the pill is consumed the agent can quickly eat the enemy ghosts.

This is a strategy that is utilized by the likes of *ICE Pambush 3* (Thawonmas and Ashida, 2010), a rule-based agent that was compared against Ikehata and Ito's MCTS-driven approach (Ikehata and Ito, 2011). Notably, their agent approaches avoidance by being wary of pincer moves from the enemy ghost. A move in which the Ms. Pac-Man agent is unable to find a path to escape with as the team of ghosts have surrounded the agent.

The MCTS agent demonstrated by (Ikehata and Ito, 2011) displays a method of restricting tree growth based on a variable rather than a maximum path, cost which is something that (Pepels and Winands, 2012) choose to implement in their MCTS-based agent. Restricting the growth of the tree based on the path cost of each branch could be considered worthwhile considering moves are only decided at junctions in the maze. Should a ghost decide to change their direction randomly (of which they are perfectly capable of doing) midway through a Ms. Pac-Man's traversal of a long C-Path, then it could invalidate their previous choice to move down that path. Based on that notion this then could be considered understandable due to how the agent in (Pepels and Winands, 2012) will not reverse on itself midway of the traversal through a c-path.

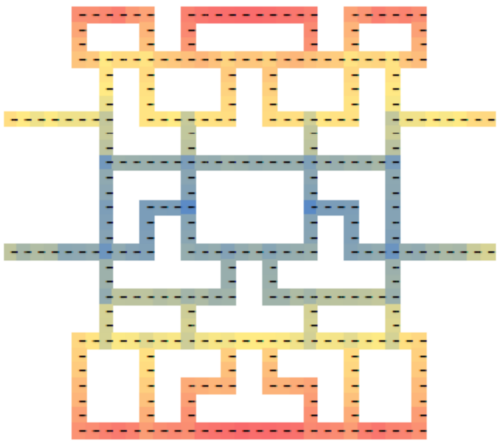

Figure 6 - Danger map of the first maze within Ms. Pac-Man (Ikehata and Ito, 2011)

Furthermore, during the evaluation of the nodes within their agents MCTS search tree, (Ikehata and Ito, 2011) comment on the idea of using a danger map to influence the reward values of tree branches. Referring to Fig. 4, the areas highlighted in red are considered as more dangerous due to higher probability in which a pincer move would occur as there are less possible routes out of the areas highlighted in red. This is a valid idea considering that while the ghosts may adhere to stochastic behaviour, the possibility for them to form a pincer move is still there. Therefore prioritising other c-paths within the maze before approaching the ones that are considered more dangerous right at the beginning would be an applicable move. However, should we desire to use this method of node evaluation with the likes of an *Ambush* strategy stated in section 2.6, it would mean that there'd be a higher chance that we neglect power pills in the maze due to them being positioned within the 4 corners of the maze.

While they approach the usage of a heuristics algorithm such as MCTS from a similar angle that we intend, we still feel they are still are missing a few key areas and applying the algorithm in circumstances that we don't believe are necessary. For instance, the same heuristic algorithm doesn't appear to be used when they are searching for the remaining pills within the maze. If this were the case with our implementation, then we would have to consider extending the expansion threshold so that the tree could reach the next set of pills.

2.2.4 Endgame approaches

The endgame can be defined as the state in the game in which the Ms. Pac-Man agent must eat the fewest remaining pills within the level. This alternates from the beginning of the game considering that the agent would have the alternative choice of simply eating all of the power pills. Upon eating the remaining pills, the maze will change onto the next level (Tong et al., 2011).

Figure 7 - Ms. Pacman can't find a suitable path to take as the paths are returning with a 0 reward due to lack of adjacent pills.

Notably MCTS works very well in evaluating and determining the most appropriate set of low level actions to take (i.e. direction in which the agent should go in) within the area of the tree-search threshold. However, the search tree begins to fail when the agent fails to find any pills within the branches of the search tree due the agent beginning to progress towards the end of the game and there being a smaller count of pills remaining. Demonstrated in in Fig. 3, the simulated game states at each branch return a value of 0 due to there being no increase in score when the controller was simulated at such a given point in the tree.

Another tactic or state is required to target pills that might be outside of the area of the agent. Rather than expanding the branch generation threshold for the heuristic algorithm. (Tong et al., 2011) approached this through simply generating the shortest linear path to the remaining pills in the game. Their implementation utilises pre-computed path costs through a simple path algorithm, a path that does not go back on it itself or visit any path nodes twice. From this they are able to directly determine the nearest node that contains pills and then determine if the target location for the agent is safe enough by evaluating the generated path frequently through Monte-Carlo path testing.

This is a valid and less computationally expensive approach than simply expanding the threshold of tree simulations. The reason for this is that it enables Pac-Man traverse to the other size of the maze without having to perform expensive heuristics to find few remaining pills within the remaining environment. This limitation of a pure MCTS approach is again demonstrated by (Samothrakis et al., 2011), who demonstrate that in able to ensure that a calculation can be performed within a 50-60ms time frame, a tree depth value must be enforced. Providing the agent with a linear path to remaining cells in the maze then becomes an appropriate idea given the limited computation time preventing us from extending the tree expansion further.

Taking this all into account, we believe that with the implementation of the MCTS algorithm we would have to take two conditions into account during the runtime of the game. The first being the amount of pills available within the level (if there are only X amount left, change state), and the second being the scoring output that is being returned from the search tree. (Pepels and Winands, 2012) additionally state within their agent implementation that a viable condition for changing to an End Game tactic would be by responding to a condition regarding the time that has passed within the game. While we think that this is an interesting idea, it would be better applied if we timed the period in which Ms. Pac-Man consistently received a maximum score of 0 from the children of the search tree and then switching to the *End Game* state. Moreover, should the agent become astray from the nearest pills in the maze then a distance condition should be met that enforces the *End Game* strategy also.

2.2.5 Other approaches

Rather than determining the outcome of future game states, other methods have been approached by academics such as the idea of using evolution strategies. The premise of an evolutionary algorithm is that the agent will inductively learn about the surrounding environment. From this information it will make appropriate decisions based on trial and error. The information that has been learned about the given game state is then applied through fixed parameter based strategies (Galván-López et al., 2010) or weighted neural networks. (Gallagher and Ryan, 2003) demonstrate the idea of implementing a method of evolutionary learning and base their agents actions similar to that of a human player. Through this, their research consists of developing a controller that learns inductively to play Ms. Pac-Man.

While a choice such as this might be preferable for short-term moves within a certain area, considering the volatility in move choices by the ghosts within the game, it could be

considered impractical due to the stochastic nature of the AI. (Gallagher and Ryan, 2003) comment on the idea of generating a long sequence of moves being a bad idea considering the volatility of the game state and the general behaviour of the ghosts. Although inadvertent, this statement could be in contrast to the idea of implementing a more heuristics method of using Monte-Carlo Tree Search considering the stochastic nature.

(Gallagher and Ledwich, 2007) implement a neural network (ANN) with a form of an evolution strategy to optimise the way that the agent responds to the maze. The four outputs of the ANN are the four possible directions that the Ms. Pac-Man agent can move within the maze, where the inputs to the network is surrounding information to the current position of Ms. Pac-Man within the game. They simulate a population of agent for several weeks to acquire behaviour that is considered as competitive against other referenced agents utilising Pentium 4 CPUs of which they note to be one of the reasons as to why the generation times are slow.

Tuning parameters such based on evolutionary simulations in the future could be a viable idea for ensuring that the constraints and conditions that we have in place for changing states are leaner and responsive to the behaviour of certain the ghosts in the maze. What is meant by this is simply that although all the ghosts in the game make use of stochastic behaviour at random intervals, they still utilise their own behavioural patterns. This meaning that the agent would be more capable of responding to certain ghosts and changing states when appropriate.

(Galván-López et al., 2010) demonstrate a more dynamic approach to the usage of rules by generating them through grammatical evolution. They relate the work that they conducted to be similar to that of (Szita and Lõrincz, 2007), in that the strategies that are applied to their agent are generated through an evolutionary algorithm. Through using a simple and readable *if* <*condition*> *then* <*action*> statement, they could generate complex set of rules that the agent would abide by. They conclude within their results that their evolutionary based method achieved a higher score in comparison to a hand-coded approach, which within the context of our research would be a finite state machine.

2.3 Decision weighting and Finite State Machines

A *Finite State Machine* (FSM) is a set of states that determine the behavioural actions of an AI agent. States may transition between each other if a condition is met, otherwise the operation of the state will repeat continuously. The FSM dictates agent behaviours and decisions at a given time-point based on its current state.

(Thompson et al., 2008)

Recent works by (Thompson et al., 2008) demonstrate promising results through the method of controlling their agent through pure state-based strategies. The research demonstrates that they apply their agent's logic through a simulator that only replicates certain features to the original Ms. Pac-Man software. They combine the Finite-State Machine with underlying strategies to alter the behaviour of the agent. Through the usage of their finite state machine, they apply 3 separate strategies for determining the most optimal direction to take once they arrive at a junction. The first being a total count of all available pills from all possible directions that can be taken at the junction that Ms. Pac-Man is at and then heading in that direction.

This approach becomes rather short-sighted however, as its method of avoidance is purely reactive to the adjacency of ghosts. Notably the greedy look-ahead strategy that is described by (Thompson et al., 2008) for finding the nearest tunnel where pills are available is something worth considering when combining with an MCTS implementation.

As stated in 2.3, the MCTS can fall short of functionality should there be no pill within the reach of search tree. Extending said search tree comes at the cost of the CPU and slow decision making times meaning that to fill the gap by having a state machine that is responsive in producing a A* path to the nearest pill node could be a viable option. Furthermore we can consider that the MCTS algorithm could be used as a means of determining the safety of the Ms. Pac-Man agent during the traversal to the nearest pill node (Tong and Sung, 2010). Should the oncoming C-Path be considered dangerous based on the returned results of the MCTS tree then a re-plan of the path could be done to ensure safety.

Additionally, (Fitzgerald and Congdon, 2009) utilize a rule based approach when targeting their agent towards a Java version of the Ms. Pac-Man game. Although heavily modified from Simon Lucas' original Java-based software, their agent receives an average high score of 10,364 by applying a set of hand-coded rules and parameters. Similar to that of a typical FSM, they apply conditions from a range of vocabulary that the agent must respond to. These

conditions in turn are determined by parameters that are implemented by hand. Although their agent is purely reactive to the information that is provided to it (i.e. responding to each captured image separately with no previous history of the maze), they enhance the chosen rules by applying an evolutionary algorithm.

It was noted that the intention during the development of the agent to implement an evolutionary component to producing dynamic conditions and properties. It was never developed before the CIG 2009 competition thus leaving the question open as to whether an evolutionary algorithm can produce a more comprehensive rule set and ultimately better performance. We believe that an approach like this could be considered flawed given the non-deterministic nature of the ghosts. For instance, if the parameters are adjusted in accordance to the agents interaction with the ghosts (adjacency etc.), and the behaviour rules of each respective ghost is non-deterministic then then the information that is gained is going to be consistently thrown off.

(Gallagher and Ryan, 2003) demonstrate a rule-based agent that places emphasis on generating parameters using evolutionary algorithms that appropriately respond to dangerous situations within the environment. Their approach begins with using a two-state finite state machine of which consists of explore and retreat. While the strategy of the agent is very much static, the parameters that are applied are tuned appropriately based on what is occurring within the game state.

Another example of a successful rule-based approach to solving the problem was demonstrated at the 2009 CIG IEEE conference by the *Ice Pambush 3* agent (Thawonmas and Ashida, 2010). Although the screen-capture interface that was used had been optimised, there were noticeable gains in the way that the agent performed through the logic that was implemented. It's worth mentioning that the applied rule set to the agent is not increased at all, instead the parameters are finely tuned based on the information that is provided from the state space.

This information then determines the radius in which the Pac-Man agent should be concerned about such as the proximity of the ghosts. When the agent then meets the condition for wanting to head for an adjacent pill, the *Depth-First Search* (DFS) path planning algorithm is applied to determine the most optimal route to get there.

Applying an evolution strategy, similar to what (Fitzgerald and Congdon, 2009) intended to implement, (Thawonmas and Ashida, 2010) were able to optimize the distance and cost

parameters with their agent. These are values that are used for determining the appropriate distance between the ghost and the agent as well as how effective certain paths around the maze are to achieving the highest possible reward. Through their rule-based system however, they additionally apply a fixed short-term strategy to ensuring that the agent can "ambush" the ghosts when they are adjacent. When the appropriate rule condition is met, the agent will remain in relative position so that it does not stray from a certain radius of the power pill. When the ghost is within distance then the agent will pursue the power pill in question. I believe that applying a strategy such as this as well as utilising some form of game state awareness (MCTS in this instance), could ultimately mean that we have an agent that has a strategy in mind whilst having accurate enough knowledge to avoid ghosts down certain C-paths within the maze.

(Szita and Lõrincz, 2007) follow a similar pattern in their work. However, rather than modify the parameter values such as what would be considered dangerous distance to the agent for example, they generate a list of the rules based on a collection of "action modules" and observations. The results presented after the experiments of (Szita and Lõrincz, 2007)'s work demonstrated that the usage of the Cross-Entropy-Method would be beneficial for generating these policies for targeting multiple goals at the same time. Furthermore, their method is capable of determining the priorities within decision lists (a collection of rules), therefore should more than one rule have their condition met at the same time then the one with the preferable priority would have its action executed. Such an idea of prioritisation could be considered worthwhile of pursuing for enhancing a static FSM approach. For instance, if Ms. Pac-Man were within imminent danger of being consumed by a ghost but was nearing a power pill, the task of acquiring the power pill within the maze would be prioritised. This would be so that it would prevent the Ms. Pac-Man agent changing direction right before consumption.

It's still worth noting however that the previously mentioned approaches are negligent of the direction that the ghosts are coming from, they instead determine solely the adjacency and use this value for determining whether the ghosts are a danger. On the contrary, the MCTS algorithm could be utilised to evaluate the likelihood of survival when simulating the game state at a junction within the maze. The reason we believe this would be preferable is because by determining the proximity of the ghost to the agent itself doesn't provide any additional information as to which direction the ghost will go in next. We consider this problematic

seeing as the ghosts within the environment operate within a stochastic nature (Thompson et al., 2008).

2.4 Ghost Avoidance and Detection

Within the original version of the Pac-Man game, the behavioural patterns of the ghosts are already known, as in to say that they are deterministic. In contrast, ghosts within the original Ms. Pac-Man game operate in a non-deterministic nature (Thompson et al., 2008). However it is considered that there are some circumstances in which the ghosts will target Ms. Pac-Man in such a way that it will consequently block every potential path of escape for the agent. (Thompson et al., 2008) apply a somewhat simple method of avoidance against the ghost by determining adjacency measured in terms of the Manhattan distance (Krause, 1987). Similar method is applied but in a more dynamic sense by (Szita and Lõrincz, 2007) where they generate priority based rule sets using actions such as "*NearestEdGhost" or "NearestGhost"* which returns the distance in which the Ms. Pac-Man agent should respond to. These rules in turn are determined by applying the Cross-Entropy Method.

(Tong and Sung, 2010) approach this problem through utilizing the MCTS algorithm and applying a shallow search within the immediate state space of the Ms. Pac-Man controller. A shallow-depth search in this context is when the depth of the MCTS search-tree is shortened to represent intermediate game states rather than ending ones (i.e. game states that would occur just after a turn at a junction). The method is applied due to idea that if the Ms. Pac-Man agent was considered to be in a dangerous state then the ghosts would be nearby and therefore would be detected by simulations carried out within the branches of the MCTS tree due to less thorough nature of the generation.

2.4.1 Pincer moves

Considering that the behavioural patterns of the ghosts within the environment are random in nature, there are some instances where there can be the possibility of one or two ghosts teaming together to target the Ms. Pacman agent within a corner. This is referred to as a "pincer" move and in essence means that every path that Ms. Pacman would otherwise have to escape is then blocked causing for the agent to be forced to lose (Pepels and Winands, 2012).

As such, it would mean that during runtime of the games simulation, we would have to take into consideration the positioning of the ghost AI so that we can tell early on if there was some kind of ambush being formed. (Pepels and Winands, 2012) discuss within their

implementation of MCTS for both ghost and Ms. Pacman agents, that by applying a LGR policy (Last-Good-Reply) (Baier and Drake, Dec.), it is possible to tell early on when a pincer move is occurring for the Ms. Pacman agent.

(Baier and Drake, Dec.) describes the MCTS algorithm as an inductive machine learning method in that in that results of the previous play out, in which this context could be considered a junction within the Ms. Pac-Man maze, affect the moves of the future game states. While this might not be considered entirely true within the context of Ms. Pac-Man considering ghosts behave partially within a stochastic nature, there is some relevance.

Through this, (Baier and Drake, Dec.) state that each "reply" (i.e. a good move made in the game), are stored within the predecessor during the back-propagation process of the MCTS generation when playing a game of *Go*. (Pepels and Winands, 2012) conclude within the results of their research that applying such a method doesn't provide any significant performance enhancement when it comes to effectively avoiding a team of ghosts. This would conclude that ultimately using something such as Last-Good-Reply alone would be futile in determining a pincer move and would only add more computation time onto an already expensive algorithm. Rather than recording moves, (Tong and Sung, 2010) use an influence map to avoid parts of the maze that would otherwise be considered dangerous due to the high possibility of a pincer move forming. As mentioned in 2.4.3, the influence map places weighting against the scoring of the search tree branches of MCTS based on the parts of the maze that the tree expands to. This makes it less likely that the agent will head towards the direction of a corner of the maze unless it has to due to the applied penalty of going within that area.

Through the provided literature we have demonstrated that there has been positive progress made through the research of using either of our proposed AI methods. Moreover, we have also seen varying methods of application which has aided us to refine our approach to implementing our agent. While we recognize the MCTS algorithm to be CPU intensive, we see also that there are varying useable techniques that enable us to reduce the effects of this. We see also that through previous competition entries that there are various fixed hand-coded strategies that have been consistently proven to be successful such as the Ice Pambush 3 agent (Thawonmas and Ashida, 2010). Through our agent design we will demonstrate the varying approaches that were taken to enable both the MCTS algorithm and the finite state layout to be competitive.

3. Implementation

Referring back to the *Computer Intelligence and Games conference* mentioned in section 2 of our literature review, there are two ways that are regarded as the main methods of implementing an AI agent into the likes of Ms. Pacman gameplay. The first method is through the usage of simulator and the second through a screen-scraping interface observing the original game software.

It was taken into consideration that the method of obtaining graphics buffer information from the original software would apply additional latency to our agent's computation. The reason for this is because of the time required to analyse the image of the game and extract the conditional information that we would require from it (locations of Pacman, pills, ghosts etc.).

Furthermore it was considered that we would have spent more time optimising the efficiency of object recognition and ensuring that the game state was accurately represented when information was missing from the captured images. We believe instead that our efforts would be best placed in ensuring that the agent's behaviour is optimal and in accordance to our agent design specification as stated in section 4.

Our simulator at each tick will request for our agent to return a move that it wants to go in and will provide our agent with the current *GameState.*

Refer to the abstract method definition below.

```
        /// <summary>
        /// Called at every tick
        /// </summary>
        /// <param name="gs">A copy of the game state instance</param>
        /// <returns>Returns the direction that we want to go in</returns>
        public override Direction Think(GameState gs)
```

Figure 8 - The abstract method stub that is provided to us through the simulators API.

The direction value that is returned is an enumerator that consists of the following values.

```
public enum Direction { Up = 0, Down = 1, Left = 2, Right = 3, None = 4,
Stall = 5 };
```

Figure 9 - The direction that has to be returned to the simulator at each tick.

- **Stall** – Informs the simulator to keep the agent stationary.

- **None** – Informs the simulator to keep moving in the same direction that Ms. Pac-Man was moving in previously.

The other values in the enumerator type are considered to be self-explanatory.

4. Agent Design

By applying a finite state machine structure to the implementation of our agent, it provides us with a strategic approach that can provide context for a heuristic best-first search algorithm such as MCTS. Given the context in which the algorithm is used can be defined by the finite state that the agent is in during the time of usage. For example, the algorithm could be used for ghost avoidance or for simply navigating the maze based on the highest reward output of each branch within the tree.

Furthermore, we intend for the MCTS simulations to be used solely within the *Wander* state as it will be able to determine the best c-paths to traverse through the reward output of the simulations. The *Wander* state will change states should any of the pre-defined parameters have their condition met. In this instance, should the agent be within a certain distance of a ghost (based on the parameters we provide), then the state will change. We measure the distance between each entity within the game based on a metric called the Manhattan distance.

The "Manhattan distance" is a form of measurement that derives from the idea of taxi-cabs that move from one end of Manhattan Island to the other through the city blocks. Which route that is considered the shortest can vary or be considered equal but at the same time appear as completely different (Krause, 1987). We can apply this method of measuring distance in the context of the game of Ms. Pac-Man due to junctions and turns within the maze.

Due to the nature of the simulator's current version, it's worthwhile noting that rather than producing decisions at each intersection, Ms. Pac-Man will have to determine a low-level decision (i.e. a direction) at every tick of the simulators runtime. The available outputs of our agent controller is the four possible directions (up, down, left, right) along with none (carry on moving with the previously selected direction), or stall. Stalling informs the simulator to repeatedly inverse the direction of the agent so that Ms. Pac-Man remains in relatively the same position. This is particular output is used with the likes of our *Ambush* strategy. Refer to Fig. 7 for the values that the agent must return at each tick.

4.1 Tree Search Implementation (MCTS)

4.1.1 Structure

We intend for the structure of our MCTS implementation to be similar to that of a Min-Max tree. As stated by (Browne et al., 2012), a min-max tree is typically used within an a perfect environment (i.e. an environment that is completely observable). Given the context of the game of Ms. Pac-Man, the entire state space is viewable by the player, so in this regards we would consider it to be the same for the AI agent as well.

4.1.2 Usage

We aim to make use of the MCTS algorithm at each junction that the agent arrives at in the maze. A junction is defined as a point within the maze in which the agent has 3 or more possible directions that it is able to move in. It is only at these points in which the agent will evaluate all the possible directions and determine which direction to go on the next tick. This consequently means that during L-shaped paths (junctions with only 2 possible moves) within the maze the agent is still going to have to move constantly. A problem arises from this due to the fact that the agent will not be called upon to choose a direction when heading through this type of tunnel. This is fixed simply by applying the function displayed within Fig. 10.

```
        // Attempt to go within the provided direction. If it's not
possible, then
        // return the next nearest direction.
        private Direction TryGoDirection(GameState gs, Direction
pDirection)
        {
            var _directions = gs.Pacman.PossibleDirections();

            // Determine whether or not we are able to go in that direction
            if (_directions.Contains(pDirection))
            {
                return pDirection;
            }
            else
            {
                // Just return the first that is not the inverse of the
direction
                // that we are aiming to go in
                foreach (var dir in _directions)
                {
                    if (GameState.InverseDirection(dir) != pDirection)
                    {
                        return dir;
                    }
                }
            }

            return Direction.None;
        }
```

Figure 10 - Simple function for returning the next possible move when traversing irregular C-Paths

It is noted that during the preliminary part of our tests, we aim to make use of the algorithm only within the *Wander* state of the agent so that the decisions made within that state are determined by the amount of pills that are consumed more than anything.

4.1.3 Direction Selection

Stated previously, the agent's navigation around the maze will mostly be influenced by the MCTS search algorithm and the UCB scores that are generated by it. This is because I believe that through the simulation stages of the generation of the tree, it will determine the best possible score through certain c-paths whilst taking into consideration the ghost through adjacent routes.

Through the research stages, we alternated between several different methods of tree node selection as stated by (Browne et al., 2012), once the appropriate scores are generated by the UCB evaluation.

- **Max child:** Select the most visited root child (higher sample size)

- **Max-robust child:** Select the root child with both the highest visit count and the highest reward. If none exist, then continue searching until an acceptable visit count is achieved.
- **Secure child:** Select the child of which maximises the lower confidence bound.

For the evaluation of the tree nodes within the search tree, we aim to use the formulas stated within section 2.1 of our literature review which are UCB1 and UCB-Tuned.

4.2 Considerations

Having reviewed the literature in section 2.1, we are lead to believe that there are a few things that we should be concerned in regards to our how we approach implementing our agent.

4.2.1 Simulation Cycles

When a node within a generated tree is evaluated for its reward, the game state at the time of evaluation is simulated by how many is defined as a parameter. The agent during this period will not behave by the same rules and conditions as our actual implementation, rather it will move randomly for the given amount of cycles based on

Based on the ideas mentioned within the agent implementation of (Thompson et al., 2008), I concluded that it would be appropriate to base the agent that we would use for our simulations two separate strategies.

- **Greedy-Random**
    - Upon arrive at a junction, the Ms. Pac-Man agent will determine which direction contains the most pills and chose it as the next choice within the tick.
- **Random-No-Inverse**
    - A random move is selected, but it cannot be the inverse of the current direction that Pac-Man is going in.

```
        public override Direction Think(GameState gs)
        {
            List<Direction> possible = gs.Pacman.PossibleDirections();
            if (possible.Count > 0)
            {
                int select = GameState.Random.Next(0, possible.Count);
                if (possible[select] !=
gs.Pacman.InverseDirection(gs.Pacman.Direction))
                    return possible[select];
            }
            return Direction.None;
        }
```

Figure 11 – Random-No-Inverse: The logic used for the simulation cycles in the tree

4.2.2 Tree Depth

There are two main ways of measuring the depth of the tree.

4.2.2.1 Layer Depth

The layer depth within the search tree is considered as the recursive level that the tree can expand to. For instance, when the root expands in 4 possible directions that is considered as one layer, and should one of the children from that expansion gets chosen to expand again, that would be constitute as a layer as well.

4.2.2.2 Child Depth

The child depth is considered to be the total amount of children that can be generated within the tree. This accounts for every branch within the tree that is not considered a leaf (i.e. no children from a given node).

4.3 Finite States

With my implementation of the state machine, upon transitioning between states, one set of actions will be called once before entering a loop in regards to the state. Following a similar FSM structure demonstrated by (Thompson et al., 2008), I aim to implement specific strategies that are responsive to certain scenarios. For example when a power pill is consumed by our agent or when ghosts are within close proximity of our agent. Furthermore, I almost hope to put in place a method of ambushing the enemy ghosts within the environment.

Due to the promising results that were demonstrated within their result we concluded that it would be appropriate to follow a similar structure with a few additional enhancements. Refer to *Figure 12* to observe a visual representation of the states that are to be used in our agent.

4.3.1 (Default) Wander

Being the first state that is executed during when the controller is launched for the first time, Ms. Pacman will apply the MCTS algorithm to generate the tree at each junction of the maze. As mentioned previously, based on the selection parameter that is chosen through testing (Max Child etc.) will determine how the next direction at the maze junction is selected.

For each set of tests, we will be replicating the same conditions but changing the parameters to see what the outcome is of these tests. Stated in section 2.3.4, we concluded that it would be appropriate for our AI to respond to three conditions that would determine whether or not the agent would change its currently active state to that of another.

4.3.1.1 Conditions

- **Endgame**
    - If the remaining pill threshold within the maze has been met.
    - If the highest scoring child within the tree is 0. This means that there are no adjacent pills nearby that would otherwise raise the score of the child node to something above 0.
- **Flee**
    - If the nearest ghost is 2 Manhattan distance (Krause, 1987) value away from the agent, change to the fleeing state.
- **Ambush**
    - If a power pill is considered close enough to the Pac-man agent based on a fixed constant, then change to the Ambush state.

4.3.2 Flee

If the Ms. Pac-Man agent is within a close proximity of a ghost and the MCTS hasn't provided a better alternative route during the Wandering state, then this state is considered as a failsafe. The agent will recognise the direction that the ghost is coming towards the agent and will attempt to go the opposite direction if possible. It is considered dangerous when the nearest ghost is within configurable nodes distance (FLEE_THRESHOLD) of the agent.

This state is implemented due to the way that that the MCTS algorithm is applied within our agent. Our agent will not change direction midway through traversal of a C-Path in the maze which leaves the agent vulnerable to g hosts that would perhaps otherwise make moves that are different to those that were simulated. This is entirely possible based on the stochastic nature of the ghosts mentioned in section 2.1. The *Flee* state then becomes an intermediary that guides the agent away from the dangerous scenario before returning to *Wandering* again.

During this state, upon arriving at a junction within the maze, it will evaluate all possible directions and determine whether there is a ghost in that direction. Should there be a ghost in all provided directions, then the agent will determine the furthest ghost and then head in that direction in off-chance that there will be another junction.

Should the distance of the nearest ghost to Ms. Pac-Man be farther than that of the parameter FLEE_CHANGE_THRESHOLD, then the agent will change state back to *Wander*.

4.3.2.1 Conditions

- **Wander**
    - The agent will return back to the wandering state when it is considered safe again. It is considered safe when the agent is at least 2 nodes distance values away from the nearest ghost.

4.3.3 Ambush

The functionality of this state is considered to be rather simple and is only transitioned to during Wander should the condition be met that there is a power pill within a certain fixed radius.

Within this state, the agent will repeatedly change directions back and forth between its current direction and the one opposite to that to ensure that it remains in relatively the same position. Due to the way in which the game works, Ms. Pac-Man has to be moving at all times

unless the agent is in a corner which means that the controller has to interact with the game in such a way to ensure that the agent doesn't progress any further around the maze.

This will continue until the condition is met determining that the enemy ghosts within the maze are within a pre-defined constant radius of the agent (AMBUSH_DISTANCE_THRESHOLD). Should the ghost be within the radius that is defined in the controller, then the agent will go for the power pill and then transition to the *Hunt* state. We've applied this method based on the success that (Thawonmas and Ashida, 2010) had with their implementation of the Ice Pambush 3 agent.

4.3.3.1 Conditions

- **Hunt**
    - Once the agent has consumed the power pill, change to *Hunt* so that the mortal ghosts can be pursued when the agent is immortal.

4.3.4 Hunt

Upon consuming the power pill, the agent will immediately target adjacent ghosts. Upon selecting the nearest ghost based on the Manhattan distance, a Dijkstra-based path will be generated and a sequence of low-level moves (directions) will be provided to the agent to follow. However, should the agent fail to find adjacent ghost that are within approachable distance during the energise time (4 seconds – although changeable) then the agent will return to the "Wander" state.

The value that will determine the adjacency of the ghosts will be determined through the experimentation phases of the agent.

4.3.4.1 Conditions

- **Wander**
    - If the ghost that the agent is targeting comes out of being mortal (i.e. inedible), then the agent will return back to the *Wander* state.

4.3.5 End Game

This state is activated when there are no immediately adjacent nodes to the Ms. Pac-Man agent within the maze or the MCTS search tree is returning a best score of 0. The purpose of this state is to generate the shortest available path to the nearest node that contains a pill. Once we are near enough, we return back to the *Wander* state where the MCTS algorithm will be close enough to the remaining pills to determine the best scoring route for the agent. This is

similar to the strategy that was applied as a part of the work carried out by (Thompson et al., 2008) in which they state it as being *Greedy-Lookahead*.

4.3.5.1 Conditions

- **Wander**
    - If we're considered to be near enough a pill, it would be safe to assume that the MCTS search depth is within range of the remaining pills within the environment.
- **Flee**
    - The agent will resort to fleeing if at any point it considers itself to be in danger with any of the ghosts in the maze.

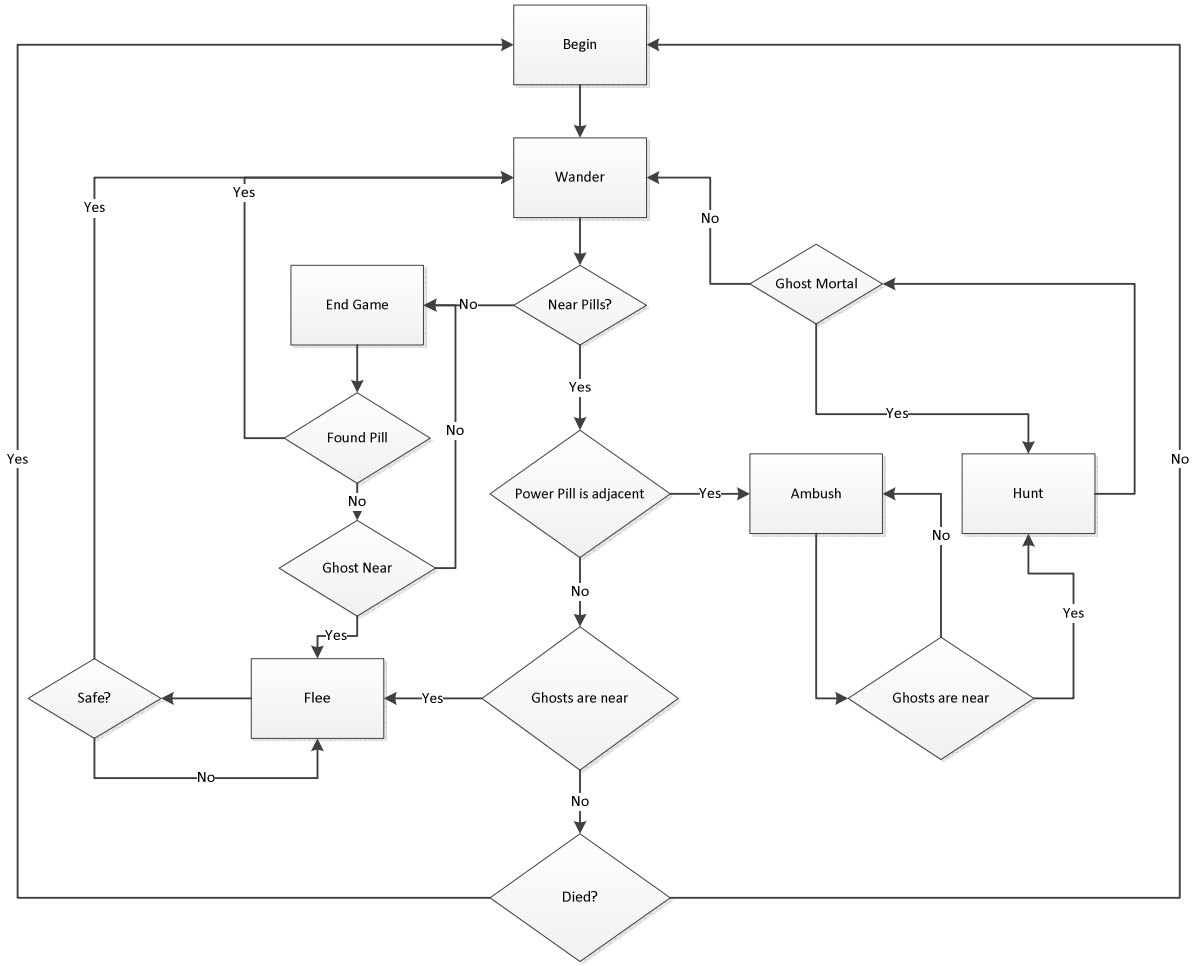

Figure 12 - Diagram displaying the finite state machine layout of our agent.

4.4 Development

For the implementation of our AI agent, I decided to utilize the *IEEE CIG* suggested C# based simulator developed by (Flensbank and Yannakakis, 2008). My reason for choosing this is to

do with our familiarity and understanding of the programming language and knowing how to exploit it to implement the AI controller better. Utilising *Visual Studio 2008*, the solution consists of the simulator, the screen capture adapter (that utilises the original Microsoft Ms. Pacman "Revenge of the Arcade" Software) and the library project that is loaded into it that will contain the implementation of my Ms. Pac-man agent. This separate DLL (dynamic link library) containing the implementations code is then loaded into the simulator during runtime using the .NET *System.Reflection* namespace. It is noted that the simulators choice of rendering the game state to the player through the means of WinForms and GDI+ is not entirely ideal based on the fact that it utilizes the CPU for the majority of its rendering processing (Microsoft, 2012). However, we believe it should remain sufficient enough in our endeavours for the agent that we intend to develop.

4.5 Modifications

Our agent required several pre-requisites to be met before being developed, some of which were not available with the original version of the simulator. Extensive modifications had to be carried out on the engine and structure of the software before-hand so that we could proceed.

4.5.1 External File Management

Minor modifications have already been made to the simulator environment, including problems that were found in which the assembly (.dll) containing the data regarding our agent implementation was unable to load. As it turned out, this was partly to do with the way that the simulator environment was managing external files to operate with which in turn caused it to crash frequently.

4.5.2 Cloning Game States

The fundamental part of the Monte-Carlo Tree Search algorithm is for its ability to simulate potential game states in a discretized state space. In order to achieve this, it would mean to replicate the information that is used for processing the gameplay at runtime. The original source code that was provided for the simulator appeared to have the appropriate method stub in place for cloning but unfortunately there was no functional code that would enable us to do such a thing.

We discovered that through utilising C# there were various methods of cloning object data at runtime through the means of manually generating new objects and copying information

between the current game state and the replica or simply parsing the object through *Reflection*.

Although using the means of *Reflection* saved us a substantial amount of development time, there was still a significant delay when the cloning process began during runtime of approximately 5 seconds or more. Considering that the game state has to be cloned numerous times during the recursive tree generation process, this was simply unfeasible. As such, we discovered that although it would be somewhat error-prone, it was better to implement our own manual method of cloning. We carried this out by modifying the main objects within the simulator by making sure that the implemented the *IClonable* interface. This interface was put into place with all relevant entities in the game state included Ms. Pac-Man, the four ghosts, map information and the game state itself.

Through this method, there was a decrease in the time that it took to clone certain games when using certain parameters for the MCTS generation. Upon removing irrelevant map data from the cloning process, such as levels that were not immediately relevant to the simulation procedure, we were furthermore able to decrease the time required to clone.

Furthermore we encountered an issue in the way that the information regarding each node within the game state was being copied. This is to do with how the *Node* class is laid out within our simulator. Within each *Node* object, there is a refer to the node that is

4.5.3 Capturing screen buffer and saving to images

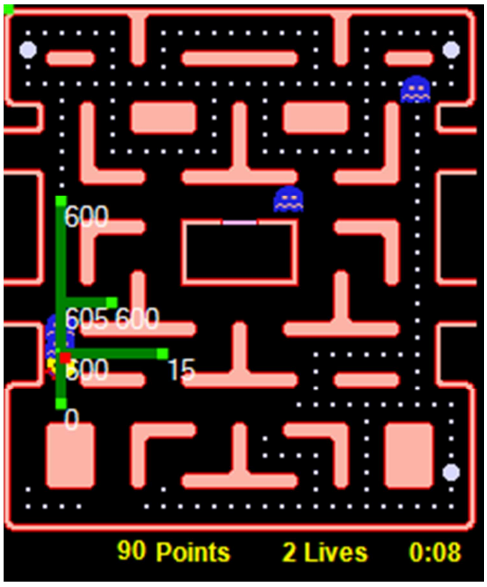

Figure 13 - An example image that is captured from the simulator upon arriving at a junction.

As an invaluable tool for debugging and determining the effectiveness of the recursive tree algorithm, I implemented a function within the simulator that enabled me to automatically take screen shots every time the controller arrived at a junction and generated a new search tree.

From this, we can ensure appropriate scores are being generated during the runtime of the simulations within the environment. An example of this would be when Pacman arrives at a junction and a ghost would be to the immediate right. I would expect the branch to the right of the generated tree to have a severely penalised

score after evaluation because taking that route would be suicide for the agent (e.g. a value that is a lot lower than 0).

Due to the structure of the simulator, it also meant that we could generate a visual representation of game states that we were not necessarily able to see such as the ones that were simulated by the Monte-Carlo Tree Search algorithm. This meant that we could accurately determine that conditions were being met when the tree generation process was occurring.

Lastly, it also meant that we could identify the ways in which our agent would lose the game. For instance, as stated within section 2.4.1, pincer moves are entirely possible within the game. Therefore as a means of counter-acting this kind of behaviour from the ghosts, we must identify first in what state the game was in when the agent lost.

4.5.4 Logging information

For better debugging capabilities, we integrated a logging system that would append new messages onto a text file that correlated to the session that we were debugging within. Saved with a date timestamp, it afforded us flexibility in finding out values of game states that we couldn't necessarily see. For instance, with the generation of the MCTS tree, there are several simulations of a game state going on that are now visible to us as the user. The logging functionality enabled us to output the information that was appearing.

To enable for our logging output to be efficiently recorded without much effort on our part to store and reuse, we made use of the Newtonsoft JSON libraries developed by (James, n.d.) for serializing our *TestStats* object that stores all the information regarding the test. By serializing it within this format, it means that afterwards we are able to read the text file into an external tool and deserialized it back into an object within code at runtime. This saves us time having to prepare a proprietary file format that would be used by the simulator. Instead, we can store the information in a widely recognized format and do so without costing us much development time.

Refer to the code below to overview the *TestStats* class.

```
public class TestStats : JsonSerializer
{
    public int MinLevelsCleared = 0;
    public int MaxLevelsCleared = 0;
    public int AverageLevelsCleared = 0;
    public int TotalLevelsCleared = 0;

    public string SessionID = "";

    public long ElapsedMillisecondsTotal = 0;

    public int TotalGames = 0;

    // The amount that each of the ghost kills the Pac-Man agent.
    public int RedKills = 0;
    public int PinkKills = 0;
    public int BlueKills = 0;
    public int BrownKills = 0;

    public int TotalPillsTaken = 0;
    public int MaxPillsTaken = 0;
    public int MinPillsTaken = int.MaxValue;
    public int AveragePillsTaken = 0;

    public int TotalGhostsEaten = 0;
    public int MaxGhostsEaten = 0;
    public int MinGhostsEaten = int.MaxValue;
    public int AverageGhostsEaten = 0;

    // Used for recording how long each game round takes.
    public float LongestRoundTime = 0;
    public float ShortestRoundTime = float.MaxValue;
    public float AverageRoundTime = 0;
    public float TotalRoundTime = 0;

    public float MinLifeTime = float.MaxValue;
    public float MaxLifeTime = 0;
    public float AverageLifeTime = 0;
    public float TotalLifeTime = 0;
    public int TotalLives = 0;

    public int MCTSTotalGenerations = 0;
    public int MCTSMaximum = 0;
    public int MCTSMinimum = int.MaxValue;
    public int MCTSAverage = 0;
    public int MCTSTotalTime = 0;

    public int TotalScore = 0;
    public int AverageScore = 0;
    public int MinScore = int.MaxValue;
    public int MaxScore = 0;

    public void Reset()
    {

    }
}
```

Figure 14 - The TestStats object that is serialized when the testing is complete.

4.5.5 Simulator

Other discrete modifications were also made to the way that the simulator launched such as argument handling so that we had greater flexibility in the way that we were able to simulate

games with our controller without having to modify code directly within our instance of *Visual Studio 2008.*

List of available arguments

- **-c**
    - How many games do we wish to simulate.
- **-g**
    - Ghosts that we want available in the simulation. Referring back to the testing methods that (Gallagher and Ryan, 2003) demonstrate, I considered that it might be preferable to implement something like this for debugging and performance analysis purposes.
- **-q**
    - Prevent the agent from generating any log output to the console screen.
- **-a**
    - Name of the agent that we want loaded from the aforementioned dynamic linked library.

5. Research & Analysis

5.1 Setup

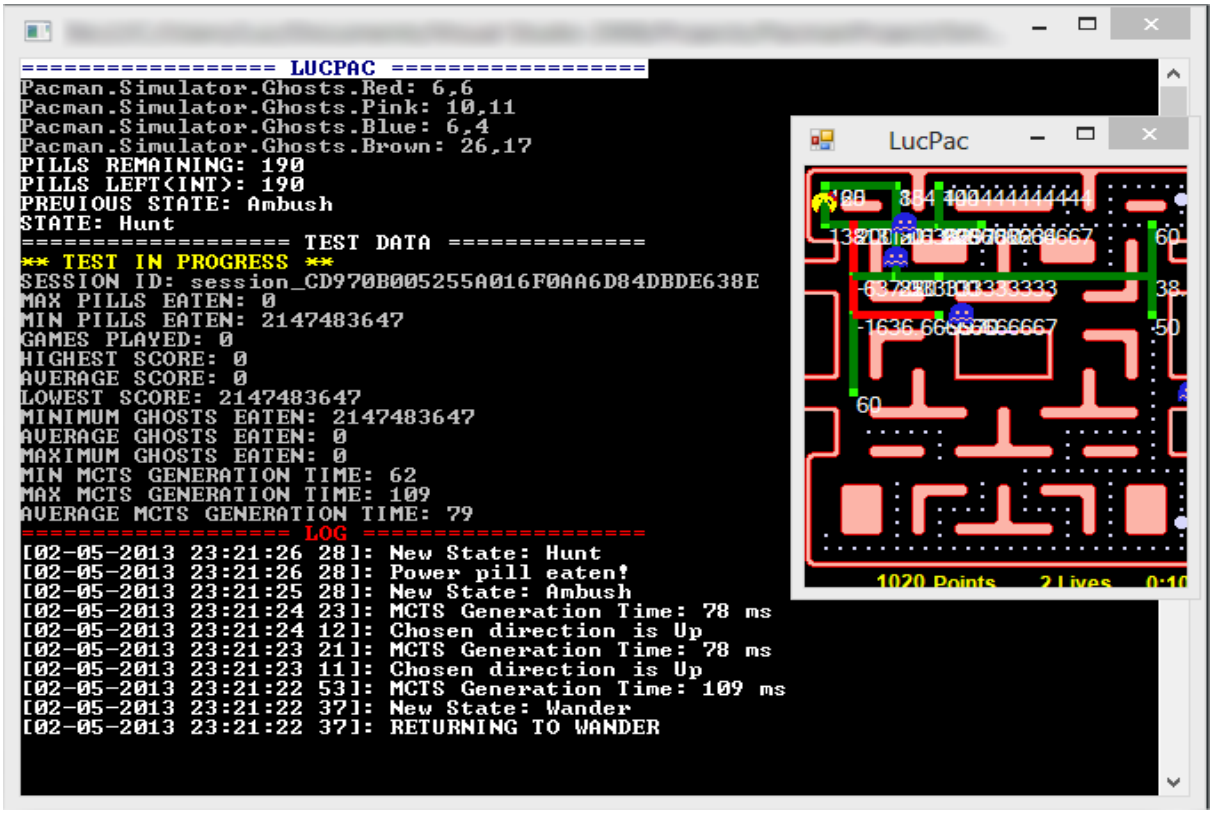

Figure 15 - The appearance of the simulator during runtime. The debug console (left) and the visual interface of the game state (right).

The simulator that we chose to develop our agent with contained functionality that enables us to run game simulations without any form of visual interface (refer to Fig. 15). Additionally, through this method it will produce the results without us having to wait in real time for the agent play outs to finish.

We concluded that for the setup of our testing that it would be appropriate to simulate at least 100 games with various tweaks made to the *Pacman Controller.* The MCTS implementation utilises several different constant parameters that are used to tweak the depth and simulation count.

5.1.1 Monte-Carlo Tree Search Parameters

- **Max Simulations – MAX_SIMULATIONS**

  How many simulations do we perform on the tree before stopping and extracting the next direction that the AI has to go in? This value could be either based on how much time is provided to the simulation stage, or a fixed constraint. Refer to section 2.1 for further understanding.

- **Shallow Simulations – SHALLOW_SIMULATIONS**
  Similar to that of *Max Simulations*, however this value is applied for then traversing C-paths and the agent is not at junction. Shallow simulations consist of simulating games states with a much stricter constraint considering that we are only after the immediate values.

- **Max Cycles – MAX_CYCLES**
  How many random simulations should we perform on a node? During a simulation, another Ms. Pac-Man controller is used to play out these simulations. It is a very simplified agent and only carries out random moves each time it is called.

- **Expansion Threshold – EXPANSION_THRESHOLD**
  How many times does a child node within the tree have to be visited before we consider it exhausted and ready for expansion?

- **Evaluation Method**
  As mentioned with section 2.1, we displayed that (Pepels and Winands, 2012) utilized two separate evaluation methods for when expanding nodes within the search tree of which were UCB1 and UCB-tuned. They state within their research output that they had more consistent and better results when there was a higher simulation and expansion count.

- **Layer Threshold – LAYER_THRESHOLD**
  This value is considered to be the absolute depth that the tree can expand the children to regardless of how many samples have been done on the node that is returned as the *Upper Confidence Tree*.

5.1.2 Finite State Machine Parameters

- **Ambush Distance – AMBUSH_DISTANCE_THRESHOLD**
  This is considered as the distance in which the agent must be from the power pill before entering the *Ambush* state.

- **Flee Change Threshold – FLEE_CHANGE_THRESHOLD**

The value defined will be the distance in which the agent has to be away from the nearest ghost in order to transition back to the wandering state.

- **Flee Distance – FLEE_DISTANCE**
  The ghosts within the maze must be within a certain distance of the agent before the "Flee Distance" condition is met.

- **End Game Distance – END_GAME_DISTANCE**
  How far away must the agent be from the nearest pill before the *End Game* state is activated?

Furthermore I aim to determine how the results will vary when I enable for child nodes of a branch to return back in the same direction that their parent came from. For example, child A from the root node may extend towards the left. Should *Child A* expand and generate children, one of those children may expand back in the direction that *Child A* came from technically causing for an overlap in paths. (Pepels and Winands, 2012) remove the possibility of this happening (noted as *reverse moves*) when they conduct their tests on their MCTS enhanced agent, however I feel that by enabling it, it could enable for more thorough results.

In regards to the back propagation process itself, we intend on looking at the performance difference between using evaluation formulas UCB1 and UCB-tuned, both presented by (Auer et al., 2002) as a means of determining the most optimal reward from a search tree. Referring back to section 2, (Samothrakis et al., 2011) displayed within their MCTS research that with fewer simulations and a higher search-tree depth they were consistently able to achieve a range of scores that were much higher. Whereas when the UCB1 formula was used for tree evaluation, there was no real correlation between the tree-depth and the amount of simulations performed.

For each agent that we have prepared, we will utilize varying configurations during testing to determine what the difference in agent performance is like. Each configuration will make use of a different set of values for each of the available parameters of the respective agent. These are stated in section 5.1.1 and section 5.1.2.

5.1.3 Testing Machine

Based on the research that was conducted by (Gallagher and Ledwich, 2007), we believed that it would be important to note the hardware that we would be carrying out these test on. The

reason for this is that the speed at which the MCTS simulations are completed would vary dependant on the machine that it's run on, therefore it's important to note that the results generated from our tests are in context of the hardware we have. These are the specifications of the computer that the tests will be completed and evaluated on.

- I7 950 @ 3.6 GHZ
- 12 GB DDR3 RAM
- Nvidia GTX 580 video card.

It is noted that the strength of the video card will not have any real effect in the calculations of the MCTS simulation at runtime.

5.1.4 Controller

The following is a summarisation of the controllers and the metrics that we aim to measure that we are going to be testing for our research.

5.1.4.1 Metrics

For each configuration during testing, we will be looking for a collection of values from our simulator test bed that will be used to determine the performance of our agent. Additionally, we will be recording screenshots of the game state of which will be stored in the corresponding configuration directory stated section 5.2.1. Through the observation of the screenshot, we will be able to recognize any trends that may have occurred in the behaviour of the agent such as pincer moves which was stated in section 2.1 of our literature review.

Scoring

Referring back the section 2.2.1 of the literature review, using the score of the game within our simulator test-bed would not achieve would not provide any reliable information on their own. Therefore for the purpose of our research, we will be using the score solely for comparing the varying performance between the varying setups that we have in place to display the effectiveness of combining the strategy of an FSM and the heuristics of the MCTS algorithm.

MCTS Generation times

To understand whether or not our MCTS implementation has been effective enough in terms of generation times we will be additionally monitoring how long it takes to generates varying lengths of a tree when the parameters are adjusted accordingly. Please refer to our testing setup to learn more.

Ghosts Eaten

Recording how many ghosts were eaten during the game simulations will helps us determine how successful our agent is when being placed in the offensive. For instance, with the usage of our *Ambush* strategy within the Ms. Pac-Man, we will be determining how successful it is in consuming the enemy ghosts when placed next to the power pill until the ghosts come close.

Pills Taken

Due to the way that the MCTS algorithm works, we aim to find out how successful it is in consuming a large quantity of pills whilst surviving for a long period of time.

Ghost Kills

Before the end of every game, we aim to record how many times our agent was killed by each respective ghost. By using this metric we can then determine how each configuration interacts with the ghosts and how likely they are to be eaten by the ghost.

Life Time / Round Time

As a way of determining how our experiment configurations are at surviving for long periods of time against the enemy ghosts, we will record the life time within the game as well as how long in total they survive within each round.

5.1.4.2 Pure MCTS behaviour

To compare our implementation of the Monte-Carlo Tree Search algorithm, we decided that it would be appropriate to determine how well our evaluation formulas performed in contrast to the other published research such as (Robles and Lucas, 2009). In which the behaviour of the agent was completely based on the scores that were generated from the heuristics of the simple tree search. Therefore with this test we will remove all fixed strategies from the agent and see how it fares when it simply has the results of the MCTS algorithm to base its moves from. From this, we will be able to determine how effective the algorithm on its own would be without any form of hand-coded strategies to dictate the navigation of the agent around the maze.

5.1.4.3 FSM with scripted behaviour

The same finite state machine structure as our main implementation, this will utilize a set of hand-coded scripted conditions within the wander state rather than basing its method on based its preferred direction based on the UCB scoring that is generated from the tree node.

The behaviour in particular of this agent will emulate some of the details mentioned by (Thompson et al., 2008) in that at each junction the agent will evaluate every possible direction that it can take and determine which one offers it the largest amount of pills for consumption. The reason for our choice in using this methodology in our testing is due to the success that was demonstrated in their experiments. Therefore, comparing this behaviour in contrast to a heuristic based method would help determine if there was an improvement in performance.

Just like the End Game state that we aim to implement within our main agent, should the agent fail to find any adjacent pills then it will generate the shortest path to the nearest one in the maze and resume to the *Wander* state.

Ambush

The functionality of this state will be identical to that of our main implementation. While in the wandering state, the Ms. Pac-Man agent will detect whether it is adjacent to a power pill. When a condition is met that it is in fact close enough, the controller will remain idle within the same position until a ghost becomes closer. When the ghost is considered close enough, the controller will head towards the power-pill and then transition to the "Hunt" state.

Flee

During the fleeing state, the actions of the agent will be similar to that of the original implementation specification in section 4.1.1. The agent will continue to move in the inverse direction of the agent, should it determine that the nearest ghost is within a distance of 2 Manhattan values (i.e. nodes).

Wander

While Ms. Pac-Man is within this state, the aim will predominantly to consume the pills that are adjacent to the immediate position of the Pacman agent itself. When there are no pills that are within close proximity to the agent, it will then proceed to find the shortest path towards one. The key difference between this and our main implementation is that there will be no heuristic information detailing whether certain tunnels are considered safe or not. Therefore it will simply follow which ever maze that has adjacent pills at.

End Game

As previously mentioned, the End Game state is activated when there is a limited of pills remaining within the maze. For this variation of the agent, I will utilize a threshold of 25

remaining pills within the maze. When the game starts there are initially 285 pills that are available within the maze.

5.1.5 Ghost Behaviour

Throughout the testing stage, all four ghosts will be present within the maze. In our simulator, each ghost has a separate set of rules in how they behave. For additional testing in some cases I will utilize several instances of the same ghost to determine how agile the agent when presented with enemy characters that portray behaviour that is the same or at the very least similar.

Red

- A random distance threshold is generated when the ghost is created for the first time.
- Should the Ms. Pacman agent be within a fixed radius of Red, then run the following conditions.
    - The Red ghost at any given point cannot stall or forever continue in the same direction.
    - On each tick, check the distance between the current node that the ghost is occupying and the node that the Pacman controller is occupying.
    - If the Pacman agent is within distance, then do the following move within the direction that Pacman is in.
        - For example, if Pacman meets the condition of being to the right, then change the preferred direction for the next update tick to the right.

Pink

- If the pink ghost is within a certain distance of the Pacman entity within the Game State and the random number that is generated returns as 0, then move randomly around the maze.
    - Moving randomly entails that the ghost will select a random direction to move in so long as it's not the inverse of the previous direction that they moved in (i.e. last direction was right, so next can't be left).
- If Pacman is within a certain pixel distance of 120 and the current direction value of the Pink ghost is not set to "none", then move in a similar pattern to the Red ghost.
- Else if the Pacman agent is not within a certain distance then, the pink ghost will then determine whether the Pacman agent is above, below, left or to the right of the ghost.

- Based on the aforementioned conditions, the Pink ghost will attempt to go in the direction presuming that it is not the inverse of the direction that it is currently going in when the change of direction is attempted.

Blue

- If the Ms. Pacman agent is not within a fixed distance threshold, then select a random direction to move in other than the inverse of the current direction that the ghost is going in.
- If the Blue ghost is within a certain distance of the Red ghost, then start using the same move set as the Red ghost to generate a mob like movement pattern.
- Else if the Blue ghost is within a certain distance of the Pacman agent, proceed to move in the direction that the agent is in regards to the Blue ghost.

Brown

- Moves randomly around the maze, however the next direction of movement cannot be the inverse of the previous current direction that the ghost is moving in. No further conditions or actions are applied.

5.2 Data Collected and results

After simulating a total of 100 games for each setup within our simulator environment we concluded with the following results.

We've provided each test case with a unique identifier that is generated from an MD5 hash of the current time stamp (provided by the *DateTime* object) up to the second. This unique identifier enables us to collate the testing information that is relevant to each agent that we are working with. For instance, should we wish to capture any images or serialize and save JSON text, the files will be appropriately placed in the folders named after the session ID.

5.2.1 Folder Structure

{SessionID}/

- **Logs/**
  - EndOfTest_{datetimestamp}.txt – The final stats that are recorded before.
  - Output_{datetimestamp}.txt – The total collection of log entries that were stored.
- **Images/**

- Endofround_{datetimestamp}.bmp – Image that is scraped of the game state before the game is considered over.
- Eatenpowerpill_{datetimestamp}.bmp – Image that is scraped of the game once the agent has consumed a power pill.
- Eatenbyghost_{datetimestamp}.bmp – Image that is scraped of the game before the agent is eaten by the enemy ghost.

5.2.2 Main Implementation

5.2.2.1 Configuration 1

Test Session ID: session_C37E7E8698A4DEFCFB7ABE00AA858C0F

1. MCTS Parameters

Parameter Name	Parameter Value
Max Cycles	5
Expansion Threshold	3
Layer Threshold	7
Evaluation Method	UCB1
Max Simulations	5 (Fixed Constraint)

2. FSM Parameters

Parameter Name	Parameter Value
Power Pill Adjacency (Ambush)	5 Nodes Distance (Manhattan)
Flee Distance (Ghost Adjacency)	7 Nodes Distance (Manhattan)
Ambush Distance Threshold	5 Nodes Distance (Manhattan)
End Game Distance	4 Nodes Distance (Manhattan)

3. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	32 minutes 6 seconds
Average Time Per Game	15 seconds 15 milliseconds
Longest Time Per Game	46 seconds 36 milliseconds

4. Scoring

Minimum Score	**Average Score**	**Max Score**
500	3823	9510
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
26	115	215
Minimum Ghost Eaten	**Average Ghost Eaten**	**Maximum Ghost Eaten**
1	6	13

5. Monte-Carlo Tree Search Generation

Minimum Generation Time	Average Generation Time	Maximum Generation Time
109 ms	147 ms	250 ms

5.2.2.2 Configuration 2

Test Session ID: session_29EBF1B5F210DD719E950AF2C65002E0

The purpose of this configuration is to determine whether a more in-depth heuristics search by the MCTS algorithm yields any more positive results than from the previous test. Referring back to section 2.1 of the literature review, (Samothrakis et al., 2011) mention that the performance of their agent varied with the UCB1 evaluation method based on the amount of simulations were conducted.

Therefore, for the configuration in this experiment I aim to alleviate the fixed constraint on the Max Simulations to something higher and increase the amount of possible layers that there can be in the tree. We aim to achieve more appropriate low-level direction choices from the agent in question by enabling for the simulations to be more thorough.

1. MCTS Parameters

Parameter Name	Parameter Value
Max Cycles	5
Expansion Threshold	15
Layer Threshold	12
Evaluation Method	UCB1
Max Simulations	7 (Fixed Constraint)

2. FSM Parameters

Parameter Name	Parameter Value
Power Pill Adjacency (Ambush)	5 Nodes Distance (Manhattan)
Flee Distance (Ghost Adjacency)	7 Nodes Distance (Manhattan)
Ambush Distance Threshold	5 Nodes Distance (Manhattan)
End Game Distance	4 Nodes Distance (Manhattan)

3. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	31 minutes 28 seconds 51 milliseconds
Average Time Per Game	14 seconds 89 milliseconds
Longest Time Per Game	37 seconds 4 milliseconds

4. Scoring

Minimum Score	**Average Score**	**Max Score**
140	4015	8600
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
14	112	208
Minimum Ghost Eaten	**Average Ghost Eaten**	**Maximum Ghost Eaten**
0	6	12

5. Monte-Carlo Tree Search Generation

Minimum Generation Time	**Average Generation Time**	**Maximum Generation Time**
171 ms	221 ms	358 ms

5.2.2.3 Configuration 3

Test Session ID: session_F96E5588BCC184DCCFA7BA22B460CB07

The same MCTS configuration settings are used this time around as the ones provided within the first configuration, however the only difference is that this time is that we are making use of a different method of evaluation for choosing the most optimal direction from the tree. Referring back to our literature review in section 2.1, we state that (Pepels and Winands, 2012) demonstrated positive results when making use of this formula for generating UCB scores with the current node (i.e. a simulated game) and its parent's score.

1. MCTS Parameters

Parameter Name	Parameter Value
Max Cycles	5
Expansion Threshold	3
Layer Threshold	7
Evaluation Method	UCB-Tuned
Max Simulations	5 (Fixed Constraint)

2. FSM Parameters

Parameter Name	Parameter Value
Power Pill Adjacency (Ambush)	5 Nodes Distance (Manhattan)
Flee Distance (Ghost Adjacency)	7 Nodes Distance (Manhattan)
Ambush Distance Threshold	5 Nodes Distance (Manhattan)
End Game Distance	4 Nodes Distance (Manhattan)

3. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	27 minutes 64 seconds
Average Time Per Game	13 seconds 10 milliseconds
Longest Time Per Game	42 seconds 48 milliseconds

4. Scoring

Minimum Score	**Average Score**	**Max Score**
280	3035	10230

Minimum Pills Eaten	Average Pills Eaten	Maximum Pills Eaten
28	103	208
Minimum Ghost Eaten	**Average Ghost Eaten**	**Maximum Ghost Eaten**
0	4	13
Minimum Levels Cleared	**Average Levels Cleared**	**Maximum Levels Cleared**
0	0	0

5. Monte-Carlo Tree Search Generation

Minimum Generation Time	Average Generation Time	Maximum Generation Time
124 ms	181 ms	281 ms

5.2.2.4 Configuration 4

Test Session ID: session_580C95A11A10AD6853CD4F0E258FC100

Within this configuration we aim to determine whether or not our agent is capable of returning optimal results if it is able to carry out more simulations during generation of the MCTS tree. (Pepels and Winands, 2012) demonstrate the UCB-Tuned formula consistently returns better scoring from the game when the simulation constraint is higher. Based on the results returned from the previous configurations we expect for there to be a higher latency in the generation of the tree, however we hope for the max latency to be under the time of 500 ms.

1. MCTS Parameters

Parameter Name	Parameter Value
Max Cycles	5
Expansion Threshold	15
Layer Threshold	12
Evaluation Method	UCB-Tuned
Max Simulations	25 (Fixed Constraint)

2. FSM Parameters

Parameter Name	Parameter Value
Power Pill Adjacency (Ambush)	5 Nodes Distance (Manhattan)
Flee Distance (Ghost Adjacency)	7 Nodes Distance (Manhattan)
Ambush Distance Threshold	5 Nodes Distance (Manhattan)
End Game Distance	4 Nodes Distance (Manhattan)

3. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	33 minutes 89 seconds 65 milliseconds
Average Time Per Game	12 seconds 55 milliseconds
Longest Time Per Game	44 seconds 84 milliseconds

4. Scoring

Minimum Score	**Average Score**	**Max Score**

160	2705	8160
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
16	91	215
Minimum Ghost Eaten	**Average Ghost Eaten**	**Maximum Ghost Eaten**
0	4	11
Minimum Levels Cleared	**Average Levels Cleared**	**Maximum Levels Cleared**
0	0	0

5. Monte-Carlo Tree Search Generation

Minimum Generation Time	**Average Generation Time**	**Maximum Generation Time**
608 ms	680 ms	842 ms

5.2.3 Scripted behaviour and Finite State Machine

As stated in section 5.1.2.1.3, with this experiment setup we intend to determine whether or not there is a significant performance boost when utilising a hand-coded strategic based agent. 3 separate configurations will be applied for testing to determine which set of configurations are considered ideal for testing. Based on the results outputted from this setup, we will conduct one final experiment combining the most optimal FSM parameters found from these configurations with our main implementation.

Configuration 1

Test Session ID: session_79C98964B07899B3A4D61CD07B83F208

1. FSM Parameters

Parameter Name	Parameter Value
Flee Distance (Ghost Adjacency)	7 Nodes Distance (Manhattan)
Ambush Distance Threshold	5 Nodes Distance (Manhattan)
End Game Distance	4 Nodes Distance (Manhattan)
Flee Change Distance	5 Nodes Distance (Manhattan)

2. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	1 hour 6 minutes 31 seconds
Average Time Per Game	39 seconds 91 milliseconds
Longest Time Per Game	1 minute 14 seconds 8 milliseconds

3. Scoring

Minimum Score	**Average Score**	**Maximum Score**
460	4646	10370
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
46	150	311
Minimum Ghost Eaten	**Average Ghost Eaten**	**Maximum Ghost Eaten**
0	6	15
Minimum Levels Cleared	**Average Levels Cleared**	**Maximum Levels Cleared**
0	0	0

Configuration 2

Test Session ID: session_F1970BFDEFCD9E4A073C812C0A50DDB0

1. FSM Parameters

Parameter Name	Parameter Value
Ambush Threshold	4 Nodes Distance (Manhattan)
Flee Threshold (Ghost Adjacency)	5 Nodes Distance (Manhattan)
End Game Distance	4 Nodes Distance (Manhattan)
Flee Change Threshold (Return to Wander)	3 Nodes Distance (Manhattan)

2. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	1 hour 2 minutes 38 seconds
Average Time Per Game	37 seconds 48 milliseconds
Longest Time Per Game	1 minute 40 seconds 76 milliseconds

3. Scoring

Minimum Score	**Average Score**	**Maximum Score**
850	4065	14760
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
73	162	393
Minimum Ghost Eaten	**Average Ghost Eaten**	**Maximum Ghost Eaten**
0	5	20
Minimum Levels Cleared	**Average Levels Cleared**	**Maximum Levels Cleared**
0	0	1

5.2.4 Pure MCTS approach

The same MCTS configurations will be applied to the MCTS-based agent as our main implementation (as stated in Configuration 1). The directions in which the Ms. Pac-Man agent will head in will be based entirely on the results that are generated from the search-tree. Our aim with this experimental setup is to determine how effective using the Monte-Carlo Search Tree is.

Configuration 1

Test Session ID: session_0554FF1D685CB94E47F3D4040CB7030F

1. MCTS Parameters

Parameter Name	Parameter Value
Max Cycles	5
Expansion Threshold	3
Layer Threshold	7
Evaluation Method	UCB1
Max Simulations	7 (Fixed Constraint)

2. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	1 hour 7 minutes 6 seconds
Average Time Per Game	13 seconds 9 milliseconds
Longest Time Per Game	31 seconds 28 milliseconds

3. Scoring

Minimum Score	**Average Score**	**Maximum Score**
20	1346	3230
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
2	107	213
Minimum Ghosts Eaten	**Average Ghosts Eaten**	**Maximum Ghosts Eaten**
0	0	5
Minimum Levels Cleared	**Average Levels Cleared**	**Maximum Levels Cleared**
0	0	0

4. Monte-Carlo Tree Search Generation

Minimum Generation Time	Average Generation Time	Maximum Generation Time
93 ms	144 ms	250 ms

Configuration 2

Test Session ID: session_694C95EAF7BD03FE1FF9B6C6269FC766

Due to the improved performance noted in our *Main Implementation* when put against the UCB1 formula, this configuration aims to determine what the performance of the MCTS algorithm is like when it makes use of the UCB-Tuned formula.

1. MCTS Parameters

Parameter Name	Parameter Value
Max Cycles	5
Expansion Threshold	3
Layer Threshold	7
Evaluation Method	UCB-Tuned
Max Simulations	7 (Fixed Constraint)

2. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	1 hour 4 minutes 5 seconds
Average Time Per Game	11 seconds 43 milliseconds
Longest Time Per Game	35 seconds 2 milliseconds

3. Scoring

Minimum Score	**Average Score**	**Maximum Score**
160	1145	3070
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
16	86	158
Minimum Ghosts Eaten	**Average Ghosts Eaten**	**Maximum Ghosts Eaten**
0	0	4
Minimum Levels Cleared	**Average Levels Cleared**	**Maximum Levels Cleared**
0	0	0

4. Monte-Carlo Tree Search Generation

Minimum Generation Time	**Average Generation Time**	**Maximum Generation Time**
110 ms	186 ms	312 ms

Configuration 3

Test Session ID: session_ 4A3E1C7D84159273F027EF0B6C258CD8

To further determine whether a more lenient simulation constraint can improve the performance of the agent, we reused the same evaluation method as the first configuration and increase the amount of configurations that it was allowed to do.

1. MCTS Parameters

Parameter Name	Parameter Value
Max Cycles	5
Expansion Threshold	5
Layer Threshold	7
Evaluation Method	UCB1
Max Simulations	12 (Fixed Constraint)

2. Other Results

Result Name	Result Value
Total Games	100
Total Simulated Time	1 hour 4 minutes 8 seconds
Average Time Per Game	13 seconds 93 milliseconds
Longest Time Per Game	42 seconds 76 milliseconds

3. Scoring

Minimum Score	**Average Score**	**Maximum Score**
340	1446	3120
Minimum Pills Eaten	**Average Pills Eaten**	**Maximum Pills Eaten**
34	118	216
Minimum Ghosts Eaten	**Average Ghosts Eaten**	**Maximum Ghosts Eaten**
0	0	3
Minimum Levels Cleared	**Average Levels Cleared**	**Maximum Levels Cleared**
0	0	0

4. Monte-Carlo Tree Search Generation

Minimum Generation Time	**Average Generation Time**	**Maximum Generation Time**
265 ms	325 ms	453 ms

5.3 Analysis and Critical Evaluation

5.3.1 Finite State Machine and Scripted Behaviour

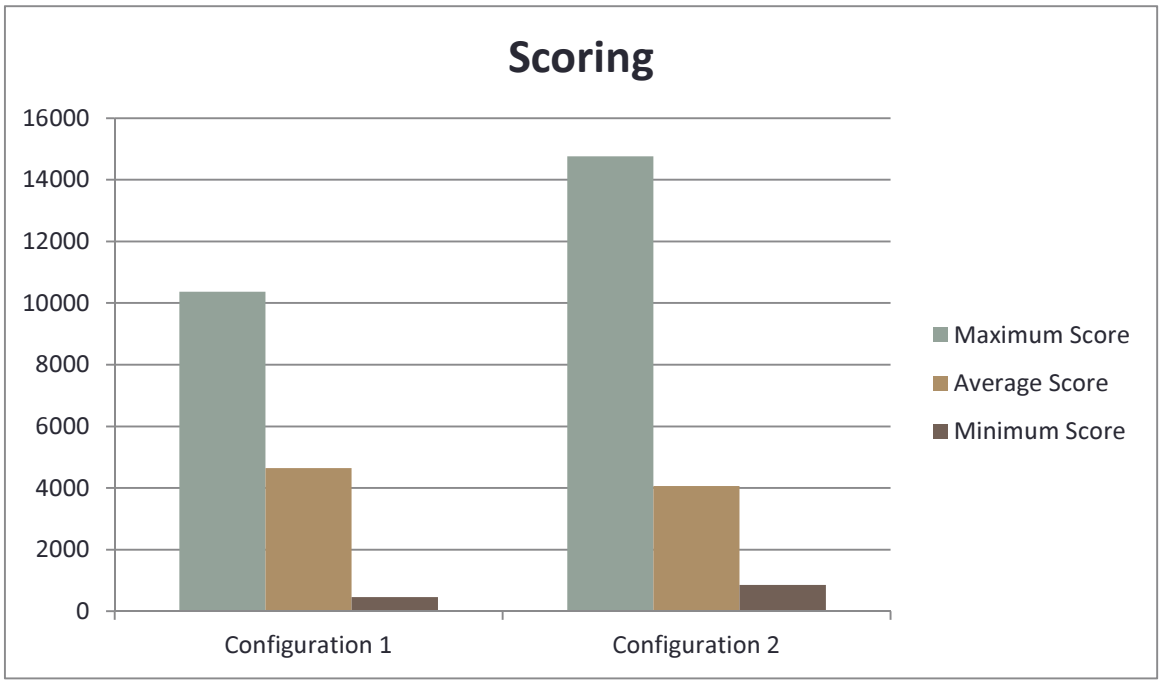

Figure 16 - Scoring output from the scripted behaviour agent.

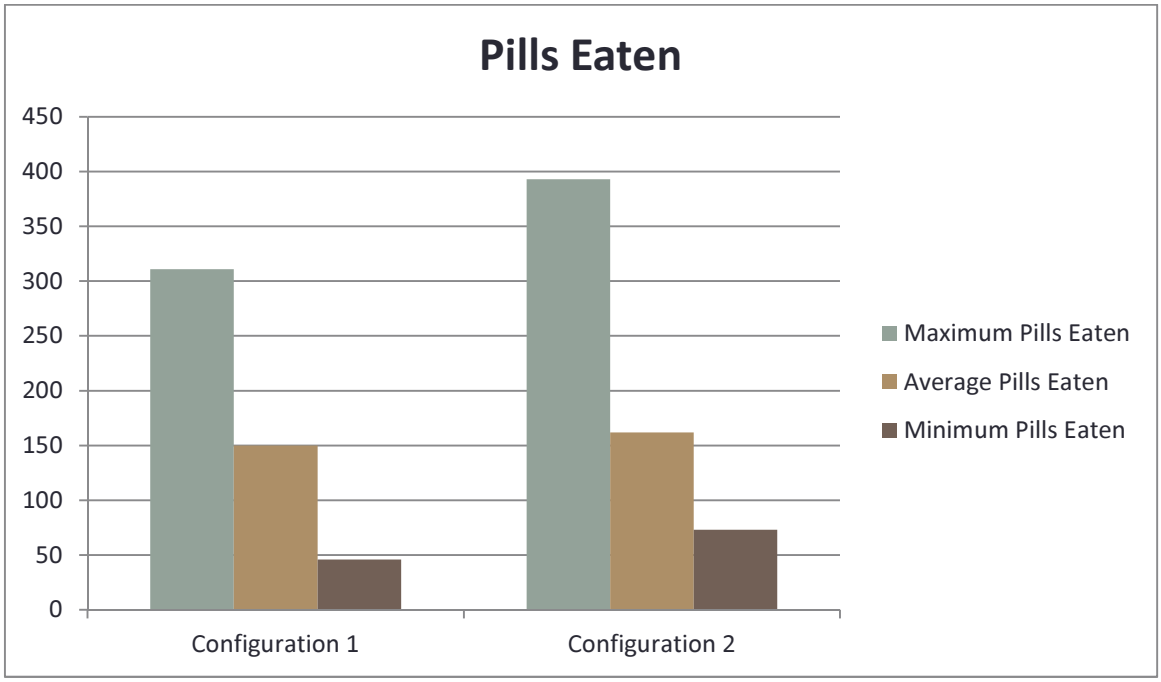

Figure 17 - Pills eaten from the scripted behaviour agent

5.3.1.1 Configuration 1

Immediately we can tell that an AI with scripted controls is capable of generating a high enough score in the first level of the game with a resulting maximum score of 10000. Additionally we see that the Ambush & Hunt strategy on average is working appropriately

due to the average count of ghosts being eaten totalling at 6 per game and the max amount of ghosts eaten in out of the 100 games is 15. As a result, the greedy-random strategy presented by (Thompson et al., 2008) for evaluating the best route to take is being applied well considering that the highest amount of pills consumed is a 311 and on average the agent will take 164.

5.3.1.2 Configuration 2

In an effort to determine the most optimal hand-coded strategies, we altered the Finite State Machine parameters slightly to see whether we could obtain any better performance from the agent should the agent be allowed to leave the *Flee* state quicker. With the adjustments applied, the result set returned appears to be very positive with a maximum score of more than 14,000 and on average the agent is scoring approximately 4,000. If we were to compare this configuration with the former, we can determine that by enabling the agent to get out of the *Flee* state early on that the agent is capable of scoring better and losing less. Strangely however, we noticed that the time in which the agent survives for within the game is substantially less than the first configuration.

5.3.2 Pure MCTS Approach

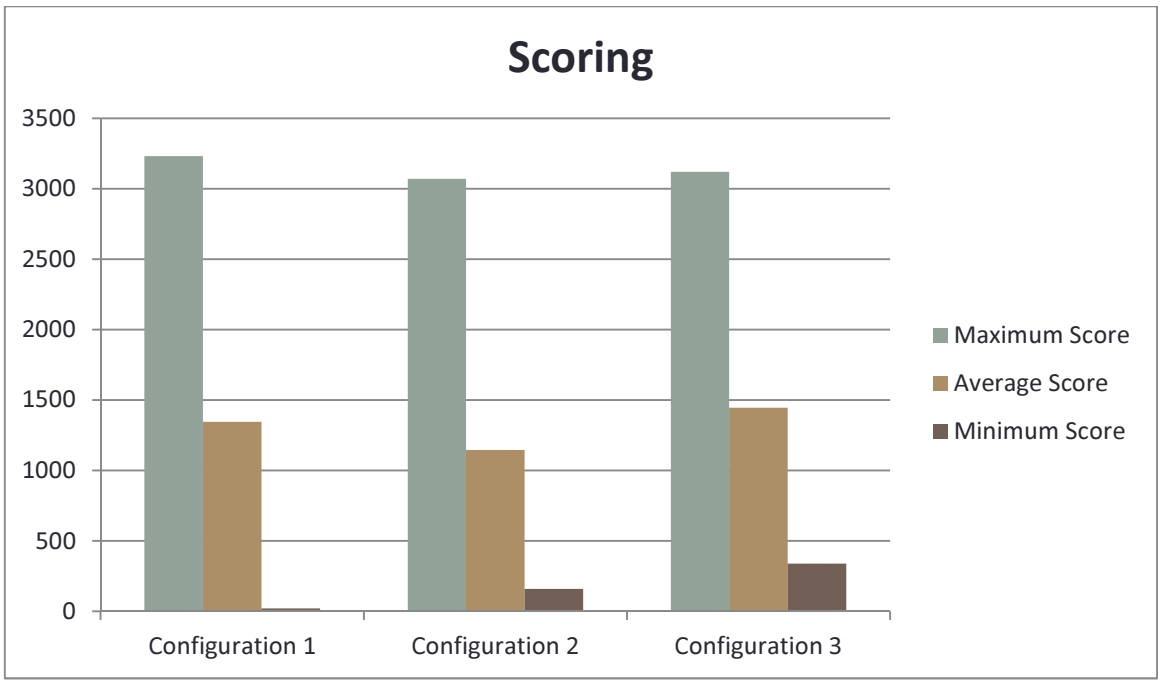

Figure 18 - Scoring output for our Pure MCTS agent

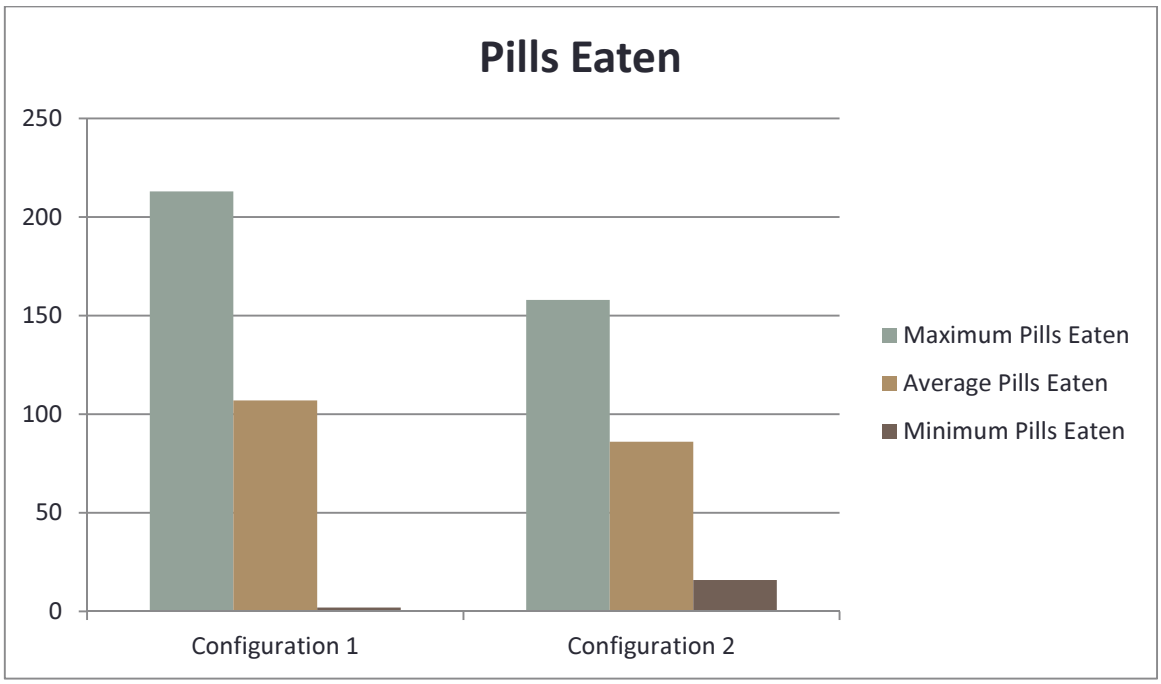

Figure 19 - The pill consumption scoring from our Pure MCTS agent

With the results from this test we begin to see some of our concerns mentioned in the literature review come to light. For instance, when the Pac-Man agent consumes the majority of the pills at one part of the level, we begin to notice that the returning results from the agent are consistently 0 meaning that the agent has no particularly focused direction to be headed in.

In comparison to our main implementation, we can see that the agent tends to survive for a longer period of time when using nothing but the MCTS algorithm inside the maze. This is consistent with both configurations under this test bed. This leads us to believe that perhaps by applying the MCTS algorithm in other states there will be a higher rate of survival. For instance, we would allow for the algorithm to influence the pathfinding measures as stated in (Tong and Sung, 2010).

5.3.2.1 Configuration 1

We can see immediately that through using a Pure MCTS strategy that the scoring is typically lower due to the lack of ghosts that were eaten by the agent during the test session. This consequently suggests that for the most optimal score, there has to be some form of strategic input in the way that the Ms. Pac-Man agent moves within the maze. Furthermore we can see that the MCTS algorithm is capable of navigating the maze efficiently based on the pills that are available through each C-Path. This is demonstrated by our results in which a maximum of 213 pills out of 250 were consumed and the average round time was approximately 2 minutes.

During our preliminary tests with this configuration we did notice that the scoring of the trees that headed towards the direction of edible ghosts didn't necessarily provide a more rewarding return for the agent. We believe that this could be something to do in the way that the UCB scoring formula averages out the scores from child nodes within branches of the MCTS search tree.

5.3.2.2 Configuration 2

Similar to how we approached the second configuration in the in the testing of the main implementation, we decided that this time around we wanted to determine how well the UCB-Tuned formula stated in (Auer et al., 2002) operated on its own. Surprisingly, there was no drastic performance difference between the usage of this formula and that of the UCB1 that was used in the previous configuration.

Furthermore, it would appear that when the UCB-tuned formula is used during the back-propagation process in the generation of the tree that there is on average a slight increase in latency in contrast to the UCB1 formula.

5.3.3 Main Implementation

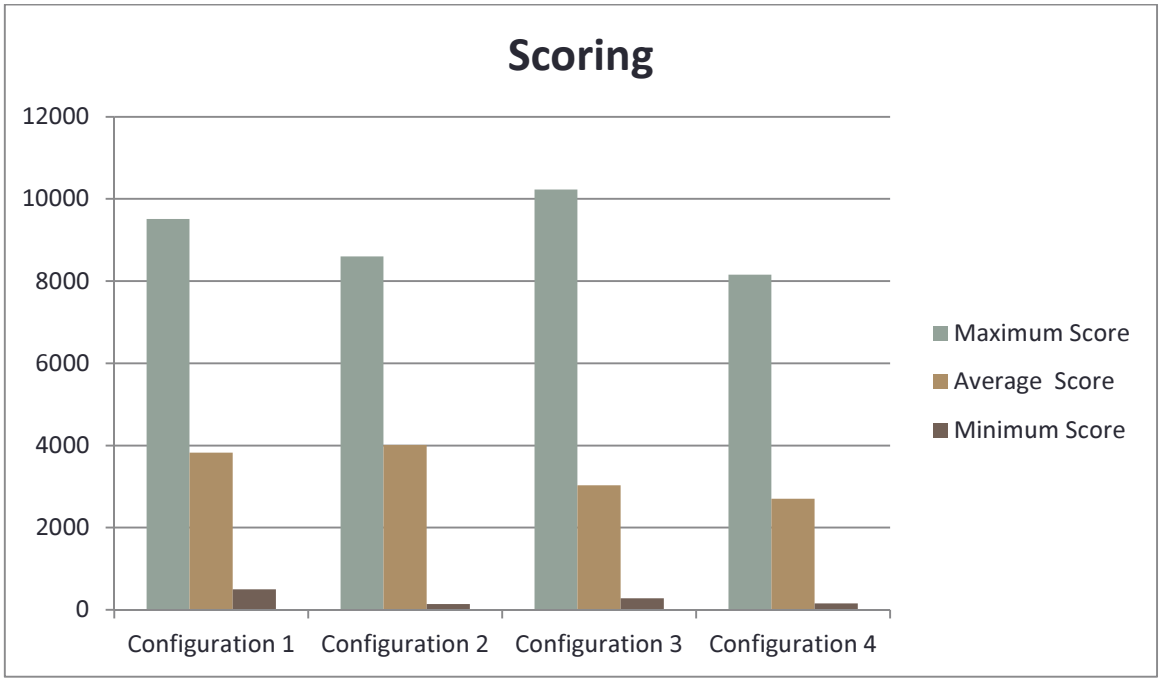

Figure 20 - Scoring output for our Main Implementation

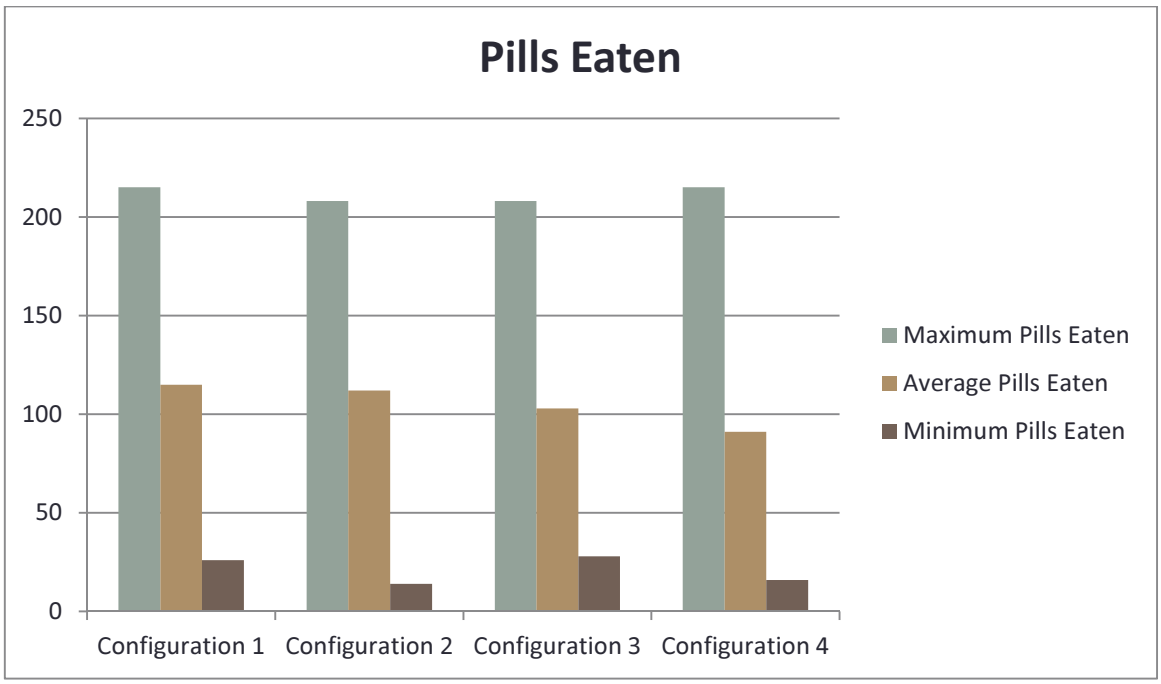

Figure 21 - Pills eaten by our Main Implementation

5.3.3.1 Configuration 1

Comparing to our previous test conducted with the *Scripted Behaviour* controlling our agent, we can see that overall by taking the score into account that the MCTS algorithm combined with a fixed strategy is returning a score that is better than our agent with purely hand-coded strategies. Furthermore, we can see that there are safer traversals across the maze considering that on average more ghosts are being, and at best more are being consumed also.

Although these results are highly positive, it's worth mentioning that the agent is very much capable of losing within a short period of time and that there is a larger spread between the lowest and max score. After evaluating the screenshots that were captured at the end of each game, it's fair to say that the agent was caught out frequently when the ghosts performed a pincer move. As expected, more frequently than not, the Red ghost was responsible for eating Ms. Pac-Man.

Lastly, we can see that the generation times of the MCTS algorithm are within appropriate ranges based on the previous work that has been conducted using this algorithm by (Samothrakis et al., 2011). Referring back to section 2.1 in the Literature Review, if we were to apply this agent to the original software of Ms. Pac-Man through the means of a screen grabbing interface, then the response times that our agent is returning would be viable. Should we wish to pursue this option of agent implementation in the near future, then by using this particular configuration it would be possible.

5.3.3.2 Configuration 2

Using this configuration, we aimed to see if more appropriate decisions would be made at each junction within the maze due to a more lenient fixed constraint that was applied to the generation of the MCTS tree. Having run the tests using the aforementioned configuration, we noticed that there was no real improvement in performance and the scoring was in fact worse. This might have been due to the overreaching nature of the MCTS tree. What is meant by this is that, considering it is simulating so many future game states that are not necessarily relevant to the immediate decision of Ms. Pacman, it might mean that the agent is selection turns based on results that are not even relevant.

The statistics regarding the consumption of ghosts during their mortal state is a similar set of data to the previously used configuration suggesting that the Ambush and Hunt states are performing just as well. We could argue that the minimum ghosts eaten being 0 is due to the Wander state causing the agent to get killed before even reaching the power pills within the maze.

Understandably the MCTS generation times are taking longer time to complete due to the fact that we are allowing it more time to simulate branches across the tree. With the maximum of 340~ ms to complete the computation of the MCTS, this would still be considered respectable should we later make the agent engage with a screen-scraping interface for the original software of the Ms. Pac-Man game.

5.3.3.3 Configuration 3

As we stated within section 2.1, we believed that it would be appropriate to apply various methods of tree evaluation when enabling our Ms. Pac-Man agent to choose the most optimal path to head in. Based on the research demonstrated by (Pepels and Winands, 2012) we can see that there is a higher consistency in better scores when the simulation constraint is a lot more lenient in comparison to our previous configurations used. Additionally, within the results they displayed, the UCB-tuned formula demonstrates much better results (refer to Fig. 17).

Having observed the results that we gained from our tests, we can certainly see that we generate a better maximum score from our agent in contrast to the other configurations of our main implementation. However, it’s fair to notice that the average is of a lower number thus displaying that there was a far less consistency in the performance that was outputted from the

agent in comparison to the previous sets of configurations. Pill consumption within the maze is relatively the same.

6.1 Conclusion

6.1.1 AI Performance

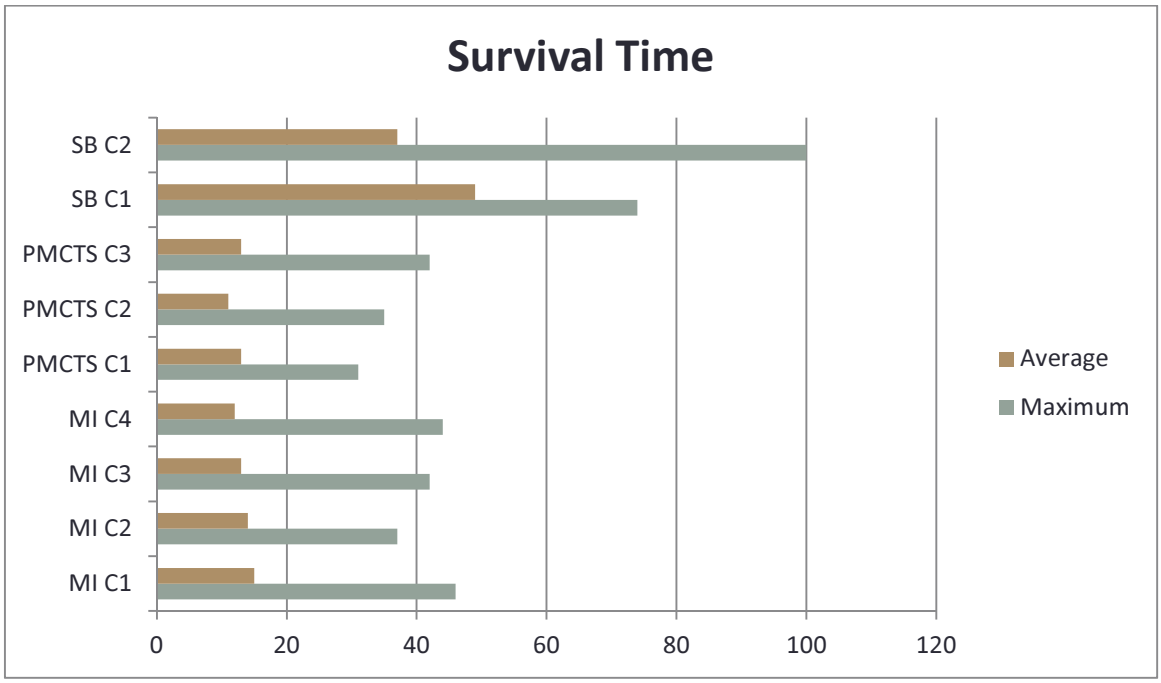

Figure 22 - Chart displaying the total survival time of all our agents and their respective configurations

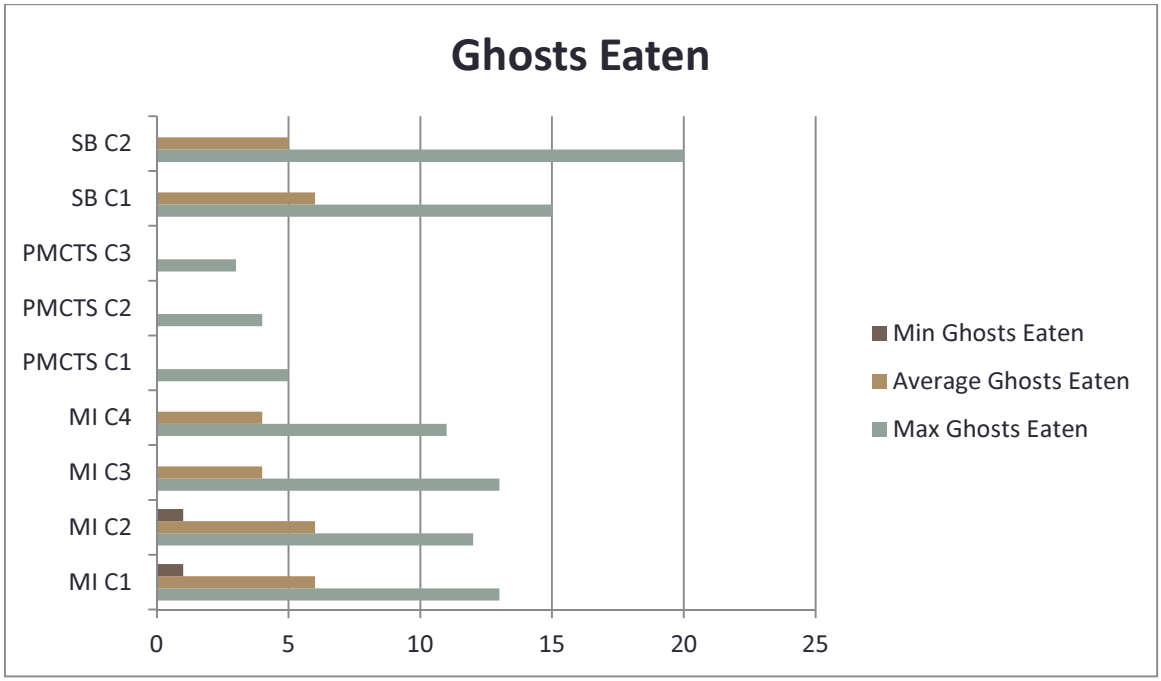

Figure 23 - Graph demonstrating the total sum of ghosts that were consumed by each agent.

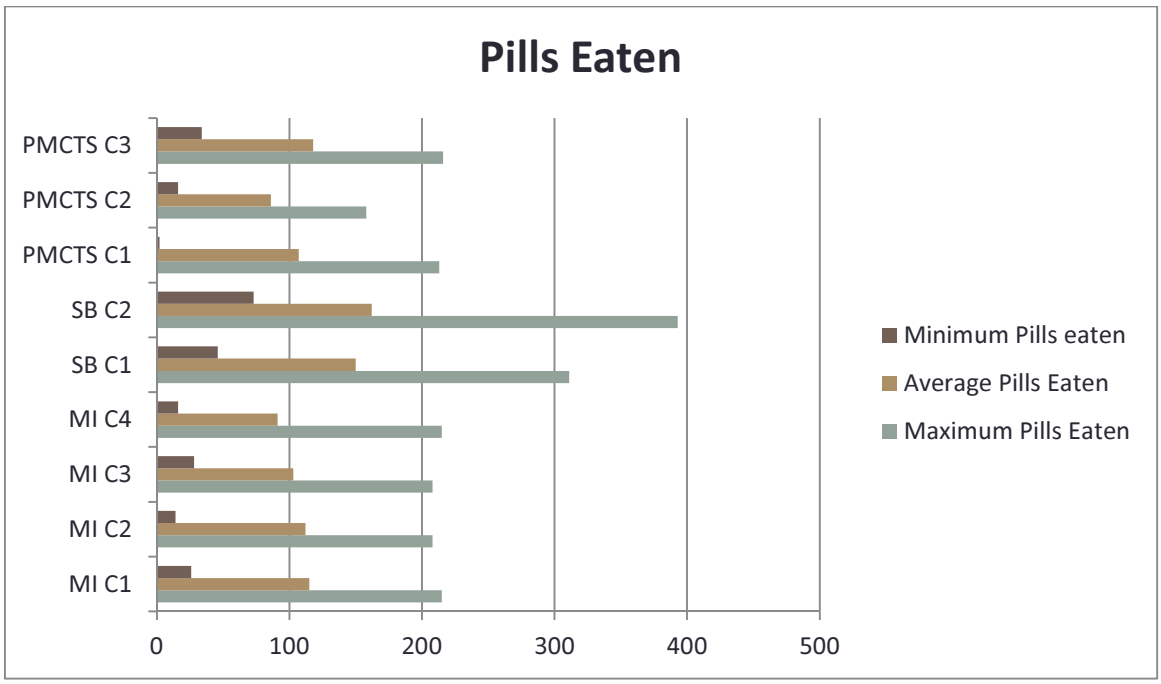

Figure 24 - Pills eaten out of all our agent implementations

Through testing each respective agent, we came across a varied set of results that enabled us to determine that heuristics best-first search method such as Monte-Carlo Tree Search, is better off when making use of a fixed-strategy such as our Finite State Machine. This can be seen in the amount of ghosts that are consumed between our main implementation and that of a Pure MCTS approach. Referring to Fig. 23, we can see that both the scripted-behaviour agent and our main implementation are competitive in that they are capable consuming a large number of ghosts within a game. In a similar sense, if we look at Fig. 24 we can see that our main implementation was capable of consistently consuming more pills than our Pure MCTS agent.

This leads us to believe that by making use of fixed strategies with MCTS, it enables our agent to be substantially more competitive, than if we used MCTS on its own. As discussed within section 2.3, we mentioned that making use of an Ambush strategy similar to that of (Thawonmas and Ashida, 2010) would be beneficial in enabling our ghost to acquire more points. Fortunately as we can see from Fig. 23, this has been largely the case.

In addition, we noticed that when our agent made use of nothing but the MCTS algorithm, the time in which it survived on average was marginally smaller to that of our main implementation (refer to Fig. 22). This leads us to believe that making use of the intermediary *Flee* state is an effective move for enhancing the time in which it survives for. We assume that this is due to the fact that for a short period of time the agent will aim to move as far

away as possible from the nearest ghost before resuming to *Wander*. This is so that should the MCTS tree fail to return an accurate state based on the simulation returning different results to the active game state, then we are able to respond to it appropriately.

More surprisingly, the usage of the Greedy-Random AI (Thompson et al., 2008) demonstrated that there was significant improvement in pill consumption over our main implementation which went against our initial expectations. With our implementation of the MCTS algorithm was intended to navigate the AI within the maze based on the C-Paths that provided the most rewarding score output for the agent. Instead, the scripting behaviour demonstrated that by simply counting the available pills in every direction and navigating based on that premise was sufficient enough, and was even capable of moving onto the next level.

The reason for this may have something to do in the way the agent evaluates potential paths to take at each direction. Considering that it takes into account the future game states of from junctions that are not immediately adjacent to the agent, it might force the agent to not take an otherwise more rewarding path. This could be down to the fact that due to the stochastic way that the ghosts behave, the random simulations carried out within the generation of the tree could be rendered irrelevant. This is because the behaviour of the ghosts within simulation of the tree may not match that of the current game state that the agent is playing in. This relates to a similar idea presented within section 2.1 of our literature review, in that evolutionary algorithms are to be consistently thrown off due to the stochastic nature of the ghosts.

Moreover, we can see that by experimenting with the likes of another node evaluation method such as UCB tuned that we do in fact achieve a worse scoring output. Referring to Fig. 20, we can see the maximum scoring output of the third configuration of our agent (UCB tuned) is better than the other configurations however struggling from a lower average. Similarly in the fourth configuration we are seeing worse performance overall suggesting a similar pattern. Comparing this to our Pure MCTS agent, we notice that the scoring output is just as weak when our agent makes use of the UCB-tuned evaluation method enabling us to conclude that there is no gain through the usage of the formula.

Referring to the first and second configurations used for our main implementation, we noticed that the scoring outcome was in fact worse when we extended the simulation constraint. This leads us to conclude that using a smaller expansion of the tree will produce better results. We believe that this is due to the UCB score from the tree is returning values that are more

relevant to immediate game states. Refer to Fig. 20 to see the scoring difference between the two.

It's noted as well that out of all of our agent implementations, the agent that made use of scripted behaviour was the only one that was able to get onto the next level during testing. From this we can conclude that the Greedy-Random approach specified by (Thompson et al., 2008), although negligent of adjacent ghosts, is in fact effective in clearing pills within the maze. This is reaffirmed by the fact that the same agent consumed a maximum of 392 pills during testing (refer to Fig. 19). Combined with the End Game strategy that our Main Implementation uses, it makes sense as to why it was capable of progressing onto the next level. The *End Game* strategy enables the agent to discover pills that are not immediately adjacent to it, therefore negating the need for the agent to randomly select a direction should it not find a pill within any available directions at a junction.

While the scripted behaviour testing implementation appear to demonstrate better performance overall, there is still a possibility that the MCTS approach can be improved and eventually exceed that of our scripted agent. The reason we believe this is due to two set of results returned by our *Main Implementation* agent. The first configuration displayed better performance than that of second configuration purely because of subtle differences in the parameters used for the MCTS.

We believe that if the MCTS parameters are adjusted accordingly through further investigation that there is the possibility that we could achieve an agent that can compete that of our *Scripted-Behaviour* agent. The problem lies in carrying out more testing with various parameters for both the FSM and the generation of the MCTS. It is noted however that we are hugely constrained by how many simulations we can carry out on the tree with our current implementation having reviewed the results for *Configuration 4* of our main implementation. This constraint could perhaps be alleviated through further improvements with our choice of simulator.

If we were to observe the aims and objectives that were outlined within section 1.1, it's fair to note that we have successfully developed a high scoring agent that combines the functionality of a fixed-strategy approach along with the usage of a heuristics based search method. Additionally we can see that from the returned results that the AI is capable of generating high-level decisions within a reasonable time frame when using the appropriate parameters. Admittedly, the current performance of the agent lacks in comparison to that of our pure finite

state machine solution. It's worth noting however, should the parameters used for the MCTS simulation be adjusted through iterative testing, we are confident that it would produce a more competitive agent for our main implementation.

We have noticed that the AIs performance in terms of how fast it can generate a strategy through MCTS could be improved. While using a fixed constraint of 15 simulations we are seeing latency on average reaching 250 milliseconds. Having investigated this during the preliminary stages of testing, we concluded that it was mostly due to the way that the simulator was cloning objects.

To conclude, we can see that a competitive and high-scoring agent emerges through the usage of both MCTS and Finite State Machines, however both the application of the MCTS algorithm and the configuration behind it could be improved significantly. For instance, we can see between the two UCB1 configurations that were used for our main implementation that they return different results. This leads me to believe that with further tweaking we can obtain even more competitive parameters leading to better performance. The scores demonstrated in the second configuration used for the scripted agent displayed a better min and max scores out of the 100 that were played in total. This means that in regards to our objectives, we were unable to successfully create a hybrid agent results in better performance to that of a scripted agent overall. The scripted-behaviour agent is based on the work demonstrated by (Thompson et al., 2008). We do recognize however that our scripted behaviour agent made use of several more states and methods to achieve the score that it did. However, the results still state that it is capable of generating better results than that of our main implementation.

Although we have concluded that it is an arduous task to select what the ideal configurations between the two combined methods of AI behaviour are, we believe it is worth pursuing. This is demonstrated by the performance improvements displayed over the Pure MCTS approach when a hard-coded strategy is applied. We should also investigate the possibility of implementing the MCTS algorithm in other states so that we can use its heuristics for other purposes. As stated in (Browne et al., 2012), there are various ways in which the result output of a search-tree can be evaluated and such this requires further investigation before we rule out MCTS entirely. Potential usage of the algorithm in other states could include the likes of biasing the reward output to favour paths with power pills, or simply if there are any pills to begin with (End Game).

7. Future Work

7.1 Agent Improvements

7.1.1 Application of MCTS

After carrying out the experimentation runs, we noticed that the usage of the UCB-tuned algorithm did not display any real beneficial advantage to node evaluation within the tree. This may have been largely down to how our tree was structured, as mentioned by (Pepels and Winands, 2012) to utilise the appropriate evaluation function would require the correct structure of the tree and how it is expanded (min-max etc.). Nonetheless, the end of game results shows that there was no improvement in the score that was achieved by the agent when used on its own. In the future, should we decide to proceed with this evaluation formula again, we would have to ensure that the generation of the tree was done so in such a way that would enable for such an evaluation formula to work.

We also believe, in the future, that this algorithm could be used as method of danger detection rather than a means of directing the agent through the maze. We relate this idea to what was presented by (Tong et al., 2011), in which they demonstrated that generating the shortest path to the farthest pill within the maze whilst conducting shallow MCTS searches that they gained promising results from their agent. In context to our main implementation, the MCTS algorithm could influence the generated path of the agent by returning whether or not the current direction is dangerous. If it was discovered to be dangerous, then we would re-plan and find an alternative route.

The reason that we do this is because of the way that the current scripted behaviour works in the *End Game* state. Currently it is simply reactive to the radius in which the ghost is in comparison to the agent. This means that although the ghost could be close, it doesn't necessarily mean that it is of a threat to the agent. For instance, if we refer to section 5.1.5 in regards to the ghost behaviour within our simulator solution, we notice that the behaviour of the brown ghost is completely stochastic. This means that if our agent was adjacent to the Brown ghost, it would erroneously change its currently active state to fleeing when the ghost wouldn't necessarily be approaching the agent. Making use of a shallow MCTS simulation would counter our previous concerns with the algorithm. An example is that it would provide us with relatively accurate results, as the simulation would return what would be likely to occur in immediate future game states. Due to the shallow nature of tree simulation, we can be left assured that the future simulated states would be considered relevant to the direction

that our agent is heading in. The reason for this is that there would be a smaller possibility that the respective ghost's behaviour (refer to section 5.1.5) will change drastically a few moves ahead from the agent's current position.

7.1.1.1 Improving the cloning of game states

We noticed that during the simulation of the MCTS algorithm that one of the major bottlenecks for the algorithm was the way in which the game state was being cloned. In order for the MCTS algorithm to operate, the current game state has to be copied numerous times to be simulated on. Unfortunately, due to the way that the original simulator source code works, it meant that it originally took a long period of time to deep-copy the game state. Due to the way that the *GameState* object is developed, the maze information of the other levels within the game are cloned a long with the one that is being actively used.

This was originally considered costly as we were never actively making use of the other levels data, so it unnecessary for it to be cloned. In the end, we attempted to resolve the issue by only ever cloning the map that was being actively used and the next map within the game. This is so that the level changes to the next during the simulation of the MCTS tree, then the data is available for it to use. It is important to note that it is essential for this information to be available considering that the level progression reward has to be applied to the MCTS evaluation. We feel that this could be optimised further considering that during our tests we managed to hit 500 ms when using the UCB-Tuned evaluation method, with a fixed constraint of 25 simulations.

Like many other AI controller interfaces for Ms. Pac-Man agents, our simulator through the use of discretization, presents the Ms. Pac-Man maze layout as a series of nodes (Fitzgerald and Congdon, 2009). This is the same for the simulator that we used, except for each node within the game also contains a reference to the 4 adjacent nodes to its position in the maze. This is a problem during the cloning state, considering that it will automatically attempt to clone the 4 adjacent nodes as the *Node* object holds a reference to those within its class definition. This then initially lead to a stack overflow issue when attempting to clone the entire game state. In the end, we prevented this by simply storing a 2D array of enumerable values that displays the state of a given node in our graph.

In the future we aim to refactor the simulator that we use by changing how the nodes are represented in the maze so that there isn't a similar time consuming issue. Moreover, we aim to speed up the rate that the simulator clones the game state object by removing the amount of

objects that it has to copy, and simply copying vital information required for the simulation. Upon achieving this, we may be able to see some further improvements in terms of lower latency and a less strict constraint for our simulation threshold.

7.1.2 Counter-acting Pincer Moves

Figure 25 - Ms. Pac-Man being caught out by a pincer move in our Configuration 2 of Main Implementation

Upon evaluating the screenshots that were generated through testing, we noticed that across all our testing implementations our agent was frequently losing when a pincer move was formed. We noticed that this was occurring mostly within the corners of the maze, which is similar to what (Tong and Sung, 2010) stated when they introduced the idea of a danger map for dangerous parts of the maze. We believe that this could be in conflict with the aforementioned Ambush strategy that we aimed to implement. The flaw in the Ambush strategy is that the agent will remain stationary within the corner of the maze until a ghost is close. This could suggest that using a pure Ambush strategy without any kind of foresight might cause problems for the agent should we pursue this strategy feature. We could implement a more in depth Ambush algorithm by taking into consideration the fact that at least two of the ghost move in a similar manner (Red and Pink). In addition, we may have to use a more effective means of allowing our agent to retreat from a dangerous situation. Currently it makes use of the Flee state that simply states that the agent should move in the opposite direction to the ghost. This means that eventually the agent could end up in a no win situation if we fail to at least predict what could happen should the agent retreat down a certain path.

Should we proceed to use MCTS in our future work, we most certainly will have to take into account the possibility of this occurring again and what effective ways could be utilized with our current methodology to counter-act it. In addition, we may have to use a more effective means of allowing our agent to retreat from a dangerous situation. Currently it makes use of the *Flee* state that simply states that the agent should move in the opposite direction to the ghost. This means that eventually the agent could end up in an awkward situation if we fail to at least predict what could happen should the agent retreat down a certain path.

7.2 Re-evaluating our tools

Revisiting the problems that were described within the literature review, the simulator most definitely offered a large sense of freedom and flexibility. This was demonstrated when it came to enabling us to modify the values that we required to change such as temporarily changing the behaviour of the ghosts. Furthermore it also meant that we had a fully configurable test bed for testing the MCTS algorithm with. The main issue that we faced however, when it came to the implementation of the MCTS algorithm with our own simulator was the inability to accurately render the screen while expensive computations were being done during each tick. While we adjusted settings for the tree growth and evaluation within our tests to determine where the CPU bottleneck was, we noticed that there were no real plausible differences regardless of the changes that we made. We hope that in the near future should we approach this method again that we would perhaps develop a simulator that utilises a graphics rendering API that places the work load onto the GPU such as OpenGL or DirectX.

The reason we believe that this would be a suitable idea would be due to the way that the current implementation works now. Our current simulator uses the likes of GDI+ and WinForms which in most instances utilises the CPU for its graphical processing (Microsoft, 2012). As mentioned within section 2.1, we detailed the costly nature of the MCTS algorithm on the CPU and therefore combining with additional strain of rendering would explain the current issues we are having. This would include the visual artefacts such as inconsistent tick rates in the rendering of the graphical buffer that we were experiencing. This meant that once MCTS calculations had completed, any instructions on the call stack for rendering the simulator to the screen would be completed all at once. Presuming that we wish to remain with the functionality of the C# programming language, we could pursue the possibility of using the now deprecated by functional XNA game framework, or a more modern and managed interface for DirectX such as SharpDX (Mutel, 2010).

7.3 Usage of MCTS with other games

As we can see from our research, the application of MCTS is very much ideal to the game of Ms. Pac-Man as the future moves can be consolidated to the AI having to choose between 4 different directions in which the agent could move in. Likewise, this has been previously demonstrated with games such as *Go* where equally there is a high branching factor of decisions that must be evaluated rapidly at each tick (Browne et al., 2012). Our concern with the usage of the algorithm in other games is that the game state would have to be discretized in a way that would be appropriate for MCTS simulations. For instance, if we were to

consider a game based in an open world and 3D environment, there would be substantially more data to consider and it would be harder to determine what parts of the game state we would simulate. No longer is the agent then required to move in simply 4 directions but instead depending on the game, the agent may have to consider a variety of other factors that would have to be simulated in order to make appropriate decisions at each tick. Fortunately within the game of Ms. Pac-Man the process of doing this is rather straight forward as we can consider each junction within the maze a point of decision for the agent. This then depends on how appropriately the environment would be discretized so that once simulation is completed, the outputted results become dependable.

7.4 Closing Statement

Referring back to section 1.1 of our aims and objectives, we can say with certainty that we have successfully applied the Monte-Carlo Tree Search algorithm with the strategy of a finite state machine. Having investigated the current literature surrounding the topic of MCTS, it's evident that there are large advancements in the area, with countless methods of how the algorithm can be applied. However, while demonstrating positive results through our testing, we unfortunately can still conclude that it does not output the same level of performance as that of an agent which uses purely scripted behaviour. Through the implementation of the agent we recognized that while responding appropriately to the ghosts in the maze, there were certain instances in which it would fail to detect danger down C-Paths adjacent to its position.

We are lead to believe at this stage that it is down to several reasons. The first being the behaviour of the ghosts cannot be depended upon during the simulation of the tree due to their stochastic nature as stated before in section 1. Regardless of how many times the tree can be simulated to conclude the reward of a given path in the maze, there is always the possibility that the ghosts will behave differently within the current game state.

The MCTS algorithm is perhaps still a viable method should it be applied to our agent differently. Referring to Fig. 23, the reason that our main implementation was capable of eating ghosts at all was because it was not depending on the path generation produced by the MCTS. In our opinion, it would be better to make use of the approximations generated by MCTS as a supplement to our agents understanding of the environment rather than a direct form of navigation.

8. Bibliography & References

Auer, P., Cesa-Bianchi, N., Fischer, P., 2002. Finite-time Analysis of the Multiarmed Bandit Problem. Mach. Learn. 47, 235–256.

Baier, H., Drake, P.D., Dec. The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. IEEE Transactions on Computational Intelligence and AI in Games 2, 303–309.

Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S., 2012. A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games 4, 1 –43.

De Bonet, J., 2006. Learning to Play Pacman using Incremental Reinforcement Learning [WWW Document]. Learning to Play Pacman using Incremental Reinforcement Learning. URL http://www.debonet.com/Research/Learning/PacMan/

Fitzgerald, A., Congdon, C.B., 2009. RAMP: A rule-based agent for Ms. Pac-Man, in: IEEE Congress on Evolutionary Computation, 2009. CEC '09. Presented at the IEEE Congress on Evolutionary Computation, 2009. CEC '09, pp. 2646 –2653.

Flensbank, J., Yannakakis, G., 2008. Ms. Pacman Competition [WWW Document]. Ms. Pacman Competition. URL http://mspacmanai.codeplex.com/releases/view/15706

Gallagher, M., Ledwich, M., 2007. Evolving Pac-Man Players: Can We Learn from Raw Input?, in: IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007. Presented at the IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007, pp. 282 –287.

Gallagher, M., Ryan, A., 2003. Learning to play Pac-Man: an evolutionary, rule-based approach, in: The 2003 Congress on Evolutionary Computation, 2003. CEC '03. Presented at the The 2003 Congress on Evolutionary Computation, 2003. CEC '03, pp. 2462 – 2469 Vol.4.

Galván-López, E., Swafford, J.M., O'Neill, M., Brabazon, A., 2010. Evolving a Ms. PacMan Controller Using Grammatical Evolution, in: Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.-K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (Eds.), Applications of Evolutionary Computation. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 161–170.

Handa, H., Isozaki, M., 2008. Evolutionary fuzzy systems for generating better Ms.PacMan players, in: IEEE International Conference on Fuzzy Systems, 2008. FUZZ-IEEE 2008. (IEEE World Congress on Computational Intelligence). Presented at the IEEE International Conference on Fuzzy Systems, 2008. FUZZ-IEEE 2008. (IEEE World Congress on Computational Intelligence), pp. 2182 –2185.

Ikehata, N., Ito, T., 2011. Monte-Carlo Tree Search in Ms. Pac-Man, in: Computational Intelligence and Games (CIG), 2011 IEEE Conference On. pp. 39–46.

James, N.-K., n.d. Json.NET [WWW Document]. Json.NET. URL http://james.newtonking.com/pages/json-net.aspx

Koza, J.R., 1992. Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA, USA.

Krause, E.F., 1987. Taxicab Geometry: An Adventure in Non-Euclidean Geometry. Dover Publications.

Microsoft, 2012. Comparing Direct2D and GDI Hardware Acceleration [WWW Document]. Comparing Direct2D and GDI Hardware Acceleration. URL http://msdn.microsoft.com/en-gb/library/windows/desktop/ff729480(v=vs.85).aspx#gdi_and_direct2d_hardware_acceleration

Mutel, A., 2010. SharpDX [WWW Document]. SharpDX. URL http://sharpdx.org (accessed 3.5.13).

Pepels, T., Winands, M.H.M., 2012. Enhancements for Monte-Carlo Tree Search in Ms Pac-Man, in: 2012 IEEE Conference on Computational Intelligence and Games (CIG). Presented at the 2012 IEEE Conference on Computational Intelligence and Games (CIG), pp. 265 –272.

Pittman, P., 2011. The Pac-Man Dossier [WWW Document]. The Pac-Man Dossier. URL http://home.comcast.net/~jpittman2/pacman/pacmandossier.html#CH4_Blinky (accessed 4.21.13).

Robles, D., Lucas, S.M., 2009. A simple tree search method for playing Ms. Pac-Man, in: IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009. Presented at the IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009, pp. 249 –255.

Samothrakis, S., Robles, D., Lucas, S., 2011. Fast Approximate Max-n Monte Carlo Tree Search for Ms Pac-Man. IEEE Transactions on Computational Intelligence and AI in Games 3, 142 –154.

Szita, I., Lõrincz, A., 2007. Learning to play using low-complexity rule-based policies: illustrations through Ms. Pac-Man. J. Artif. Int. Res. 30, 659–684.

Thawonmas, R., Ashida, T., 2010. Evolution strategy for optimizing parameters in Ms Pac-Man controller ICE Pambush 3, in: 2010 IEEE Symposium on Computational Intelligence and Games (CIG). Presented at the 2010 IEEE Symposium on Computational Intelligence and Games (CIG), pp. 235 –240.

Thompson, T., McMillan, L., Levine, J., Andrew, A., 2008. An evaluation of the benefits of look-ahead in Pac-Man, in: Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On. Presented at the Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On, pp. 310 –315.

Tong, B.K.-B., Ma, C.M., Sung, C.W., 2011. A Monte-Carlo approach for the endgame of Ms. Pac-Man, in: 2011 IEEE Conference on Computational Intelligence and Games (CIG). Presented at the 2011 IEEE Conference on Computational Intelligence and Games (CIG), pp. 9 –15.

Tong, B.K.B., Sung, C.W., 2010. A Monte-Carlo approach for ghost avoidance in the Ms. Pac-Man game, in: Games Innovations Conference (ICE-GIC), 2010 International IEEE Consumer Electronics Society's. Presented at the Games Innovations Conference (ICE-GIC), 2010 International IEEE Consumer Electronics Society's, pp. 1 –8.

9. Appendices