TÉCNICO LISBOA

Ms.Pac-Man vs Ghost Team

A Monte Carlo Tree Search Approach

Miguel de Carvalho Maximiano

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha
Member of the Committee: Prof. Manuel Fernando Cabido Peres Lopes

November 2019

Acknowledgments

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.

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.

Resumo

Esta tese aborda o problema apresentado pela competição *Ms. Pac-Man vs Ghost Team Competition*. Esta competição desafia *developers* em todo o mundo a criarem agentes inteligentes com o objetivo de obter a melhor pontuação possível no jogo *Ms. Pac-Man* lidando com a observabilidade parcial do ambiente de jogo. A abordagem escolhida foi um algoritmo Monte Carlo Tree Search. No fim, foram desenvolvidos, testados e comparados dois agentes semelhantes. Ambos alcançaram resultados satisfatórios, atingindo o nível 2 do jogo, e providenciaram melhor compreensão sobre a dificuldade do desafio e possíveis soluções para abordagens futuras.

Palavras Chave

Monte Carlo Tree Search; Inteligência Artificial; Sistemas Inteligentes; Ms. Pac-Man Vs Ghost Team.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

1

Introduction

Contents

1.1 Motivation

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. However, research in the field of AI is not a recent trend. In fact, after surging in the 1950s, this field has seen a rapid evolution over the last couple of decades, with a focus on games as a test-bed for research efforts. Games are one of the main ways to develop AI, in particular intelligent agents, because they are defined by a strict set of rules, which means the testing environment is more easily understood by the developers and, consequently, the agents being developed, and they usually provide the agent with a unambiguous way to evaluate its performance via game score achieved or outcome. In 1997, DeepBlue [1] was able to beat, in a game of chess, Garry Kasparov, the world champion at the time. More recently, in 2016, AlphaGo [2] beat Lee Sedol, considered the best Go player in the history of the game, in a match of Go. These two softwares achieved what are now considered milestones in the field of AI.

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. In fact, *Ms. Pac-Man* is a much more complex game than it seems at first sight: the ghost's movement is semi-random and unique for each ghost, there are multiple maze layouts that, after level 4, alternate randomly, the game's speed increases with each level and, above all, the game doesn't end until Ms. Pac-Man dies, which means that "infinite" playthroughs are, theoretically, possible.

1.2 Goals

This thesis' main goal was to develop an agent suited for the international *Ms. Pac-Man vs Ghost Team Competition* [3]. Using the competition's engine and development environment, this thesis focuses on a Monte Carlo Tree Search (MCTS) based approach to try to achieve the highest in-game score possible. During development, we ended up with 2 variations of an agent, thus enabling the comparing of a "standard" MCTS aproach to a more complex one.

1.3 Organization of the Document

In Chapter 2 of this document, a short introduction to the background of the *Ms. Pac-Man vs Ghost Team Competition* is provided, along with an explanation of the Monte Carlo Tree Search standard algorithm and a look at 2 papers that were essential in the conceiving of the agent developed in this thesis.

In Chapter 3, we will look at the development environment used for this thesis, consisting of the package provided by the *Ms. Pac-Man vs Ghost Team Competition* organizers, and the initial development process of the agent.

In Chapter 4, everything about the agent is explained in detail, from the model of the game and agent itself to the MCTS algorithm and the adaptation we made to improve the agent's performance.

In Chapter 5, the results achieved by the agent are presented and discussed.

In Chapter 6, conclusions are drawn and we talk about how this agent could be worked on in the future and its limitations.

2

Background

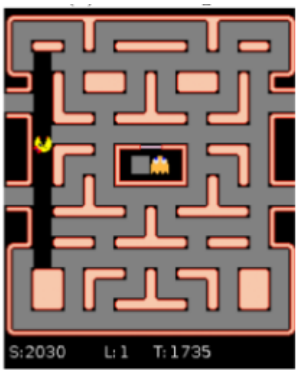
Contents

This chapter 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* [4] and *Ms. Pac-Man vs Ghosts Competition* [5], which had the same objective of challenging developers but different formats. The previous one provided agents with data about the game state via screenshots, while the latter one did so using the game engine itself. The difference between the latter and the current format is that the more recent one, running since 2016, implements partial observability for the agent as an element of increased difficulty. This means that the agents can't "see" through walls and, thus, have access to limited information about the game state at any point.



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.

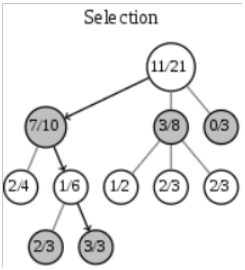
2.2 Monte Carlo Tree Search

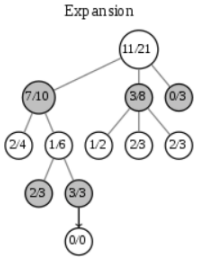
The *Monte Carlo Tree Search (MCTS)* is a heuristic search algorithm for decision processes in the context of computational problems. It is based on the original Monte Carlo method, applying it to a game-tree search, as described by Rèmi Coulon in 2006 [6].

This algorithm is commonly used in decision problems in games. Its first implementation was in the game Go [2], 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:

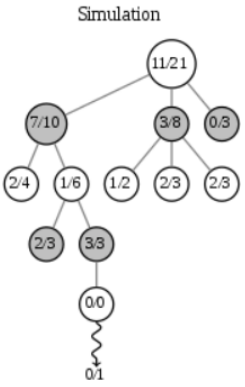
Selection: The first step in the algorithm. In this step, starting from a root node, successive child nodes are selected until a leaf node is reached. The root node is the current game state, while leaf nodes are any nodes that haven't gone through a playout. Usually, the leaf node selected is the one with highest win rate at this present time, as shown by the image, but other heuristics can be used to describe the selection process.

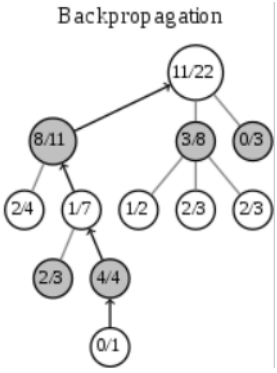




Expansion: After selecting a leaf node, let's call it node L, one or more child nodes are created. These child nodes are simply game states reachable from L after one step in game time. After creating this child nodes, one of them is chosen to be played out in the next step of the algorithm. It is common practice to create all the possible child nodes in order to make sure that all possible outcomes are evaluated by the algorithm, resulting in better results overall.

Simulation: Also known as the **Playout** or **Rollout** phase, this step consists of doing one complete random playout from the child node chosen in the Expansion step. This means choosing moves until the algorithm reaches a game state where the game is decided, i.e. the game is won, lost or draw. However, some games might be too long or complex or simply not have a clear game ending state (for example, Ms. Pac-Man if the player never loses) for this basic approach to work because the tree would never stop growing and the algorithm would never progress beyond this step. As such, most implementations of MCTS limit the number of moves executed in a simple playout. This is called the maximum playout depth. In this case, the algorithm proceeds to the next step after reaching the maximum playout depth **or** a game ending state.





Backpropagation: The final step of the algorithm, simply consists of updating the information of the nodes from the child node played to the root node. Usually this means simply updating the weights of the nodes but in more complex implementations each node might store more relevant information.

These 4 steps are repeated in each time step of the game, with the root node changing each time. This guarantees that, ideally, the algorithm makes the best move in each time step and adapts to changes in the game environment *on the fly*

Selecting child nodes is the hardest part of the algorithm to implement, because it requires a good balance between *exploration* and *exploitation*. Usually, there is limited time to make a decision. Thus, it is important to allocate the time available wisely between simulations of several moves with low playout steps (*exploration*) and simulating a lot of playout steps for a more restricted number of initial moves (*exploitation*). To deal with this problem in a systematic way, the most common formulas used are UCB and UCT, both based on win-rate calculations to help the algorithm make the best decision.

3

Related Work

Contents

In this chapter, we will look at some agents submitted to the competition in previous years. These will provide a deeper insight into different approaches developers have taken over time and what results they achieved.

3.1 Training Pac-Man bots using Reinforcement Learning and Case-based Reasoning

[7]

This project aimed to create bots for the *Ms. Pac-Man* game, particularly for the Ms. Pac-Man character. For this, the authors implemented a reinforcement learning technique with Q-Learning algorithm for decision making of the agent, replacing the typical Q-table with a case base, which helped deal with rich game state representation and inject domain knowledge.

This project was developed on a game engine that did not implement partial observability, so the agent had complete access to game state. This game state was modeled as a graph where each tile of the game space was represented by a node. For each direction that the agent can move, it saves 4 variables that represent distance to closest pill, closest power-pill, closest non-edible ghost and closest edible ghost, which means it needs 16 variables to describe the game state. To reduce the amount of possible game states, these variables can have 1 of 5 values: WALL, VERY CLOSE, CLOSE, MEDIUM, FAR.

The agent can choose one of the possible four directions to move, but the decision process only takes place in intersections or when a ghost is nearby. To choose which direction to take, the agent uses a similarity function to find the case stored most similar and makes the same decision. After each decision, the new case is stored and the case base is *cleaned* by removing cases too similar to each other.

This approach is rather different from the one developed in this thesis. However, it is mentioned here because it was crucial for the success of our agent, as this is where the idea of *adaptation* mentioned further down in this document came from. If the agent developed by these participants only ran the decision process on intersections, it would not have a way to avoid ghosts, just like our agent wouldn't be able to avoid ghosts if we hadn't worked around the impossibility of the agent to choose to turn backwards.

3.2 A Model-Based Approach to Optimizing Ms. Pac-Man Game Strategies in Real Time

[8]

This project uses a model-based approach to create an AI player for the character Ms. Pac-Man. The game model is represented in the form of a graph which is updated over time and, like in the previous example, each node represents one position on the maze. However, the game model is different. In this case, walls aren't part of the model. Instead, each node represents one cell and the adjacency of cells in a maze is represented by edges connecting nodes. Based on this graph, a decision tree is created, with root in the position of Ms. Pac-Man and each branch representing a sequence of cells that can be visited by Ms. Pac-Man.

To choose a branch to explore, a profit function was made for Ms. Pac-Man. This function takes into account the risk of getting captured by a non-edible ghost and the reward from reaching a pill, power pill, edible ghost or bonus item. This function is used to calculate the overall reward value of a branch, which is then used to choose the next action. In this process, the agent uses an exponential discount factor, thus prioritizing short term reward over long term because the game state changes so fast.

This agent achieved good results: it performs better than beginner and intermediate human players. This showed the value of a well-thought profit function. In our project, we implement something like this in a method called *calculateGameScore*, which takes into account the game score, game time and level of a game state to calculate the value of a branch in the decision process.

3.3 A Simple Tree Search Method for Playing Ms. Pac-Man

[9]

This paper describes a tree-search software agent. Game-tree search is very commonly used in many games, but this was the first attempt to apply it to Ms. Pac-Man. It was utilized for path finding and tested in a simulator as well as the original game via screen capture.

For the purpose of testing this agent, the authors of this paper created a Ms. Pac-Man simulator. This simulator, written in Java, uses the Model-View-Controller design pattern and replicates most of the features of the original game, albeit with some differences. This also allowed much faster and deep evaluation of the agent's performance. They also created a *screen capture adapter*, which by itself extracts all the information from the game screen, thus allowing any agent to play in the simulator and original game without any specific changes to its code.

The agent itself uses a tree to evaluate all the possible paths that can be taken from Ms. Pac-Man's current position. This tree is generated every game cycle, using Ms. Pac-Man's current position node as

the root and adding children to the tree until the maximum depth, manually chosen by the developers, is reached. Each node contains information about that position on the maze such as empty, pill, power pill, Ms. Pac-Man, ghost or edible ghost. After the tree is created, a tree search algorithm is run to find out all the possible paths that can be taken. Then, each path is evaluated as safe or not-safe. If a path has no ghosts and there are no ghosts in the tree, it is considered safe, if there is a ghost in the path it is immediately considerate unsafe. For cases in which the path being evaluated has no ghosts but there are ghosts elsewhere in the tree, the algorithm sees if it can reach the first safe node before any ghost. If yes, the path then the path is considered safe, otherwise it is considered not-safe. After evaluating all paths, the agent must choose on of the safe paths to take.

For path selection, the developers used 3 different methods: randomly choosing one safe path, choosing the one with most pills and considering the number of pills and power pills in each path, as well as the position of the path on the tree. The results of the agent when run on the simulator were as follows:

Figure 3.1: Results from the Ms. Pac-Man simulator

Controller	Min	Max	Mean	s.e.
Rand-Safe-Path	710	6820	3068	146
Most-Pills-Path	1720	22370	8894	470
Hand-Coded-Sel	4270	43170	14757	782

The *Most-Pills-Path* strategy was much better than *Rand-Safe-Path*, due to it being able to clear levels faster and more often. The *Hand-Coded-Set* achieved even better results due to the use of the hand-coded rules mentioned before.

Figure 3.2: Results from the Ms. Pac-Man simulator

Controller	Min	Max	Mean	s.e.
Rand-Safe-Path	380	1900	820	34
Most-Pills-Path	2110	8050	3878	168
Hand-Coded-Sel	3570	15280	9630	346

The results for the experimentation on the screen capture model were, as expected, much lower. This is mainly because the information obtained by the agent in this format is less reliable than the one obtained directly by the simulator. Another small detail that may have caused many deaths that would not happen in the simulator is that, in the original game, Ms. Pac-Man slows down slightly while eating pills, while in the simulator it does not.

Overall, the authors were surprisingly satisfied with the results and mention, as the next step to improve this agent, the implementation of a full game tree or Monte Carlo based tree search.

3.4 An influence map model for playing Ms. Pac-Man

[10]

As the title suggests, this paper presents an agent for playing as Ms. Pac-Man using an influence model. The authors tried to make the agent as simple as possible, using only three parameters in the influence mapping function.

An influence map is a function defined over the game world representing the (un)desirability of an area. Game objects and agents exert an influence around their current location and the influence map is the sum of all these influences. These can be positive influences, like the ones from pills and super pills, or negative, like the ones from inedible ghosts. The agent is then attracted to the positively influenced areas of the map while trying to avoid the negative ones. An example of an influence map represented as a cloud on the Ms. Pac-Man game is in 3.3. As we can see, there is a much higher positive influence on the top-left quarter of the map, where there is a higher concentration of pills.

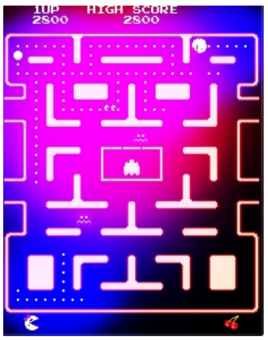


Figure 3.3: Influence map as a cloud

For this agent, the influence mapping function considers only 3 parameters when assessing the influence value of a point in the maze: the total number of uneaten pills and power pills, the distance between an uneaten pill and the evaluated position and the number of pills and power pills that have been eaten already. In each time step, the 4 adjacent positions to Ms. Pac-Man are evaluated and the one with highest value of influence is chosen as the move to be made.

The agent was able to perform and provided interesting insight into parameter and strategy evaluations. However, when submitted to the *2008 Congress on Evolutionary Computation (CEC) Ms. Pac-Man competition*, it achieved much lower results. The authors attribute this to the disparities between the testing game environment and the competition's game environment.

3.5 Monte-Carlo tree search in Ms. Pac-Man

[11]

This paper, unlike other agents, focuses solely on a single aspect of the challenge of playing as Ms. Pac-Man: to avoid pincer moves of the ghosts, in order to increase the chance of survival for Ms. Pac-Man.

Pincer moves are defined as game states where Ms. Pac-Man cannot escape, no matter what direction it takes. According to the authors, these moves are very hard to avoid and were the major downfall of many of the existing Ms. Pac-Man agents, since the ghosts do have some randomness associated to their behavior. Thus, by solving this problem, the chance of survival, and thus overall

score achieved, would be increased drastically. With this in mind, the authors implemented UCT, a Monte Carlo Tree Search algorithm that had proved successful in games like this previously.

This approach represents the game tree using Ms.Pac-Man's current position as the root node and intersections, also called by the authors "cross-points", as child nodes. Cross-points are where Ms. Pac-Man decides its next movement decision, making this points in the maze critical. After generating the game tree, the UCT algorithm is executed. As the algorithm progresses down the tree, the nodes are assign a mean reward value using UCB, which gives new nodes a very high mean reward value and otherwise assigns a value based on the average survival rate of the agent. Additionally, at any point in time, the agent is following one out of three possible tactics: *survival*, *feeding* or *pursuit*. The tactic followed by the agent changes over time, adapting to certain events like reaching a certain survivability threshold or eating a super pill.

In order to evaluate their agent, the authors executed the program according to the Ms. Pac-Man Competition rules and compared the results of their agent with the *ICE Pambush3* agent. The results were as follows:

Figure 3.4: Results

p	min	max	mean	s.d.
Proposed system	1	7	4.4	1.497
ICE Pambush3	1	5	3.4	0.970

As you can see, the proposed system outperformed *ICE Pambush3* in terms of highest level reached and average score achieved. However, it also shows a higher value of standard deviation, which means it is less consistent. Overall, the authors considered the proposed system a success, seen as it improved the performance of the previously developed agent.

3.6 Pac-mAnt: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man

[12]

The aim of this paper was to verify whether ant colony based optimization methods can be effective in the development of agents for real-time video games, focusing their testing on Ms. Pac-Man. To do this, the authors had to create some software, like a simulator for the Ms. Pac-Man game.

Ant Colony Optimization (ACO) algorithms have been applied to various combinatorial optimization problems, including the traveling salesman problem. These can be formalized by graphs where the objective is to minimize the cost of a route. The agent developed by the authors uses 2 types of ants:

collector ants which find paths with points, and *explorer ants* to find safe paths. To account for limited computing time for each decision, the maximum distance up to which an ant can explore is limited. At each times step, both types of ants are launched in all directions from Ms. Pac-Man. Then, if the distance to a ghost is lower than an established threshold, the agent chooses the best explorer ant. Otherwise, the best collector ant is chosen.

To optimize the parameters of the ACO algorithm, like the distance threshold mentioned before, the authors used a genetic algorithm with an evaluation function based on the mean score for each individual over the total number of games played.

To evaluate their agent, the authors tested it in both the simulator they created, which i mentioned previously, as well as in the actual game. The results were positive, but the authors point out the difficulty in extracting game information when testing on the real game and the high variability in the tests run on the simulator.

3.7 Enhancements for Monte-Carlo Tree Search in Ms Pac-Man

[13]

This paper tests 5 different enhancements for the MCTS algorithm and tests then on the Ms. Pac-Man game in order to understand which ones work better and why.

The first enhancement tested was a variable depth tree. In this case, the maximum tree depth is defined by the length of the edges, and subsequently the length of a path, instead of just the number of edges. This prevents the agent from choosing longer paths when in danger, as it differentiates paths with the same number of edges by their length.

The second enhancement was to create tactics to guide Ms. Pac-Man's decisions. These tactics are based on the three sub-goals of Ms. Pac-Man and, at any point in time, one of the tactics is active. If edible ghosts are nearby and the survival rate is above a certain threshold, **Ghost score** tactic is employed. If there are no nearby edible ghosts, Ms. Pac-Man is safe and the survival rate is above a certain threshold, **Pill score** tactic is employed. Otherwise, if the survival rate is under a certain threshold, **Survival** tactic is employed. These tactics affect the value used for the selection and back-propagation steps of the MCTS algorithm.

The third enhancement was to create strategies for Ms. Pac-Man to follow during the playout step. The authors considered the game to have 2 game ending states: Ms. Pac-Man dying or reaching level 16. Since it was not feasible to run playouts until one of these states was reached, the authors created 4 alternative conditions that stop the playout phase: a pre-set time length has passed, Ms. Pac-Man is dead, reaching the next maze and Ms. Pac-Man eating a power pill while edible ghosts are active. The score reached by the agent is then calculated considering if Ms. Pac-Man survived and how many pills

and ghosts were eaten.

The fourth enhancement was to create long-term goals for the agent. The MCTS algorithm considers only short-term rewards when running playouts. However, in the game of Ms. Pac-Man, there are situations that can improve the agent's performance in the long run. For example, eating edible ghosts as fast as possible reduces the probability of being trapped, and achieving 10.000 points rewards Ms. Pac-Man with a live. As such, these can be considered long-term goals. To represent these goals in the playout phase, the reward of eating a ghost is multiplied by the remaining edible time of said ghost, ensuring that ghosts earlier rather than later, and the reward value assigned to eating a power pill is set to zero if no ghosts are eaten in the following edible time, ensuring that Ms. Pac-Man waits for ghosts to be nearby before eating power pills.

Finally, the fifth enhancement was to employ end game tactics. These are meant to deal with situations where the density of pills in the maze is low enough that the pill or edible ghost closer Ms. Pac-Man is outside the search tree's range. This end-game tactic is made up of a small number of hand-coded rules that are meant to keep the agent away from defaulting to the survival tactic, seeing as this one is only useful in endangering situations.

The final agent employed all of these enhancements. As such, to test each of them individually, the agent was tested 5 times, with each time one of the enhancements being **disabled**. From all of the enhancements, the only testing that had a significant drop in performance was the one that implemented different strategies for the playout stage, which suggests that this enhancement is stronger than the others.

3.8 Evolution strategy for optimizing parameters in Ms Pac-Man controller ICE Pambush 3

[14]

This paper describes the application of an Evolutionary Strategy to optimize distance and cost parameters in a Ms. Pac-Man controller.

For this purpose, the authors use a simplified Evolutionary Strategy using only mutation. Their reasoning for not using other evolutionary operations like crossover is that each parameter has a specific role and, thus, mixing them is not suitable. Parameter optimization is divided into two stages; the first being *distance parameter* optimization and the second *cost parameter* optimization. For the first stage of optimization, the initial values could be set as the original values of the ICE Pambush 3 agent or as random values. Either distance or cost parameters would be optimized. Then, in the second stage, the parameters that hadn't been optimized would be chosen for optimization, resulting in alternate optimization. As such, 4 different approaches were possible: optimize distance first with initial values

as the original values, optimize distance first with initial values as the random values, optimize cost first with initial values as the original values or optimize cost first with initial values as the random values.

To evaluate the different approaches, the authors compared the median score achieved by the agent by the time the 300th generation was reached. The authors concluded that the optimal setup was to optimize first the distance parameters with initial values as random and then the cost parameters with initial values as random values as well, which resulted in improvement of 17% in the performance for the first game level, compared to their original agent ICE Pambush 3.

4

Development

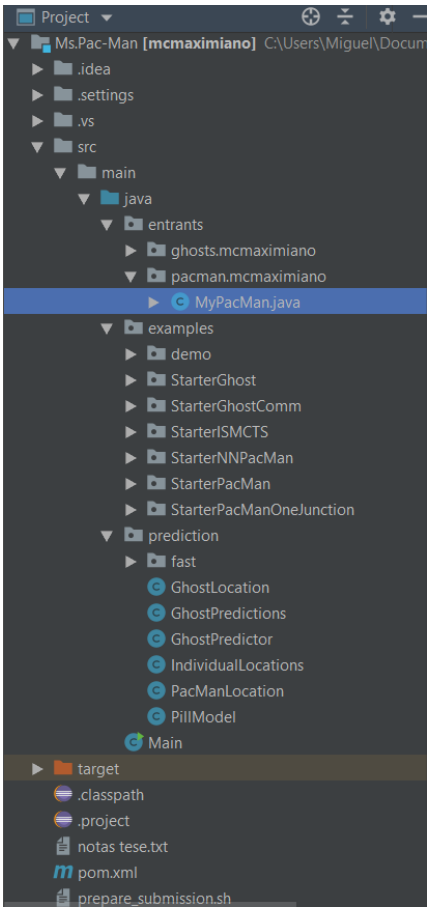
Contents

This chapter will provide a clear description of the environment in which this thesis was developed and the approach chosen to tackle the problem.

4.1 Development Environment

This agent was completely developed on the package provided by the *Ms. Pac-Man vs Ghost Team Competition* organizers, which is used by all the participants. Since most of the classes utilized by the engine are pre-compiled and have read-only access, forcing all the participants to use this package is the organization's way of making sure that everyone obeys the rules of the competition and the game. This results in a semi black boxed development environment, as the participants can, theoretically, decompile the classes that make up the engine and look at detailed implementations of all the methods they might need or want to use, but are expected to, instead, use the API available and explained in the competition's website (http://www.pacmanvghosts.co.uk).

Figure 4.1: File Structure



The package provided is all that is needed to develop an agent for the competition, along with an IDE (the competition organizers suggest Eclipse or IntelliJ, in this thesis the latter was used) with the Java SE Development Kit (JDK) installed. The package's file structure is illustrated in figure 4.1. The organizers provide the participants with several basic agents. These, located in the "examples" folder, serve as a basis for the participants to develop their own agents on, if they choose to do so. However, one could develop an agent by only opening 2 of these files: **MyPacMan.java** (highlighted) and **Main**.

MyPacMan.java is the file where the participant develops their agent. Initially it simply has a class, representing the agent, and a method, the one where the participant must place the game logic that will control the agent during the game. This method is required to return a MOVE and is the basis of the agent's behavior. The participant is free to add any sub-classes and methods they want to this class to support their solution. The code present in this file is as follows:

Listing 4.1: Initial MyPacMan

```
1 package entrants.pacman.username;
```

```
2
3  import pacman.controllers.PacmanController;
4  import pacman.game.Constants.MOVE;
5  import pacman.game.Game;
6
7  /*
8   * This is the class you need to modify for your entry. In particular, you need to
9   * fill in the getMove() method. Any additional classes you write should either
10  * be placed in this package or sub-packages (e.g., entrants.pacman.username).
11  */
12 public class MyPacMan extends PacmanController {
13     private MOVE myMove = MOVE.NEUTRAL;
14
15     public MOVE getMove(Game game, long timeDue) {
16         //Place your game logic here to play the game as Ms Pac-Man
17
18         return myMove;
19     }
20 }
```

Main.java comes ready to run. It initiates the basic executor and *controller* needed to launch the application. The code present in this file is as follows:

Listing 4.2: Initial Main

```
1
2  import examples.StarterGhostComm.Blinky;
3  import examples.StarterGhostComm.Inky;
4  import examples.StarterGhostComm.Pinky;
5  import examples.StarterGhostComm.Sue;
6  import examples.StarterPacManOneJunction.MyPacMan;
7  import pacman.Executor;
8  import pacman.controllers.IndividualGhostController;
9  import pacman.controllers.MASController;
10 import pacman.game.Constants.*;
11
12 import java.util.EnumMap;
13
14
```

```java
/**
 * Created by pwillic on 06/05/2016.
 */
public class Main {

    public static void main(String[] args) {

        Executor executor = new Executor.Builder()
                .setVisual(true)
                .setTickLimit(4000)
                .build();

        EnumMap<GHOST, IndividualGhostController> controllers = new EnumMap<>(GHOST.class);

        controllers.put(GHOST.INKY, new Inky());
        controllers.put(GHOST.BLINKY, new Blinky());
        controllers.put(GHOST.PINKY, new Pinky());
        controllers.put(GHOST.SUE, new Sue());

        executor.runGameTimed(new MyPacMan(), new MASController(controllers));
    }
}
```

The method *runGameTimed* runs the application with a graphical interface so the developer can see his agent in action.

In terms of development environment, these two files are all the participant needs to develop their agent successfully.

4.2 Development Process

After having the environment setup, the first step of the development of this agent was to look at the *StarterISMCTS* example agent provided in the competition package. One of the interesting properties of the MCTS algorithm is that one implementation can be easily ported to a different game and work, since the algorithm doesn't require any environment specific information to do its job. As such, the first step in the development process was to analyze the MCTS based agent provided by the competition, to see how the API was used and which methods from other classes it used. This resulted in a clear vision on how our solution would be implemented and which other classes had interesting and potentially useful methods.

That agent, provided by the competition, implemented information sets along with the MCTS algorithm, which is interesting but not useful for this thesis' agent. However, the base steps of the MCTS algorithm were implemented in a simple, concise fashion, so we ended up using those methods as a base for our own MCTS algorithm. Another method we ported into our agent was the *obtainDeterminisedState* method. This method will be detailed in the next chapter.

5

Solution

Contents

This chapter will take a close look at the agent developed and all the code necessary to make it work. Here, we will explain how the game state is perceived by the agent, how the MCTS algorithm and tree associated with it are implemented and a small adaptation added later into the development process to improve the agent's performance.

5.1 Overview of the Agent

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

5.1.1 MyPacMan

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

Listing 5.1: MyPacMan

```
1  public class MyPacMan extends PacmanController {
2
3      private Maze currentMaze;
4      private GhostPredictionsFast predictions;
5      private PillModel pillModel;
6      private int[] ghostEdibleTime;
7      protected Game mostRecentGame;
8      protected int maxTreeDepth = 30;
9      protected int maxPlayoutDepth = 250;
10
11     public MyPacMan() {
12         ghostEdibleTime = new int[Constants.GHOST.values().length];
13     }
14
15    public MOVE getMove(Game game, long timeToDecide) {...}
16
17    private Game obtainDeterminisedState(Game game) {...}
18 }
```

Attributes *maxTreeDepth* and *maxPlayoutDepth* are initialized manually because they are values chosen *a priori*. These values will be justified in the Evaluation chapter. *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. One could argue that this value could be hard coded since we know there are always 4 ghosts in a game of Ms. Pac-Man, but this is an easy way to account for possible changes in the future. 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. These methods are described as algorithms below:

```
Create GameInfo object from Game object;
Fill in info about Ms. Pac-Man (location and lastMoveMade);
Predict locations of Ghosts;
For each Ghost: Get Ghost's location;
if location is valid then
    Get edibleTime of Ghost;
    Get Ghost object with all info;
    Fill in info about Ghost (predictedLocation and edibleTime);
else
    Get Ghost object with default location;
    Fill in info about Ghost (default location and default edibleTime);
end
Place each Pill on the maze;
return Game object created from GameInfo object;
```

Algorithm 1: obtainDeterminisedState

.

```
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;
```

Algorithm 2: getMove

5.1.2 Node

This object is composed of 9 attributes and 15 methods, including 2 constructors.

Listing 5.2: 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;


    /* Constructors */

    //Constructor for root node
    public Node(MyPacMan MyPacMan, Game game) {...}

    //Constructor for child node
    public Node(Node parent, MOVE previousMove, MOVE[] legalMoves) {...}


    /* MCTS methods */

    public Node select_expand(Game game) {...}

    public double playout(Game game) {...}

    public void backPropagate(double value) {...}


    /* Auxiliary methods */

    protected MOVE[] getLegalMovesNotIncludingBackwards(Game game) {...}

    protected MOVE[] getAllLegalMoves(Game game) {...}
```

```
38
39      protected EnumMap<Constants.GHOST, MOVE> getBasicGhostMoves(Game game) {...}
40
41      private boolean isFullyExpanded() {...}
42
43      public Node expand(Game game) {...}
44
45      public Node selectBestChild() {...}
46
47      private double calculateChildScore() {...}
48
49      private double calculateGameScore(Game game) {...}
50
51     /* Other methods */
52
53     //Method initially used to select the best move
54     public MOVE selectBestMove() {...}
55
56     //The Adaptation
57      public MOVE selectBestMove(Game game) {...}
58      public MOVE getNextMove(Node child, Game game) {...}
59 }
```

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.

Listing 5.3: Constructors

```
1 //Constructor for root node
```

```
2  public Node(MyPacMan MyPacMan, Game game) {
3      this.MyPacMan = MyPacMan;
4      treeDepth = 0;
5      this.legalMoves = getLegalMovesNotIncludingBackwards(game);
6      this.children = new Node[legalMoves.length];
7  }
8
9  //Constructor for child node
10 public Node(Node parent, MOVE previousMove, MOVE[] legalMoves) {
11     this.MyPacMan = parent.MyPacMan;
12     this.parent = parent;
13     this.treeDepth = parent.treeDepth + 1;
14     this.prevMove = previousMove;
15     this.legalMoves = legalMoves;
16     this.children = new Node[legalMoves.length];
17 }
```

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.

Looking back at Listing 4.2, you can see in the *other methods*, there are 2 methods called *selectBestMove*. As the comments suggest, 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 sections of this chapter and results will be discussed in the next chapter.

5.2 The MCTS Algorithm

This section will look into the MCTS algorithm implemented in this thesis. In order to avoid a cluster of code, all the auxiliary methods not present in this section will be in Appendix A.

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 used on other games, but there are some changes we had to make to accommodate the algorithm to the game in question. 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.

Listing 5.4: Selection_Expansion

```
1  public Node select_expand(Game game) {
2      Node current = this;
3      while (current.treeDepth < MyPacMan.maxTreeDepth && !game.gameOver()) {
4          if (current.isFullyExpanded()) {
5              current = current.selectBestChild();
6              game.advanceGame(current.prevMove, getBasicGhostMoves(game));
7          } else {
8              current = current.expand(game);
9              game.advanceGame(current.prevMove, getBasicGhostMoves(game));
10             return current;
11         }
12     }
13     return current;
14 }
```

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.

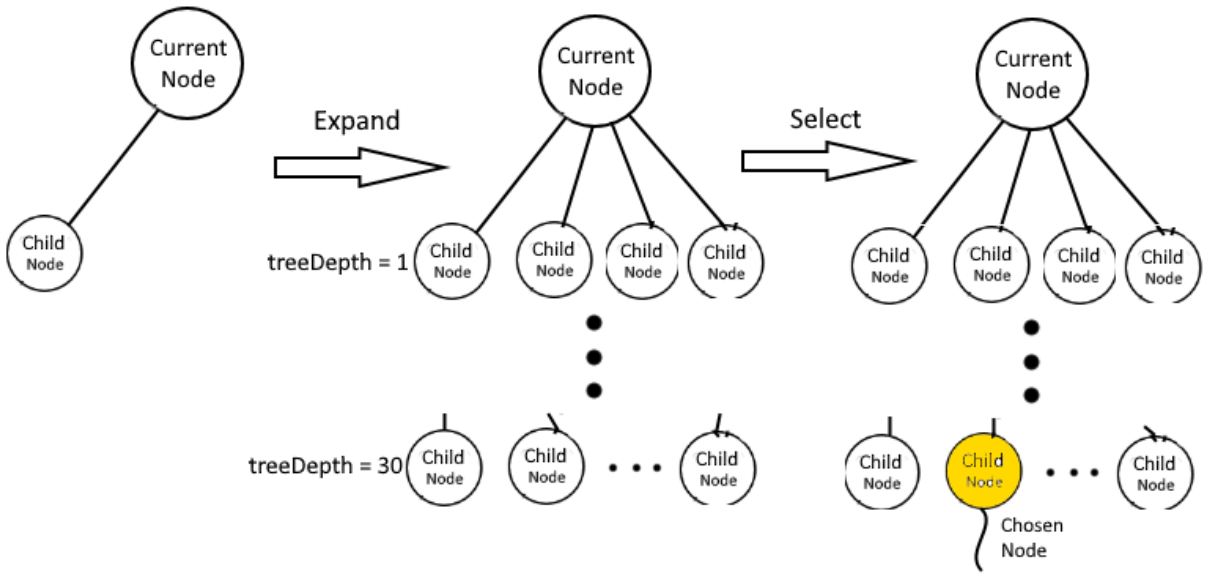
One small detail to be noticed is that this MCTS algorithm does not implement **UCB**/**UCT** in its **Selection** step. The classic UCT formula is as shown in 5.1, where *wi* stands for the number of wins for the node considered after the i-th move and *ni* stands for the number of simulations for the node considered after the i-th move, *Ni* stands for the number of simulations for the node considered after the i-th move by the parent node and *c* is the exploration parameter. This is not implemented in our agent 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.

Figure 5.1: UCT formula

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

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

Figure 5.2: Example of a decision tree



Listing 5.5: Playout

```
1  public double playout(Game game) {
2      int depth = treeDepth;
3      Random random = new Random();
4      while (depth < MyPacMan.maxPlayoutDepth) {
5          if (game.gameOver()) break;
6          MOVE[] legalMoves = getAllLegalMoves(game);
7          MOVE randomMove = legalMoves[random.nextInt(legalMoves.length)];
8          game.advanceGame(randomMove, getBasicGhostMoves(game));
9          depth++;
10     }
11     return calculateGameScore(game);
12 }
```

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. It is important to note that this score does **not** correspond to the in-game score achieved by the agent. 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.

Listing 5.6: BackPropagation

```
public void backPropagate(double value) {
    Node current = this;
    while (current.parent != null) {
        current.visits++;
        current.score += value;
        current = current.parent;
    }
    current.visits++;
}
```

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.

5.3 The Adaptation

As mentioned earlier, in Listing 4.2, 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. Let's take a look at the first iteration of the method:

Listing 5.7: Initial selectBestMove

```
public MOVE selectBestMove() {
    Node bestChild = null;
    double bestScore = -Double.MAX_VALUE;
    for (Node child : children) {
        if (child == null) continue;
        double score = child.score;
        if (score > bestScore) {
            bestChild = child;
            bestScore = score;
        }
    }
    return bestChild == null ? MOVE.NEUTRAL : bestChild.prevMove;
}
```

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. A quick solution for this problem was to make this move legal. But then the agent would run back and fourth in the same place because it was safe, there is no chance being confronted with a ghost if we don't go anywhere and, after all, game time does matter for game score calculation. That is until one ghost found the agent and then it would get ran down by the ghost team.

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. Here is the method:

Listing 5.8: Adapted selectBestMove

```
1  public MOVE selectBestMove(Game game) {
2      Node bestChild = null;
3      double bestScore = -Double.MAX_VALUE;
4      for (Node child : children) {
5          if (child == null) continue;
6          double score = child.score;
7          if (score > bestScore) {
8              bestChild = child;
9              bestScore = score;
10         }
11     }
12
13     if (bestChild == null) {
14         return MOVE.NEUTRAL;
15     } else {
16         return getNextMove(bestChild, game);
17     }
18 }
19
20 public MOVE getNextMove(Node child, Game game) {
21     int[] ghostEdibleTimeMock;
22     MOVE nextMove = child.prevMove;
23     GhostPredictionsFast predictions;
24     ghostEdibleTimeMock = new int[Constants.GHOST.values().length];
25     Arrays.fill(ghostEdibleTimeMock, -1);
26     predictions = new GhostPredictionsFast(game.getCurrentMaze()); //Predicts ghosts' move
```

```
27      predictions.preallocate();
28      for (Constants.GHOST ghost : Constants.GHOST.values()) {
29          if (ghostEdibleTimeMock[ghost.ordinal()] != -1) {
30              ghostEdibleTimeMock[ghost.ordinal()]--;
31          }
32
33          int ghostIndex = game.getGhostCurrentNodeIndex(ghost);
34          if (ghostIndex != -1) { //We see the ghost!
35              predictions.observe(ghost, ghostIndex, game.getGhostLastMoveMade(ghost));
36              ghostEdibleTimeMock[ghost.ordinal()] = game.getGhostEdibleTime(ghost);
37              MOVE ghostDirOpp = game.getNextMoveAwayFromTarget(
38                                      game.getPacmanCurrentNodeIndex(),
39                                      game.getGhostCurrentNodeIndex(ghost),
40                                      DM.PATH
41                                  );
42              if (Arrays.stream(getAllLegalMoves(game)).anyMatch(ghostDirOpp::equals) &&
43              game.getGhostEdibleTime(ghost) <= 0) {
44                  nextMove = ghostDirOpp;
45              }
46          }
47
48      }
49      return nextMove;
50  }
```

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

This is not standard for MCTS approaches but we believe that in this case it was necessary to use a solution like this to overwrite the basic movement behavior that wouldn't let the agent turn backwards even when moving towards certain death.

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

6

Evaluation

Contents

In this chapter, we will look at how the agent was tested and the results obtained. We compared the performance of both versions of the agent, with and without the adaptation mentioned in the previous chapter. We also took a quick glance at the results obtained by other participants of the competition in previous years.

6.1 Evaluation Methodology

To evaluate the agent, we used the *runExperiment* method in the **Main** file, instead of the previously mentioned *runGameTimed*. This method also belongs to the *executor* class and is very useful to test the agent's performance. It takes in one more argument than *runGameTimed* which is the number of times the game should be played. This facilitates batch testing. It also doesn't run a graphic interface, so it is less computationally intense for the computer running it, 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**. In theory, this interval would allow us to see if a certain version of the agent is clearly better than another. However, as we will see in the next section of this chapter, 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.

All the data will be available in the **Appendix B**. In the next section we organize some of this data in order to interpret the results.

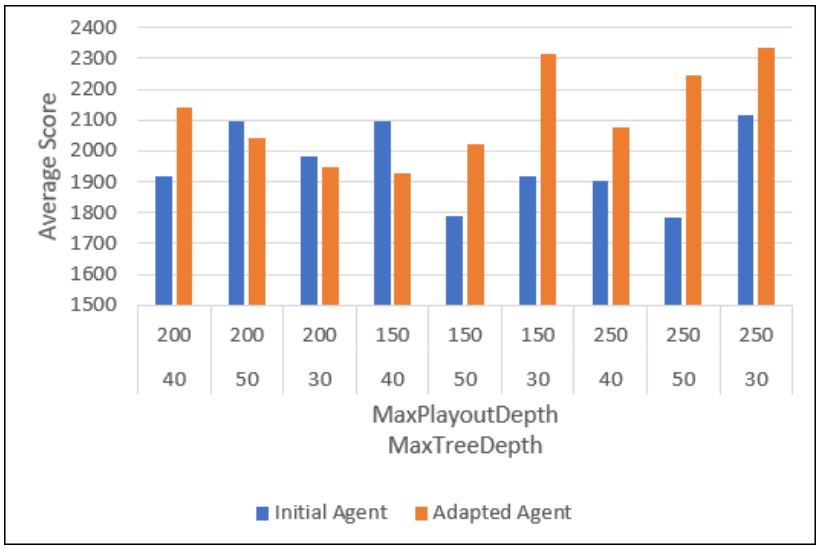
6.2 Results

In this section, we will take a look at the results obtained by the agent and relevant information we can deduce from them.

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.

As we can see in figure 6.1, 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

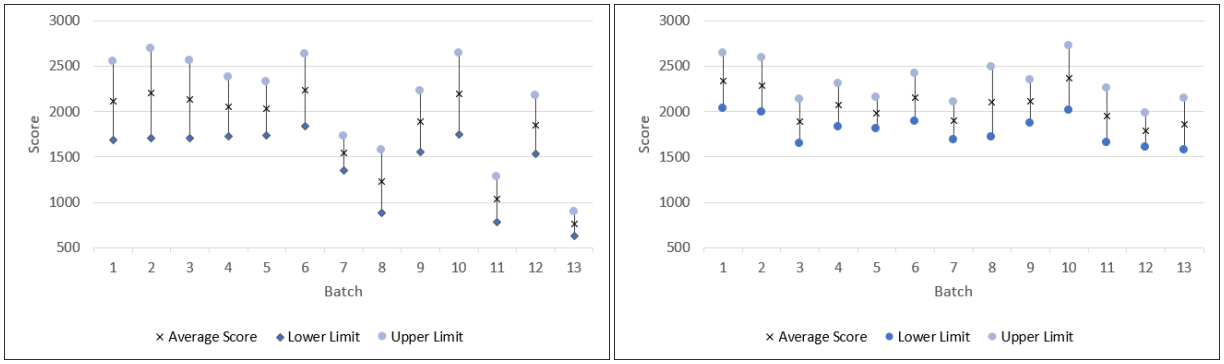
Figure 6.1: Group 1 test results



of the 2 attributes was ideal.

However, taking that into consideration, the group 2 tests were run using these values for (*maxTreeDepth*, *maxPlayoutDepth*). The results were as follows:

Figure 6.2: Group 2 test results: Initial (Left) Vs Adapted Agent (Right)



These 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*. A quick look at the two side-by-side graphs reveals what was mentioned in the previous chapter: 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 what was explained in the previous chapter: 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 obtaining 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.

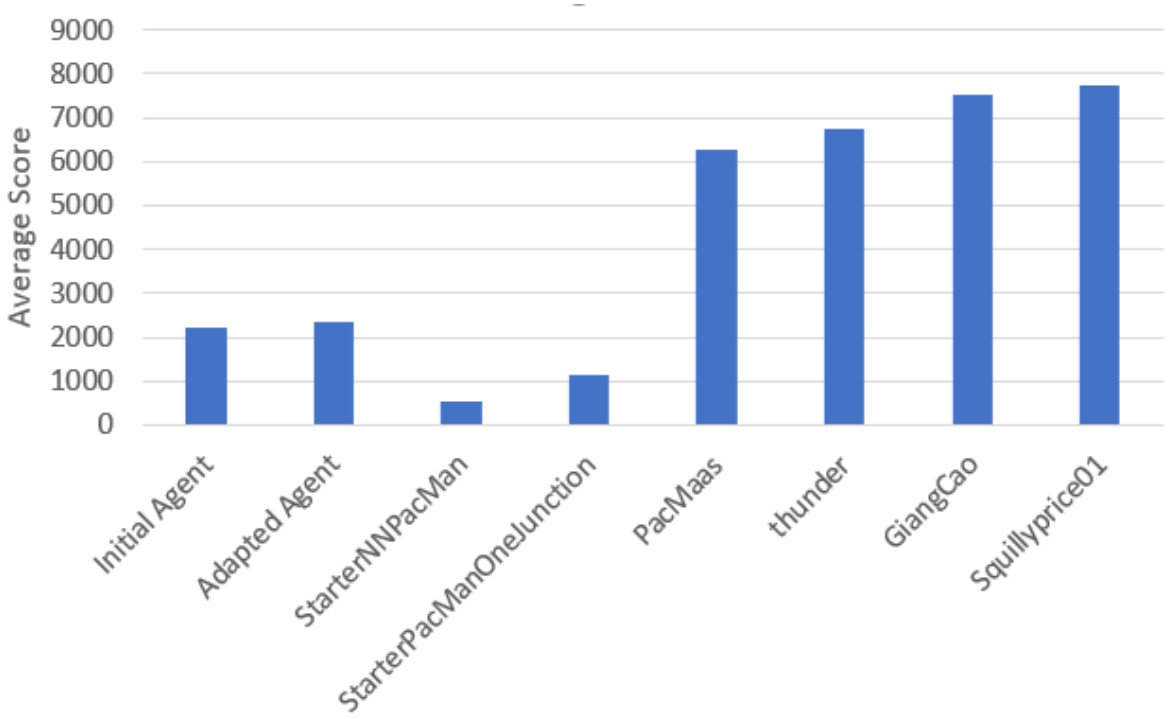
Batches **11**, **12** and **13** go a step further, and set 2 out of 3 factors to 0: **batch 11** sets the weight of *game time* and *level* to 0, **batch 12** sets the weight of *in-game score* and *level* to 0 and **batch 13** sets the weight of *in-game score* and *game time* to 0. In other words, 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 6.2, 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.

These results show that strategies balanced between maximizing score and game time result in the best performance, for both agents, which shows the importance of adaptation in the agent. Also, the

fact that the adapted agent obtained better results than the initial one reinforces the claim that having a secondary algorithm to aid MCTS deal with partial observability is very valuable for the success of such agents.

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 following graph:

Figure 6.3: Comparison to basic agents from 2018



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.

7

Conclusions and Future Work

Contents

7.1 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. These results are far from some of the agents submitted to the competition in the last 2 years, but 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.

7.2 Solution Limitations and Future Work

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. A possible fix for this problem would be to have a secondary algorithm coupled with MCTS to either help the agent deal with lack of information and randomness of the game environment in general or introduce human experts' knowledge into the agent to make it *smarter* overall.

Bibliography

[1] Murray Campbell, A. Joseph Hoane Jr., Feng-hsiung Hsu, *Deep Blue*, 2002, vol. 134, no. 1-2.

[2] 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. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/26819042

[3] Piers R. Williams, Diego Perez-Liebana and Simon M. Lucas, "Ms. Pac-Man Versus Ghost Team CIG 2016 Competition." [Online]. Available: http://www.diego-perez.net/papers/PacmanGhosts2016Competition.pdf

[4] Simon M. Lucas, "Screen-capture Ms Pac-Man," 2009. [Online]. Available: https://ieeexplore.ieee.org/document/5286506

[5] Philipp Rohlfshagen, Simon M. Lucas, "Ms Pac-Man versus Ghost Team CEC 2011 Competition." [Online]. Available: https://core.ac.uk/download/pdf/9589933.pdf

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

[7] Fernando Dominguez-Estevez, Antonio A. Sanchez-Ruiz, Pedro Pablo Gomez-Martin, "Training Pac-Man bots using Reinforcement Learning and Case-based Reasoning," 2017. [Online]. Available: http://ceur-ws.org/Vol-1957/CoSeCiVi17_paper_14.pdf

[8] Greg Foderaro, Ashleigh Swingler, Silvia Ferrari, "A Model-Based Approach to Optimizing Ms. Pac-Man Game Strategies in Real Time," 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7395335

[9] David Robles, Simon Lucas, "A simple tree search method for playing Ms. Pac-Man," 2009. [Online]. Available: https://www.researchgate.net/publication/221157530_A_simple_tree_search_method_for_playing_Ms_Pac-Man

[10] Nathan Wirth, Marcus Gallagher, “An in[U+FB02]uence map model for playing Ms. Pac-Man ,” 2008. [Online]. Available: https://ieeexplore.ieee.org/document/5035644

[11] Nozomu Ikehata, Takeshi Ito, “Monte-Carlo tree search in Ms. Pac-Man ,” 2011. [Online]. Available: https://ieeexplore.ieee.org/document/6031987

[12] Martin Emilio, Martinez Moises, Recio Gustavo, Saez Yago, “Pac-mAnt: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man,” 2010. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5593319

[13] Tom Pepels, Mark H. M. Winands, “Enhancements for Monte-Carlo Tree Search in Ms Pac-Man,” 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6374165

[14] Ruck Thawonmas, Takashi Ashida, “Evolution strategy for optimizing parameters in Ms Pac-Man controller ICE Pambush 3,” 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5593350

A

Code of Project

Listing A.1: getMove

```
1  public MOVE getMove(Game game, long timeToDecide) {
2
3          //We need a model of the game! Do this to set initial state of the maze:
4          if (currentMaze != game.getCurrentMaze()){
5              currentMaze = game.getCurrentMaze();
6              predictions = null;
7              pillModel = null;
8              Arrays.fill(ghostEdibleTime, -1);
9          }
10
11         mostRecentGame = game;
12
13         if (game.gameOver()) return null;
```

```
14
15          if (game.wasPacManEaten()) { //Ms. Pac-Man died and just respawned, she has no memory
16              predictions = null;
17          }
18
19          if (predictions == null) {
20              predictions = new GhostPredictionsFast(game.getCurrentMaze());
21              predictions.preallocate();
22          }
23          if (pillModel == null) {
24              pillModel = new PillModel(game.getNumberOfPills());
25
26              int[] indices = game.getCurrentMaze().pillIndices; //Indices = pill spawn points
27              for (int index : indices) {
28                  pillModel.observe(index, true); //Put a pill in each pill spawn point
29              }
30          }
31
32          // Update the pill model with what isn't available anymore
33          // (Because when PM goes through a pill, the pill disappears)
34          int pillIndex = game.getPillIndex(game.getPacmanCurrentNodeIndex());
35          if (pillIndex != -1) {
36              Boolean pillState = game.isPillStillAvailable(pillIndex);
37              if (pillState != null && !pillState) {
38                  pillModel.observe(pillIndex, false); //Tell the game that there is no pill
39              }
40          }
41
42          // Get observations of ghosts
43          //and pass them to the predictor (accounts for partial observability)
44          for (Constants.GHOST ghost : Constants.GHOST.values()) {
45              if (ghostEdibleTime[ghost.ordinal()] != -1) {
46                  ghostEdibleTime[ghost.ordinal()]--;
47              }
48
49              int ghostIndex = game.getGhostCurrentNodeIndex(ghost);
50              if (ghostIndex != -1) { //We see the ghost!
51                  predictions.observe(ghost, ghostIndex, game.getGhostLastMoveMade(ghost));
```

```
52                  ghostEdibleTime[ghost.ordinal()] = game.getGhostEdibleTime(ghost);
53              } else { //We do not see the ghost...
54                  List<GhostLocation> locations = predictions.getGhostLocations(ghost);
55                  locations.stream().filter(location ->
56                          game.isNodeObservable(location.getIndex())).forEach(location -> {
57                      predictions.observeNotPresent(ghost, location.getIndex());
58                  });
59              }
60          }
61          //Now we have the game modeled! Next comes MCTS:
62          Node root = new Node(this, game);
63          while(System.currentTimeMillis() < timeToDecide) {
64              //MCTS can't deal with PO by itself. We give it a copy of the game without PO,
65              //so MCTS thinks it sees everything
66              Game copy = obtainDeterminisedState(game);
67              //Select & Expand
68              Node node = root.select_expand(copy);
69              // Play-out
70              double gameScore = node.playout(copy);
71              // Back-propagate
72              node.backPropagate(gameScore);
73          }
74          predictions.update();
75          return root.selectBestMove(game);
76      }
```

Listing A.2: obtainDeterminisedState Method

```
1  private Game obtainDeterminisedState(Game game) {
2          GameInfo info = game.getPopulatedGameInfo();
3          info.setPacman(new PacMan(game.getPacmanCurrentNodeIndex(),
4                                    game.getPacmanLastMoveMade(), 0, false));
5          EnumMap<Constants.GHOST, GhostLocation> locations = predictions.sampleLocations();
6          info.fixGhosts(ghost -> {
7              GhostLocation location = locations.get(ghost);
8              if (location != null) {
9                  int edibleTime = ghostEdibleTime[ghost.ordinal()];
10                 return new Ghost(ghost, location.getIndex(),
```

```
                                    edibleTime, 0, location.getLastMoveMade());
            } else {
                return new Ghost(ghost, game.getGhostInitialNodeIndex(), 0, 0, MOVE.NEUTRAL);
            }
        });

        for (int i = 0; i < pillModel.getPills().length(); i++) {
            info.setPillAtIndex(i, pillModel.getPills().get(i));
        }
        return game.getGameFromInfo(info);
    }
```

Listing A.3: calculateGameScore Method

```
private double calculateGameScore(Game game) {
    return game.getScore() + game.getTotalTime() + (1000 * game.getCurrentLevel());
}
```

B

Result Tables

This appendix contains tables with all the data obtained during the evaluation of the agents. They are placed here in case the reader wants to look at the data in detail, since they are too big to fit in the Results chapter without disturbing the flow of text. Each line from tables B.1 and B.3 corresponds to a bar from 6.1, blue and orange respectively, while each line from tables B.2 and B.3 corresponds to a test batch from 6.2.

Table B.1: Group 1 of tests for the initial agent

Average Score	Min. Score	Max. Score	Standard Deviation	Standard Error	95% Confidence Lower Limit	95% Confidence Upper Limit	Max Tree Depth	Max Playout Depth
1917.67	890	4900	953.17	174.02	1576.5908	2258.7492	40	200
2097	910	4980	928.22	169.47	1764.8388	2429.1612	50	200
1983.67	1120	5120	1039.18	189.73	1611.7992	2355.5408	30	200
2069	180	6160	1265.28	230.82	1643.5928	2548.4072	40	150
1787.67	820	3390	736.07	134.39	1524.2656	2051.0744	50	150
1918	540	6080	1084.39	197.98	1529.9592	2306.0408	30	150
1903.67	840	3760	810.17	147.92	1613.7468	2193.5932	40	250
1785	720	5450	964.06	176.01	1440.0204	2129.9796	50	250
2115.67	790	5410	1210	219.27	1695.9008	2545.4392	30	250

Table B.2: Group 2 of tests for the initial agent (maxTreeDepth=30, maxPlayoutDepth=250)

Average Score	Min. Score	Max. Score	Standard Deviation	Standard Error	95% Confidence Lower Limit	95% Confidence Upper Limit	Score Weight	Time Weight	Level Weight
2115.67	790	5410	1210	219.27	1695.9008	2545.4392	1	1	1k
2199.33	570	6610	1368.52	249.8	1709.722	2688.938	1	0.5	2k
2130.33	570	5660	1182.83	215.95	1707.068	2553.592	1	0.2	5k
2049	960	4750	899.82	164.29	1726.9916	2371.0084	20	1	5k
2033.33	900	3820	835.71	150.75	1737.86	2328.8	50	0.1	5k
2232.33	1140	5640	1105.96	201.92	1836.5668	2628.0632	50	0.01	10k
1538.33	1060	2840	523.06	95.5	1351.15	1725.51	0	1	5k
1229	430	5000	964.14	176.03	883.9812	1574.0118	1	0	5k
1891.67	750	4290	937.61	171.18	1556.1572	2227.1828	1	1	0
2196.33	850	6250	1247.14	227.7	1750.038	2642.622	50	1	100k
1032.67	410	2630	698.96	127.61	782.5544	1282.7856	1	0	0
1850.33	720	5330	892.63	162.97	1530.9088	2169.7512	0	1	0
759.67	390	2150	370.34	67.62	627.1348	892.2052	0	0	1k

Table B.3: Group 1 of tests for the adapted agent

Average Score	Min. Score	Max. Score	Standard Deviation	Standard Error	95% Confidence Lower Limit	95% Confidence Upper Limit	Max Tree Depth	Max Playout Depth
2140.33	760	5000	842.78	153.87	1838.7448	2441.9152	40	200
2044.33	650	3570	768.88	140.38	1759.1852	2319.4748	50	200
1949.67	1070	4880	776.14	141.7	1671.938	2227.402	30	200
1928	760	4600	836.07	152.65	1628.806	2227.194	40	150
2020.67	850	3790	853.52	155.83	1715.2432	2326.0968	50	150
2317	730	4710	861.39	136.2	2104.048	2637.952	30	150
2078.75	750	3450	697.2	110.24	1862.6796	2294.8204	40	250
2243.75	990	5290	1036.94	163.96	1922.3884	2565.1116	50	250
2333.75	1220	5530	977.32	154.53	2030.8712	2636.6288	30	250

Table B.4: Group 2 of tests for the adapted agent (maxTreeDepth=30, maxPlayoutDepth=250)

Average Score	Min. Score	Max. Score	Standard Deviation	Standard Error	95% Confidence Lower Limit	95% Confidence Upper Limit	Score Weight	Time Weight	Level Weight
2333.75	1220	5530	977.32	154.53	2030.8712	2636.6288	1	1	1k
2287	880	5080	965.34	152.63	1987.8452	2586.1548	1	0.5	2k
1891.25	790	4640	786.68	124.39	1647.4456	2135.0544	1	0.2	5k
2068.75	720	4390	780.91	123.47	1826.7488	2310.7512	20	1	5k
1977.5	670	2990	556.31	87.96	1805.0984	2149.9016	50	0.1	5k
2152.25	920	4290	856.41	135.41	1886.8464	2417.6536	50	0.01	10k
1895.5	680	3540	669.29	105.82	1688.0928	2102.9072	0	1	5k
2101.25	540	5890	1252.29	198	1713.17	2489.33	1	0	5k
2108.5	1030	4020	769.53	121.67	1870.0268	2346.9732	1	1	0
2366	940	5930	1144.05	180.89	2011.4556	2720.5444	50	1	100k
1952.25	400	4360	968.69	153.16	1652.0564	2252.4436	1	0	0
1793	710	3770	615.1	97.26	1602.3704	1983.6296	0	1	0
1860	620	5450	930.84	147.18	1571.5272	2148.4728	0	0	1k