Ms. Pac-Man vs Ghost Team - A Monte Carlo Tree Search Approach

Miguel de Carvalho Maximiano
miguelcmaxi@hotmail.com

Instituto Superior Técnico, Lisboa, Portugal

November 2019

Abstract

This thesis tackles the problem presented by the *Ms. Pac-Man vs Ghost Team Competition*. This competition challenges developers around the world to create intelligent agents with the objective of getting the best score possible in the game *Ms. Pac-Man* while dealing with partial observability of the game environment. The approach taken was a Monte Carlo Tree Search algorithm. In the end, two similar agents were developed, tested and compared. Both achieved satisfactory results, reaching level 2 of the game, and provided great insight into the difficulty of the challenge and possible solutions for future approaches.

Keywords: Monte Carlo tree search; Artificial intelligence; Intelligent systems; Ms. Pac-Man vs ghost team.

1. Introduction

In recent years, Artificial Intelligence (AI) has become the most discussed field in computer science. With the advances of technology and rising popularity of smart devices, tech companies have focused more and more resources in the development of products that make use of intelligent agents in different ways. Names like Amazon's Echo and Alexa, Google's Cortana or Apple's Siri are recognized by basically anyone. These products were heavily marketed when they first came out as innovative, smart assistants to everyday tasks and their success was instant. Therefore, it is not surprising that these giants of the tech world would continue to invest in them for years to come. In the case of *Ms. Pac-Man*, several competitions have been held over the last decades, challenging participants to develop agents that can optimize the playing of the game. Even so, new entries obtain interesting results every year.

2. Background

This section serves to integrate both the *Ms. Pac-Man vs Ghost Team Competition* and the MCTS algorithm in the context of this thesis.

2.1. Ms. Pac-Man vs Ghost Team Competition

The *Ms. Pac-Man vs Ghost Team Competition* [3] is a competition organized by the University of Essex since 2016. It challenges developers to produces intelligent agents that can control either Ms. Pac-Man or the ghost team in the game *Ms. Pac-Man*. Before 2016, the university organized similar competitions, named *Ms. Pac-Man Screen Capture Competition* [5] and *Ms. Pac-Man vs Ghosts Competition* [2], which had the same objective of challenging developers but different formats. These competitions are well-known for the good results obtained by the participants in terms of novel intelligent agents and continue to be relevant for research in the field of AI.

2.2. Monte Carlo Tree Search

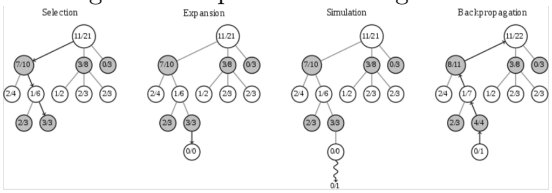
The *Monte Carlo Tree Search (MCTS)* is a heuristic search algorithm for decision processes in the context of computational problems. It is based on the original Monte Carlo method, applying it to a game-tree search, as described by Rèmi Coulon in 2006 [4].

This algorithm is commonly used in decision problems in games. Its first implementation was in the game Go [1], but it has been used in other board games like chess, games with incomplete information like poker and, more recently, in video games.

The MCTS algorithm focuses on analysing the most promising moves for a player at a certain moment in time. By utilizing random sampling to expand the search tree, the algorithm is able to playout the game many times and, considering the final result of each playout, re-weight the tree nodes so that, in future playouts, better nodes are chosen

more often. The algorithm itself consists of 4 steps: Selection, Expansion, Simulation and Backpropagation. These steps are illustrated in the following figure:

Figure 1: Steps of MCTS Algorithm



3. Solution

This section will take a close look at the agent developed and all the code necessary to make it work.

3.1. Overview of the Agent

The agent is composed of 2 classes of objects: a **MyPacMan** class a **Node** class that models tree nodes necessary for the implementation of the MCTS algorithm.

The **MyPacMan** object is composed of 7 attributes and 3 methods, including a very simple constructor.

Figure 2: MyPacMan

```
public class MyPacMan extends PacmanController {

    private Maze currentMaze;
    private GhostPredictionsFast predictions;
    private PillModel pillModel;
    private int[] ghostEdibleTime;
    protected Game mostRecentGame;
    protected int maxTreeDepth = 30;
    protected int maxPlayoutDepth = 250;
```

Attributes *maxTreeDepth* and *maxPlayoutDepth* are initialized manually because they are values chosen *a priori*. *Maze*, *GhostPredictionsFast*, *PillModel* and *Game* are all classes from the game engine. These are initialized, modified and used in the *getMove* method at each time step. The constructor simply initializes the array *ghostEdibleTime* with size equal to the number of ghosts in the game.

The *obtainDeterminisedState* method is the method that makes it possible to run a basic MCTS algorithm over a partially observable environment, by creating a copy of the real game state. It populates the points of the maze that the agent does not have access to based on probability: places pills where pills are supposed to be and, for ghosts unaccounted for, it assumes that the ghosts maintain the direction they were last seen taking. This method is used in the *getMove* method, which contains all the logic behind the agent's behavior. The *getMove* algorithm is as follows:

Figure 3: getMove Algorithm

```
if currentMaze value equals maze from game object then
    Do nothing;
else
    Initialize currentMaze with maze value from game object;
    Initialize predictor and pillModel as null;
    Initialize ghostEdibleTime array with default values of -1;
end
Update attributes predictor and pillModel using information from game object;
Run MCTS algorithm;
Update predictor with new observations;
Select and return best move;
```

The **Node** object is composed of 9 attributes and 15 methods, including 2 constructors.

Figure 4: Node

```
class Node {

    private final MyPacMan MyPacMan;
    private Node parent;
    private MOVE prevMove;
    private MOVE[] legalMoves;
    private Node[] children;
    private int expandedChildren;
    private int visits;
    private double score;
    private int treeDepth;
```

Most of the attributes are simple to understand just by looking at their names. *MyPacMan* is the object that represents the agent; *parent*, *children*, *expandedChildren*, *visits* and *treeDepth* are common attributes in tree-search algorithms, they contain essential information for MCTS. *score* is also a very important attribute for the decision making algorithm because it is this value that will determine the weight of each node considered by the selection step of MCTS.

The unique attributes for this context are *prevMove* and *legalMoves*. The first one is the last move made by the agent and it is necessary to advance the game state during the playout step of the MCTS algorithm. The latter saves all the moves the agent can make in the next time step of the game. It is necessary not only to advance the game state during the expansion step of the MCTS algorithm but also to make sure the *Move* returned to and by the *getMove* method in *MyPacMan* is valid, since the agent can't walk into walls.

The constructors are quite simple, they are used to initiallize attributes that we know the values of, taking into consideration if we want to create a root node or child node.

There are 2 methods called *selectBestMove*. The first one was used in the initial stage of development of the agent, but was later replaced by the second because the latter achieved better results overall. Both of these methods will be explained in the following subsections of this section and results will be discussed in the next section of this document.

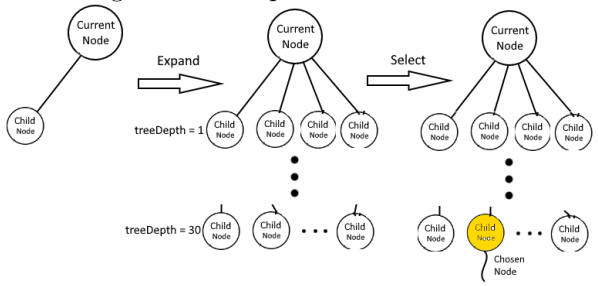
3.2. The MCTS Algorithm

Our MCTS algorithm is the made up of 3 methods: *select_expand*, *playout* and *backPropagate*. In general, this algorithm is very close to the standard MCTS algorithm. The game state is modeled by the class **Game**, which is part of the package obtained from the competition organizers. This class is quite complex, so we used it throughout the project as a *black box* and get everything we need from it via its public methods or actual objects of that class returned by the *obtainDeterminisedState* method.

Firstly, the **Selection** and **Expansion** steps are combined in a single method. In fact, there isn't a true Selection step. The algorithm is ran, at each time step, on a copy of the game state generated by the method *obtainDeterminisedState*, with the current game state being the root node of the decision tree. As such, during the Selection step of the algorithm, the only node that exists is the root node, meaning the algorithm always selects that one. However, the expansion step generates all child nodes of this root node and its child nodes, recurrently, until nodes with treeDepth value equal to maxTreeDepth are reached, and then selects one of those child nodes, the one with highest calculated score value, so one can say that there is indeed a selection process. After this process, the game state is advanced, which means that this child node becomes the root node of the decision tree used from now on. One small detail to be noticed is that this MCTS algorithm does not implement **UCB/UCT** in its **Selection** step because there is no *win state* in Ms. Pac-Man, since as long as the agent lives, the game goes on. As such, it didn't make sense to use such a formula in the agent's algorithm.

Here is an example of a possible decision tree created during this step of the algorithm through expansion:

Figure 5: Example of a decision tree



Next, the **Playout** step is executed. This step is very standard: random moves are made until the tree depth reaches the maximum playout depth, which is defined in the *MyPacMan* object. Then, the score for the game state reached is calculated and returned. The score mentioned is calculated by the *calculateGameScore* method, and takes into consideration not only the in-game score but also the game time and level reached.

Finally, the **BackPropagate** step: it simply updates the *score* values of the parent nodes, recursively, until reaching the root node of the search tree. These values work as the weights of the nodes for all purposes.

3.3. The Adaptation

As mentioned earlier, we have 2 methods called *selectBestMove*. This method is ran in the *getMove* method after the MCTS algorithm to, like the name suggests, calculate the best move the agent should make at that time step. The first iteration of the method provides a very simple solution: If there is a best child, keep moving in the same direction. The logic behind this is that, if this child is the best one, it should continue doing what it was doing before. Overall, this works. The agent tends to run around the maze looking for pills and super pills. The problem with this approach is that, when faced with a ghost in its path, the agent would almost never be able to avoid death because moving backwards was not considered a legal move. Simply making it a legal move would result in the agent running back and fourth in the same place, so it was not a good way to fix it. This was the main road block during the development of this agent. To move past it, we had to move slightly away from the MCTS based solution, so we adapted the *selectBestMove* method to deal with this problem and be more successful in this specific game environment.

So, instead of instantly returning the child's previous move like in the first iteration of this method, we instead call the *getNextMove* method. This method forces the agent to run away from an inedible ghost when moving towards one.

As the next section will demonstrate, this adaptation worked very well and the agent's performance improved overall.

4. Results

Here we will look at how the agent was tested and the results obtained.

4.1. Evaluation Methodology

To evaluate the agent, we used the *runExperiment* method in the **Main** file. This method belongs to the *executor* class and is very useful to test the agent's performance because it runs the game multiple times in one execution, which facilitates batch

testing. It also doesn't run a graphic interface, so it is less computationally intense, but on the other hand provides the developer with important stats for evaluating the agent's performance, specifically **average score** over the games played in the batch, **minimum** and **maximum score**, **standard deviation** and **standard error**.

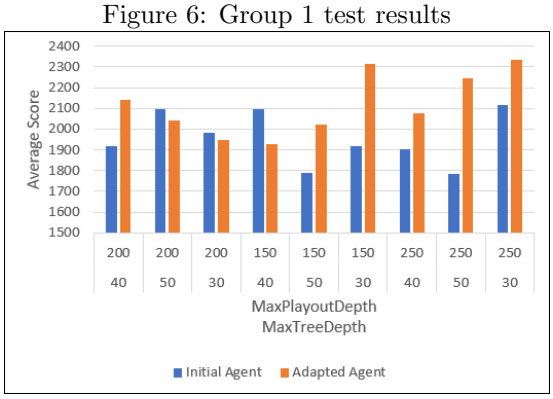
Using these values, we calculated the **95% Confidence Interval**, which could be useful to compare versions of the agent, but these intervals overlap in almost all circumstances.

The batches can be divided in 2 groups. The first group aimed at finding the ideal values for the *maxTreeDepth* and *maxPlayoutDepth* attributes of the *MyPacMan* object. The second group aimed at optimizing the *calculateGameScore* method. This method uses *in-game score*, *game time* and *level* to calculate the value used as weight for the nodes of the search tree, making it essential for the decision making process.

4.2. Results

Firstly we will take a broad look at the results of group 1 tests, which aimed to find the ideal *maxTreeDepth* and *maxPlayoutDepth* attributes for the agent.

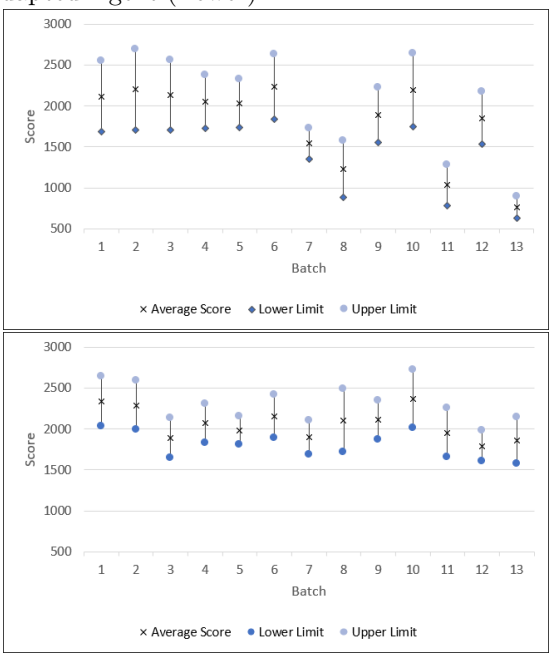
Figure 6: Group 1 test results



As we can see in figure 6, for both iterations of the agent, the best combination for (*maxTreeDepth*, *maxPlayoutDepth*), in terms of average score, was (30,250). Confidence intervals aren't included in this figure because they were not useful; they overlapped too much to provide insight on which combination of the 2 attributes was ideal. Taking this into consideration, the group 2 tests were run using these values for (*maxTreeDepth*, *maxPlayoutDepth*).

The group 2 batches tested various weights for the factors in the *calculateGameScore* method's formula, which are by default 1 for *in-game score* and *game time*, and 1000 for *level*. As we can see in figure 7, the adapted agent performs better in almost every test batch and achieves a higher average score in batch 10 than the initial agent in any of its batches. This is due to the adaptation made

Figure 7: Group 2 test results: Initial (Upper) Vs Adapted Agent (Lower)



to the agent's behavior: unlike the initial agent, the adapted agent actively runs away from inedible ghosts, resulting in longer game times and, thus, overall better scores.

Batches **1** to **6** and **10** experiment with various weights for the *in-game score*, *game time* and *level* factors of the *calculateGameScore* method. Overall, there seems to be low variance in terms of results from these batches, as the 95% confidence intervals overlap in a big portion of data. The initial agent obtains the best result by reducing the weight of *game time* by half and increasing the weight of *level* to double, while the adapted agent performs best when multiplying the weight of *in-game score* by 50 and the weight of *level* by 100. In these batches, both agents were able to reach level 2 in the game at least once per batch, but the adapted agent reached this level more often. The remaining batches are more interesting.

Batches **7**, **8** and **9** experiment with making each of the factors of the *calculateGameScore* method's formula obsolete, one at a time. To do this, **batch 7** sets the weight of *in-game score* to 0, **batch 8** sets the weight of *game time* to 0 and **batch 9** sets the weight of *level* to 0. For the initial agent, we can see that disregarding one of the factors results in worse performance. In particular, disregarding *game time* significantly hinders the agent's performance. In a way, the *game time* factor represents how much the agent prioritizes survival over immediate score. By removing this factor, the agent tends to risk too much, thus dying faster and ob-

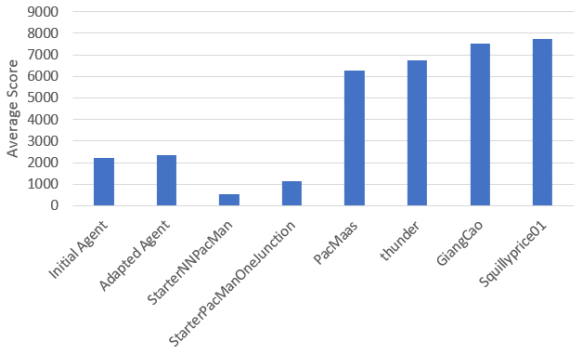
taining lower scores overall. On the other hand, the adapted agent doesn't seem to be affected by using only 2 out of the 3 factors, obtaining results in line with the ones obtained before. This is because these weights don't affect the adapted portion of the decision making process. This portion helps the agent avoid ghosts no matter what, so even when the weight of *game time* is set to 0 and the agent risks as much as it can, it still runs away from ghosts when facing them. The initial agent was able to reach level 2 in batches **7** and **9**, but not batch **8**, while the adapted agent managed to reach this level at least once in each batch.

Batches **11**, **12** and **13** go a step further, and set 2 out of 3 factors to 0: **batch 11** sets the weight of *game time* and *level* to 0, **batch 12** sets the weight of *in-game score* and *level* to 0 and **batch 13** sets the weight of *in-game score* and *game time* to 0. In other words, the agent from **batch 11** only cares about the *in-game score* so it tries to maximize the number of ghosts eaten per power pill eaten; the agent from **batch 12** only cares about the *game time* so it tries to survive above all; and the agent from **batch 13** only cares about the *level* so it focuses on collecting all the pills and proceeding to the next level. As we can see in figure 7, there is a dip in performance for the initial agent when focusing solely on *in-game score* or *level*, while focusing simply on *game time* doesn't hinder its performance, once again because it tends to risk too much in the first two cases. However, focusing solely on surviving for as long as possible seems to be a valid strategy, which makes sense considering that **Ms. Pac-Man** is an *infinite game*: if the agent never dies, the game never ends, thus the score keeps increasing indefinitely. For the adapted agent, focusing on a one-dimensional strategy results in average scores slightly lower. The initial agent managed to reach level 2 only in batch **12**, while the adapted agent managed to do the same in every batch.

To further evaluate the agent, it is possible to compare its performance to the performance of some of the basic agents from the competition's organizers and submissions from other researchers. The organizers use these basic agents as a basis for comparison during the competition itself, but only reveal the average score achieved by the agents. The results from 2018 are presented in the figure 8.

The StarterNNPacMan and StarterPacManOneJunction are part of the basic agents provided by the competition's organizers, the Initial and Adapted Agent are the ones developed in this thesis and the remaining are submissions made to the competition. It is clear that the agents from this thesis are far from the submissions to the competition in terms of performance. However, they perform much better than the basic agents from the competition. Since that was the main objective of this thesis, it can be considered a success.

Figure 8: Comparison to basic agents from 2018



5. Conclusions

All in all, we were able to obtain satisfactory results. The agent was able to reach level 2 several times and almost reached level 3 a couple of times. Nonetheless, we were able to improve on the basic agents provided by the competition and retrieve interesting information about different strategies. The values of *maxTreeDepth* and *maxPlayoutDepth* seem to not have such a big influence on the performance of the agent as initially thought, since there was found no direct correlation between the varying values tested for these attributes and the results. Additionally, the tests run with different weights for the factors of the *calculateGameScore* method formula showed the value of different strategies. Strategies that focused solely on maximizing score are slightly less valuable than strategies that tried to balance maximizing score with surviving. This means that adaptable strategies are better for agents in partially observable environments, as stricter strategies inhibit the agent's ability to adapt during gameplay time.

In the future, there is definitely room for improvement. The MCTS algorithm developed for this thesis is particularly bad at dealing with the partial observability factor of the competition's game environment and the randomness associated with the movement of the ghosts. These two aspects combined make it so many of the decisions taken by the agent are based on little information and hinder its performance massively. Therefore, better prediction algorithms could be useful in order to improve this agent.

Acknowledgements

I would like to thank my parents for making this possible by providing me with the best education possible throughout my life and always encouraging me to work hard. I would also like to thank every other member of my family for making me who i

am today

I would also like to acknowledge my dissertation supervisor Prof. Alberto Sardinha for his help throughout this Thesis.

Last but not least, a big thank you to all my friends and colleagues that helped me throughout this journey and always supported me. A special thank you to my girlfriend for making sure I stayed focused for the last 8 months.

Thank you to everyone.

References

[1] David Silver, Aja Huang, Chris Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel. Mastering the game of Go with deep neural networks and tree search. 2016.

[2] Philipp Rohlfshagen, Simon M. Lucas. Ms Pac-Man versus Ghost Team CEC 2011 Competition.

[3] Piers R. Williams, Diego Perez-Liebana and Simon M. Lucas. Ms. Pac-Man Versus Ghost Team CIG 2016 Competition.

[4] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, 2006.

[5] Simon M. Lucas. Screen-capture Ms Pac-Man. 2009.