UNIVERSITY OF OSLO
Department of Informatics

Thanatos - a learning RTS Game AI

Developing an AI capable of learning build orders in a Real Time Strategy Game Environment by using reinforcement learning

Master thesis

Richard Rørmark

November 2, 2009

Abstract

Real-time strategy games are challenging test beds for intelligent decision making algorithms with their rich and complex environments. Interaction with these games are made in form of ingame decisions. Improving these decisions over time is a key feature to succeeding in RTS games. In this thesis we examine the consituent parts of a general RTS game AI, address the importance of build orders and discuss how reinforcement learning can be applied. Moreover, we employ a representation where a single decision is selected between the decisions of multiple experts, each learning only within its own field of expertise. For this we use a slightly modified version of a blackboard architecture. Additionally, the reinforcement rewards are delayed until the game has ended. With this we would expect to see an overall improvement of the selected actions, in terms of faster and more efficient build orders. We implement the concept described as a computer controllable player AI in a fully working RTS game environment called Wargus. By experimenting with various parameters and exploration strategies we found that the overall decisions were greatly improved and human-like winning strategies emerged. Additionally, the exploration strategies showed to influence the learning ability to a high degree.

Contents

List of Figures

Listings

Acknowledgements

I would like to point out and show my appreciation to some of the people who have helped and encouraged me throughout the entire project.

First out is the Stratagus team. Even though I haven't had any first-hand interaction with the team itself, I still want to applaude their effort in developing such a great non-profit game engine. Working on Stratagus has been extremely rewarding.

Thanks to Simen Kolloen who provided me with neat graphics and a fancy logo for the developed application.

I would like to thank my good friend Djamel Adjou, whom I've had numerous weird conversations with throughout this process, discussing everything from conciousness to teleporting. His motivational support has been of great benefit to me. Also a thanks to Andreas Våvang Solbrå for his support and constructive discussions on the topic, helping me stay focused, as well as our endless battles in console games, whenever I needed a break.

A big thanks to my supervisor Herman Ruge Jervell, for his cooperation and willingness to let me combine my passion with my work.

I would also like to thank my parents, who have been of great motivational support and always believed in me, pushing me forward in life. Without them I wouldn't be here, like, literally.

A deep and sincere thanks goes out to my three roommates Sindre Aarsæther, Anders Hafreager and Bedeho Mender. Sindre; who helped trigger some of my epiphanies, during what to me felt like my darkest hours. Anders; who helped design the ideas behind the heuristic algorithm, let me run simulations on his computer and generally was a great listener. Bedeho; who gave me the idea of taking on this subject and pushed me to make sure it happened, as well as providing me with invaluable constructive help and motivation.

They have also helped me get through possibly the hardest time of my life on a personal level. For this, I cannot stress enough how greatful I am. From the bottom of my heart; **Thank You**.

Part I

Introduction

1 Introduction

After such an introduction, I can hardly wait to hear what I'm going to say.

— *Evelyn Anderson*

Real Time Strategy is a computer game genre where players construct buildings, gather resources and purchase units, used to defeat opponents with. Due to a range of different properties, the amount of entities acquired and the selected component mixture is cruicial to winning the game. Decision making is thus an important part of the game.

When a player is controlled by Aritificial Intelligence (AI), this is usually done with a decision making technique, as described in section 4.1. Using the same decisions over and over again would make you predictable and give the opponent a huge advantage. This could be avoided if the decisions were changed from time to time, or even better, improved by learning, forcing the opponent to improve as well.

With reinforcement learning, a reward is given to a decision maker if something positive happens or a punishment if something negative happens. This way it can adjust the decisions accordingly, so that better and better decisions are made over time.

In this thesis we seek to see how reinforcement learning can be used to improve the decision making of an AI player in an RTS game. Section 5 goes into more detail regarding the problem definition.

Some of the RTS terms we use throughout this thesis may be unfamiliar to the reader. We have put together a list, briefly explaining each of these words. The list can be seen in appendix A.

1.1 Project Motivation

With a history consisting of several thousand games played and an above the average interest in artificial intelligence and how the brain works, I wanted to contribute to knowledge by combining two of my passions.

With a first hand experience with commercially available AI opponents, I have always been aware of how bad they perform. Even with ways of cheating, they never tend to vary or improve their gameplay, so it is easy to find a way to beat them.

Despite online multi-player games are getting more and more common, most RTS games are still released with the option of playing against an AI opponent. If these AIs are to pose any real challenges to human players, they need to be able to adapt and learn, in order to keep up with the fast learning human players.

The project can thus be regarded as greatly motivated by the desire to direct attention and expand the current field of learning in RTS games, as well as increase the fun factor in playing against a computer AI opponent in these "online multi-player days".

1.2 Methodology

The following list describes the system of methods we have approached to solve the problem

- Analyse possible Real-Time Strategy frameworks and select an appropriate one.
- Analyse the consituents of an RTS game AI and the state of art in game AI.
- Select appropriate techniques and algorithms, and design the actual AI.
- Modify the selected framework and implement the designed AI, called Thanatos.
- Implement a control panel for running controlled simulations, AI monitoring and statistical purposes.
- Evaluate the AI by using the control panel, running a set of experiments, with the AI playing against itself and a set of fixed opponents.

1.3 Expected Results

As we disregard most superfluous aspects of the RTS game AI in our research (6) and direct our attention to build orders and decision making, the AI will be very limited. Since a lot of the functionalities a human player would normally have during a game is not present for the AI, it should be extremely weak compared to human players.

Nonetheless, we expect the AI to be able to improve the overall build orders (6.3) at a fairly slow rate. Despite restrictions and limitations, the game is still complex and there are a large amount of possible actions and a vast amount of states to be in. With the limited time we have at hand for running simulations, we should be able to see clear improvements over the course of a simulation, but not enough games to acquire near optimal strategies, if such exists.

We also expect that parameters and choice of exploration strategies will have a great impact on the learning performance. The heuristic algorithm, presented in section 8.4.4, even though it only uses two additional game related variables to adjust the rewards with, should result in strategies with a higher win rate, with a fairly mixed focus on income and a desire to end the game quickly.

1.4 Outline of the Report

The thesis is divided into 5 individual parts, in addition to this introductory section and a set of appendices. The first part describes what we regard as background information. This will entertain the reader on the Real Time Strategy genre and present a brief introduction to reinforcement learning.

The second part presents litterature and research related to RTS games and general game AI techniques. This illustrates the current state of the art in game AI and connects some of the most common techniques to the RTS domain. The research is followed by an analysis part where we clarify the problem at hand and discuss the framework to implement our AI in. We also take a deeper look

at how the AI is assembled, by discussing each of the consituents of a computer controllable player AI.

The next part takes on the implementation and experimentation phase. We describe the modifications and limitations made to the framework, which parts of the framework we will be using, and most importantly the AI design. This includes representation, algorithms and reinforcement learning strategies. Experiments are also discussed and results presented.

We end the thesis with a conclusion and a discussion on how this thesis can serve as the starting point for future work.

Part II

Background Information

2 A General Overview of Real Time Strategy Games

> *Once upon a time, you used to have a board, a lot of counters, a book of rules and maybe some dice. But with digital entertainment being so all prevailing these days, old tabletop strategy games have either disappeared or migrated across to the computer realm. [Fle06]*

Real Time Strategy is one of the many different types of computer games that can be found in stores around the world today. These games are not restricted to personal computers, but it is the most common platform as they often put the processor to the test. The games come with graphics that must be rendered, sound effects to be played and maybe even some ingame cinematic scenes. At the same time, the processor must calculate unit positions, unit movement and user decisions for possibly hundres of individual units. Most games even offer a computer controllable player as an opponent, requiring additional cpu cycles from the processor.

These games put you up against an opponent, whether this is another human being or a computer AI, is usually up to the player. Most of these games also allow for some gigantic and epic battles when multiple players collide in teams, either through a LAN[1] connection or over the internet.

The Real Time Strategy game offers a rich and challenging environment with problems like base building, resource harvesting, logistics, game world intelligence, army production and strategic manouevers to handle. For more information on various computer game genres, see [Smi08].

2.1 The Genre

> *Trying to imagine a time without real-time strategy games is perhaps a bit like trying to imagine a time before the World Wide Web. [Gera]*

Let us illustrate the genre by an example, starting out in a familiar environment. Take a game of chess[2] as an example. In this game you have a board of 8×8 squares, alternating between black and white. There is no elementary difference between the black and the white squares, but there are some rules that apply to them. A bishop can only move diagonally, so it is forced to always be located at squares of the same colour. The initial placement of kings and queens are also based upon the square colouring. Each player initially has a set of 16 pieces, and takes turns, moving one unit at the time.

We can modify the turn-based chess game to make it work as a simple RTS[3] game. We will refer to the chess board as the map, pieces as units and squares as tiles, so when referring to the modified game, we use a more RTS-friendly terminology.

First, imagine that we increase the number of tiles to 256×256, and instead of letting a tile either be black or white, we let each one take on a given terrain type, defined by the following hypothetical terrain set

[1] Local Area Network
[2] For more information on chess rules and how to play chess, see [Che08]
[3] Real Time Strategy

$T(t)_{rts}$ ={Grass, Rocks, Mud, Swamp, Sea, Empty space}

where t is the tile on the map and T(t) is the set of possible terrain types the tile can take on.

We let each player have approximately 100 units scattered around the map. Additional ones can be purchased on a regular basis. Every unit has a set of properties that defines how it acts on the map and how it should interact when facing opponent units (7.3). In the simple case of regular chess, this is limited to how a specific unit can move on the map. The properties for a chess unit would be

$P(u)_{chess}$ ={Board movement}

where u is the unit, or piece and P(u) denotes the set of properties the unit has.

In the RTS converted chess game we define 4 different properties, each more complex than in the regular chess scenario. Each unit will have one or more of these properties and two units with equal properties are said to be of the same unit type. The properties are hit points, attack damage, magical abilities and regular movement. The property space is thus

$P(u)_{rts}$ = {Health points, Attack damage, Magical powers, Map movement}

We top it off by eliminating game turns, letting each player move freely. The player can now do as many moves as possible, without having to wait for their opponent to finish their current move. We now have a Real Time Strategy Game.

If we tried to play this out on a regular board game, things might come out of hand pretty quickly. First of all, it would be such a large board, that just moving the pieces around manually would be both problematic and exhausting.

Secondly, figuring out all the allowed actions for a given unit would require some intricate calculations, especially when confronting your opponent with a large army. The game would probably turn into a calculation battle more than a strategy game.

Another problem would be to decide who acted first, since movement is done in real-time and a lot of moves from both players would be executed at a fast pace. If two units of equal strength attack each other at the same time, would they both deal damage? Would a dice be thrown to decide the winner randomly? Luckily, all of these problems go away when the game is executed on a computer, with some simple rules and guidelines.

2.2 A More Precise Definition

The example in section 2.1 shows how we moved on from a simple turn-based chess game to a more sophisticated real-time chess game. Removing turns from the game made it real-time, and introducing some complex properties to the game units added some powerful means of using strategy to win the game. There

is however, no official definition of what is and what is not a real time strategy game. The games come plenty, and they come in many flavours. Generally, people would probably agree with Rob Pardo[4] that a satisfying definition of a real time strategy game is

> *A strategic game in which the primary mode of play is in a real-time setting. [Ada06]*

By this definition, our modified chess game fits perfectly into the genre, but so do also quite a lot of games that usually are not considered as real time strategy games. An example of such a game is the original SimCity, from the SimCity[Inc08] series. You execute your orders real-time and you have to plan carefully what to build next, in order to succeed in your missions. However, the game does not have any military decisions involved and thus differs from most other RTS games.

We extend our definition to also demand some sort of military part of the game, just as Dan Adams did when he proposed a modified version of Rob Pardo's definition in his web article "The State of the RTS"[Ada06].

Most RTS games have elements that involve some sort of base-building and resource management. The games we will discuss further in this thesis all relate to these concepts, so when we refer to an RTS Game, we will employ the following definition

> *A military strategy game with elements of base-building and resource management that unfolds in a real-time environment*

2.3 The Games

There are many RTS games available on the market today, whether it be fantasy, science fiction, a medieval setting or perhaps a realistic World War II combat. The games do however, vary a lot when it comes to quality and competetiveness. For a comprehensive list of available RTS games, take a look at [Fle08].

Enjoying a particular game usually boils down to a matter of personal taste, but some games have had more success than others. Some have even had a great impact on the professional gaming community. E-sports is the term used to describe electronic sports, i.e. computer games which are competed in at a professional level. By professional we mean full time dedication and players making a living out of playing computer games. Huge tournaments and even professional leagues, with hundreds of thousands of dollars in prize money, are hosted regularly, e.g. WCG[5]. Top players have huge fan bases, just as in regular sports, like soccer and basketball.

There are two games in particular that are worth mentioning, due to their appeal to the general audience and professionals. They represent two of the ten computer games that are on the list of official WCG games.

Warcraft III

> *WarCraft III is a finely tuned game and an amazing example of the importance of polish and presentation. [Ada02]*

[4]Rob Pardo is the executive vice precident of game design at Blizzard Entertainment [Ent09a]

[5]World Cyber Games, a global e-sports tournament, see [ICM]

This game hit the shelves in 2002 and the expansion pack, "The Frozen Throne", was released only a year later. It is the third game in its series, and has taken the audience by storm. The storyline is set to a mere 15 years after Warcraft II left off, and the game introduces two completely new and unique playable races[6]. The game differs somewhat from most other RTS games, even its own predecessors, in that it has a few RPG[7] elements in it. For instance, it introduced the concept of having special units, called heroes, that gain experience when defeating enemies. The accumulated experience lets the hero reach higher levels, where more powerful skills can be acquired, that benefits the player on the battlefield. In section 5.3 we will discuss WarGus, an open source clone of Warcraft II, which will be used as an experimental environment for solving the thesis problem (5.1).

Starcraft

> *Blizzard's sci-fi strategy war game Starcraft is hugely popular in Korea, with thousands of games taking place in computer cafés all over the country, and tournaments of such high profile that good players are able to make a living as professionals. In these tournaments, players, sponsored by technology companies and wearing trendy F1-style leather sportswear, compete on stage in huge arenas holding thousands of fans. [Mil07]*

Arguably the most popular RTS game ever to be released. Both the game and its expansion pack, Brood War, were released back in 1998. One of the reasons for its popularity is the three absolutely unique playable races. Even though the game shipped with initially uneven races, the game has gone through quite an immense number of patches, making the game extremely well-balanced.

The game may not have the best looking graphics, but it is nonetheless great for gaming at a professional level. In South Korea, Starcraft is currently the most popular game, with its own professional league and multiple TV stations, broadcasting Starcraft 24/7.

Even though Starcraft can be enjoyed by anyone, to become a proffesional player requires a deep understanding of the game, as well as the use of advanced and well thought through strategies. Additionally, a high intesity, high speed control of the keyboard and the mouse is needed. Some professionals have an average APM[8] of about 360 throughout a single game. That is an astonishing 6 unique orders every second. For more information on Starcraft in South Korea, see [Bel07].

2.4 Contents

Some of the concepts that describe the individual parts of a modern RTS game were put forth in section 2.1 and 2.2. Even though every game is unique in some way, a lot of the features, or ingame concepts, can be found in most of them. We will take a brief look at some of these features, all of which can be found in the two games mentioned in section 2.3, including WarGus, which will be introduced in section 5.3 and further discussed in section 7.

[6] See appendix A
[7] Role Playing Game
[8] Actions Per Minute - the number of actions or orders the player issues every minute

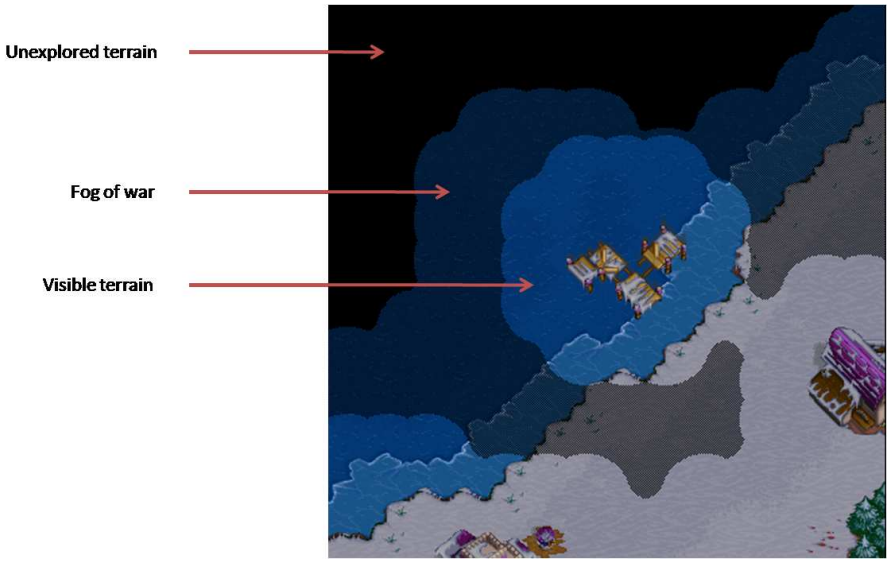

Figure 1: Fog of war in Warcraft II

World Representation The game plays out in some kind of virtual world, usually represented as a two-dimensional grid of tiles, much like ordinary board games (see chess example in section 2.1). How many tiles the game operates with depends much on the particular game you play. Usually this will be at least 64×64 tiles, as a single small unit usually takes up the space of one tile, whereas bigger units and buildings take up more. Even though newer games come with 3D graphics, the world will usually be divided into small regions like tiles.

You view the board from a distance in a top-down fashion, much like you would if you floated around on a cloud, watching down at cities nearby. Games with more advanced 3D graphics even allow for repositioning of the player view, so that you can choose the angle you feel most comfortable with.

Fog of War The fog of war is both an interesting and an important aspect of real time strategy games. The concept basically describes what can, and what cannot be seen by the player. The game world is initally covered in complete darkness.

Every unit, building and other objects a player can control, will light up a tiny portion of the world surrounding it. These regions are no longer unexplored, and can thus be seen by the player. When a region has been explored, it will never go back to complete darkness, but when there are no units left to light it up, a fog will cover that area.

The fog is thin enough to let you see the shape of the landscape the last time you explored the area, but covers for anything recent that has happened. See Figure 1.

Winning the Game To win a particular RTS game you have to defeat your opponent, and how you do this varies between different game styles. We can

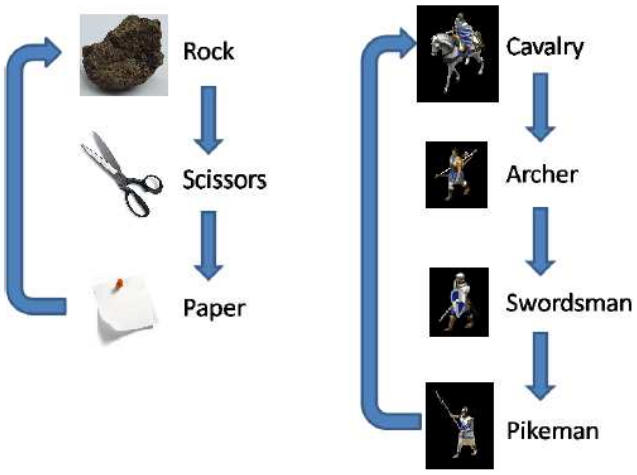

Figure 2: Unit counters in "Rock, Paper, Scissors" and Age of Empires II

differentiate between Single-Player Campaign and Multiplayer Skirmish. In the single-player mode, you follow a storyline, closely tied to the theme of that particular game. Some games can be won by holding off enemy attacks for a given amount of time, and in others you have to find and destroy specific buildings or units.

The multiplayer skirmish mode also offers several ways to victory, but the most common one is to destroy all opponent units and structures. So when all your structures and units have been lost, you lose the game.

The name multiplayer skirmish might be a bit misleading as you can play this type of game with as few as only one opponent, and that opponent can even be the computer AI.

Units, Buildings and Resources So how do you destroy your opponents? Disregarding tactics and the actual use of the individual units, you would need a bigger army, or one that consists of a better unit combination. To acquire this, you would need information about your opponent (see section 2.4) and an understanding of how units match up against each other in the particular game you are playing.

Units are usually designed in a similar fashion to the well known "Rock, Paper, Scissors"[9] game. A great example of a game with such a balance are the units in the Age of Empires series [Mic99]. Cavalry will be great against ranged archers with their excellent speed and archers will easily stop slow close-combat swordsmen from a great distance. The swordsmen can outperform the even slower and less armored pikemen, but the pikemen are a great choice against cavalry. See Figure 2.

Alternatively, you can try to outproduce your opponent. To do this, you will need to gather more resources and use them more wisely, or even faster. Resources are an essential aspect of the game, as you purchase everything ingame

[9]See [Soc09] for more details on Rock, Paper, Scissors

with the resources you gather. What kind of resources and how you collect them varies greatly from game to game, but money, gold, natural resources like wood, stone and oil are pretty common types. The resources can be found at different locations in the virtual game world, and some sort of unit, usually called a worker, will be used to bring resources from these locations back to the main base.

Strategy A lot of strategies take advantage of game specific weaknesses or imbalancies and are thus game dependent. There are however, a few ones that can be deployed in almost any game. We will present the most common one and also take a brief look at micro management, which is a concept that is closely related to the strategy element of the game.

The "Rush" plays out in the very beginning of a game. Instead of focusing on building a solid base, defences and establishing a steady economy, you "rush" to some of the most basic warfare units available, and immediately launch an attack on your opponent.

A successfull rush can lead to instant victory, or a highly reduced opponent, putting you in the lead for the rest of the game. However, a failure can lead to quite the opposite results. If you fail to inflict sufficient damage, you will most likely be in a position where your opponent has an economic advantage, leaving him in the lead position. He might even counter-attack you, and end the game right away.

The rush might be a risky move, but it adds to the strategical aspect of the game, forcing you to always stay alert, planning on what to do next, exploiting information and executing the right decisions.

Micro Management is the art of controlling individual units as to gain an additional advantage on your opponent. Most games usually have a simple unit specific AI that helps you out when you give orders like; "move from your current location to tile (4, 5)" or "hold position". The unit will then move from its current position to the place you selected on its own. If any obstacles are in the way, the computer will try to find a way around it, making it seem like the unit is somewhat intelligent.

When in battle with an entire army, units will start attacking each other automatically, since this is what the unit AI has defined as its default reaction (see section 6.1.2). It might just be the best move to attack your opponent's army, but letting every single one of your units attack a completely random opponent unit might not be. Positioning, choosing the correct opponent unit for each individual unit (See Figure 2) and even exploiting terrain features can greatly affect the battle. Twelve swordsmen might face a loss if they meet up against ten far more coordinated swordsmen of the same type.

2.5 History

The RTS genre has been in constant development since it first saw daylight back in the late eighties. Bruce Geryk[Gera, Gerb] and Mark H. Walker[Wal09] both describes the journey through the Real Time Strategy history in several parts. Consult their work for more information on the topic. Sindre Berg Stene[Ste06]

also has a short history section in his thesis "Artificial Intelligence Techniques in Real-Time Strategy Games - Achitecture and Combat Behavior".

3 A Brief Review of Reinforcement Learning

It is not hard to learn more. What is hard is to unlearn when you discover yourself wrong.

— Martin H. Fischer

Reinforcement learning is a broad concept that applies experience based learning to various applications and concerns how agents act in the application domain by trying to maximize some reward in the long term. It consists of a reinforcement function, which gives feedback when actions are selected, and an exploration strategy, which tells how and how often random actions should be explored.

3.1 Markov Decision Processes

It is convenient to model the problem domain as a finite-state Markov decision process. This is a mathematical concept that can be used to model decision-making, where outcomes are partially random and partially based on decisions of some decision-maker.

We have a state space S, where a state can be regarded as a unique configuration of information in the problem domain, and an action space A, which defines the set of possible actions to perform. We move from one state to another by selecting a particular action. The probability when transitioning from state s to s' is given by $P_a(s, s')$ and the reward by $R_a(s, s')$, where a is the selected action. The problem can now be represented as a list of 4-tuples on the form (s, a, P_a, R_a).

The action space A does not have to be the same for every state and might thus be expressed as A_e. See section 8.1.2 on page 58.

Policy The goal of a Markov Decision Process is to find a policy function that map from states to actions. This can be expressed as

$$\pi : S \rightarrow A$$

which defines the action $\pi(s)$ to choose when in state s. With each action fixed for each possible state, the MDP/policy combination thus behaves like a Markov chain, which in AI fields is usually used for prediction[Mil06].

See [RN03] for more information on variations of such MDP policies.

Observability The problem domain, and hence how the states are represented varies between different problems. When an action a is selected in the state s, the resulting state s' may or may not be known at the given time. If it is known, the MDP is said to have full observability. The policy, or solution, assumes full observability to allow for the $\pi(s)$ to be calculated.

If the next state cannot be known in advance, the MDP has partial observability, usually called a POMDP. See [EED] for more information.

3.2 Q-Learning

Q-learning is a technique that learns an action-value function that describes the expected utility of selecting a particular action in a given state. It does so without the need of a problem domain model.

With a finite set of states, we can use a simple two-dimensional array to store our state-action tuples. A single cell thus represents choosing the action a_x in the current state s_y. The value contained in each cell represents the Q-value, a measure of how good this particular action is for the current state.

The algortihm The Q-table is updated by an iterative algorithm and can be expressed as

$$Q\left(s,a\right) = Q\left(s,a\right) + \alpha\left(R\left(s,a\right) + \gamma \max_{a'} Q\left(s',a'\right) - Q\left(s,a\right)\right) \tag{1}$$

where γ is the discount factor and α the learning rate, s is the current state, a the selected action and s' the state a leads to. The value $\max_{a'} Q\left(s',a'\right)$ is given by the best possible action a' from state s'.

The discount factor, γ, describes how much future Q-values should affect the current Q-value calculation. A higher value makes the agent strive for higher long-term rewards.

The learning rate, α, describes how much newly acquired information should influence the older ones. $\alpha = 0$ would indicate no learning at all and $\alpha = 1$ to only use the most recent information.

The optimal policy π^*, is a policy that maximizes the expected total reward, which in the case of Q-learning means learning an optimal policy for the given problem. We can write it as

$$\pi^*(s) = \arg\max_a [Q(s,a)]$$

where s is the current state and a the selected action.

In Algorithm 1 we present a simple pseudo-code for solving a Q-Learning problem. We have a simple Q-table which we initialize as a zero matrix with the size of the action space in two dimensions. Then we decide how many iterations we would like to run, since an infinite amount would be rather impractical for our needs. This number may vary from problem to problem, but generally a higher number would increase the learning potential and increase the chance of convergence.

Next, we select an initial state $s \in S$. For a lot of problems, there might only be one valid start state (8.1.2), but for others, any initial state may be chosen.

The next steps are repeated until the terminal state is found. Not all problems may have such a terminal state, so a number that defines the maximum amount of repetitions allowed can thus be used.

The first one of these repetetive steps is to select an action $a \in A_e$. A random value ρ, where $0 \leq \rho \leq 1$, could be introduced so that every now and then, a completely random action is selected (3.3). When a random action is not chosen, the currently best known can be used.

The selected action leads to a new state s', and for this state we find the maxmimum Q-value q by iterating all the possible actions and keeping track of the one with the current highest value. When the maximum Q-value has been found, equation (1) can be used to calculate the new Q-value for the currently selected state-action tuple.

A transition to the next state is made, and the procedure repeats as described above. A random value ν could also be introduced, where $0 \leq \nu \leq 1$, so that every now and then a random new state is picked instead of the next one.

Algorithm 1 Q-learning

```
1: for all iterations of the problem do
2:     select initial state s
3:     repeat
4:         select a possible action a
5:         find the state s' this action leads to
6:         get the maxmimum Q-value q from this state
7:         calculate ((1))
8:         current state = s'
9:     until either a terminal state or a given number of repetitions are met
10: end for
```

In figure 3 we have illustrated an example problem as a simple Finite State Automata. A house has five separate rooms, only connected by doors that can be opened from both directions. A treasure is located in room C and the agent's job is to learn the shortest path to the treasure from any room in the house. A state thus represents being in a particular room, and selecting an action implies walking through a particular door.

The numbered labels indicate the given reward for choosing a particular action. With a 5×5 Q-table we can now easily run algorithm 1 to solve our problem.

3.3 Exploration vs Exploitation

The results of the Q-learning algorithm will be highly influenced by how we choose to explore and exploit [RN03]. By exploring we refer to the concept of choosing random actions, leading us to histories we may not have encountered before. Exploiting is more of the opposite, the concept of using the currently best known actions.

If you always choose the best action found so far, you might end up being in quite a sparse amount of states as you never try something new. As a consequence, a solution far from optimal may be found, or maybe none at all, depending on your problem domain. To avoid this, we would like to pick a random action every now and then, based on some criteria. We will look at two

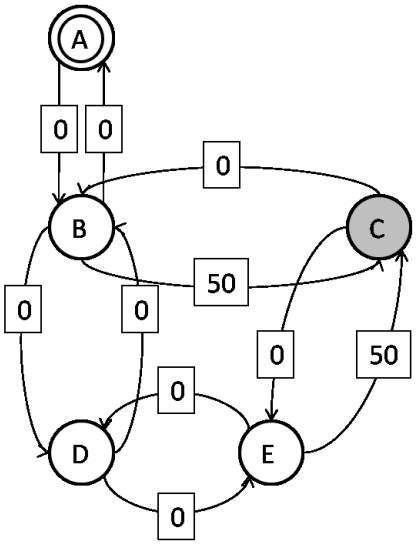

Figure 3: A Q-learning example shown as a Finite State Automata

different exploration strategies that we will be discussing later in this thesis. See section 8.4 on page 66.

Constant Randomness This strategy propose that when actions are selected by the decision maker, it selects a random one with the probability $0 \leq \rho \leq 1$. Each valid action $a \in A_e$ will be given the same probability of being selected. In the case of not using a random action, the best learned action so far is indicated by the highest Q-value for the current state, and will thus be used.

Boltzmann Exploration A different approach would be to give each action a weighted probability for being drawn, based upon how great their Q-values are, and how late in the simulation we are. This can be achieved by using Boltzmann probabilities [Mil06].

We would like to explore more in the early phases, due to the fact that states are still unexplored and the Q-table is not yet representative of optimal choices. As the training phase progresses, the values get more and more trustworthy, and we can thus choose the best known actions more often.

Part III

Litterature and Research

4 State of the art in Computer Game AI

Intelligence is what you use when you don't know what to do.

— *Jean Piaget*

In this section we will review the current state of research on computer game AI. As there is a vast number of different techniques available to AI programmers today, only a handful of the most relevant ones will be covered.

We have divided the section into three categories

- **Decision Making:** Techniques used to decide which actions to perform when faced with ingame choices
- **Learning:** Learning algorithms and concepts that can be used to create self improving decision making tools
- **RTS Related Research:** Various research papers on the RTS domain, or closely related

4.1 Decision Making

When an RTS worker, peacefully carrying a lump of gold back from the gold mine, is attacked by an enemy unit, it can choose to flee or retaliate. A human player or some sort of low level game engine specific AI agent, must make this decision.

For game AIs, often referred to as Computer Controllable Players, or CCPs, decisions are made by a decision making expert. It selects an action to perform based on information in the current game state. Structures, resources, unit placement or even opponent behaviour can be used to influence the decision making.

We here present some of the most common decision making techniques in game AI.

4.1.1 Decision Trees

A decision tree [RN03] consists of a root, one or more decision points and leaf nodes that describe the actions to be carried out. To decide which action to perform, we start at the root and follow the path led by each decision point's answer. When we ultimately arrive at a leaf node, we have found the desired action to perform.

We can illustrate this by a simple example. In figure 4 on the next page we show a possible decision tree for an archer in the game Age of Empires 2 [Mic99]. An enemy is visible, so the archer must decide how to react. The first question is whether the enemy is in range or not. Let us say it currently is, so we follow the path of yes to the next decision point, which asks if the enemy is close. In a real implementation, the term "close" would have to be less vague, but for the sake of this example, it serves the purpose.

The archer is facing a nearby swordsman, so we follow the yes branch yet again. Now we have arrived at a leaf node, leaving us with the conclusion that we should shoot and move, which seems like a reasonable thing to do as firing and repositioning will both inflict damage and reduce the chance of being dealt damage to.

In this example, the decision tree is binary, with only two possible outcomes for each decision point. This is not the only way to design a decision tree, as it can have a multitude of branches and can take on any type of questions, e.g. booleans or 3D vectors [Mil06].

Decision trees can also be used for learning purposes [fAI08, Fu03]. A very common easy-to-implement TD algorithm is ID3, which provides high efficiency at the same time as being very flexible [Bae06].

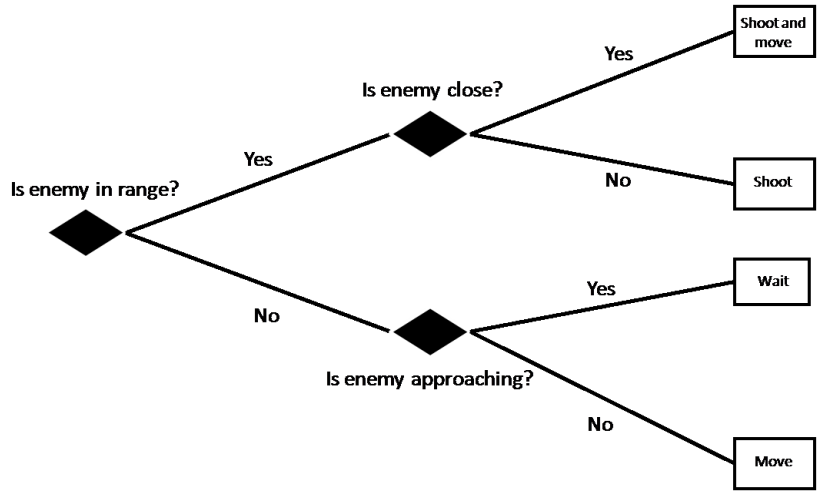

Figure 4: A hypothetical decision tree for an archer in Age of Empires 2

4.1.2 Finite State Machines

In a finite state machine [FH03, Mil06] you assign each unit or character a current state to be in. While in this state, the unit can work on a set of inputs and either perform some action specified by the given state, or transition to a new state [Buc05]. A state could be of almost any type, e.g. attacking an enemy, harvesting wood in the forest or even the act of doing nothing, known as idling.

Usually, the state transition occurs when a set of specific input conditions, called triigers, are met [Bro09]. This is like flicking a switch for an ordinary light bulb. If the light is off when the switch is flicked, it allows electricity to flow through the system (input), the light is turned on (action) and the current state is changed to "On" (state transition).

In Figure 5 on the following page we see a simple finite state machine for a worker in an RTS game. From the start state it automatically transitions to the idle state, where it remains until all of the "need resources" conditions are met.

These will typically be met whenever the game engine signals that resources are needed in order to perform some higher level task.

When the gold has been collected and the worker leaves the gold mine, a transition will be triggered by the game engine, switching the current state to "return to base". This state makes sure the worker gets home safely with the goods.

When the gold has been dropped off, a new transition occurs and the worker is now back to the idle state again, waiting for a new order. In this simple example, the worker can only perform one task. However, in a more realistic setting, there would be a multitude of states, conditions and possible transitions that could occur.

In [Sho06], FSMs were used to make RTS workers (also known as citizen) repair, construct buildings and gather resources. Constructing a single building is a multi-step operation which can be represented by a simple FSM.

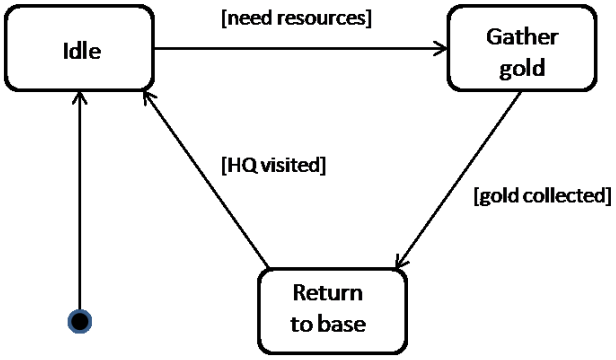

Figure 5: A simple finite state machine for a worker in an RTS game

4.1.3 Fuzzy Logic

It is not always clear in what state a unit should be in or what kind of transition should be made when some conditions are met. Whether a unit is in a healthy or a hurt state really boils down to the personal consideration of the game developer. However, with fuzzy logic [McC00], the unit can be both healthy and hurt to a certain degree at the same time. Instead of being either one or the other, the unit can partially belong to so called fuzzy sets with numerical values called degrees of membership.

Membership functions are used to map input values to degree of membership values, i.e. for each set (healthy, hurt, highEnergy, lowEnergy, ...) a separate function describes how much an input value belongs to the given set. These functions can vary greatly and will have a great impact on the end results [Mil06].

In Figure 6 on the next page we have illustrated two sets, highEnergy and lowEnergy for a mage[10] in an RTS game. We can see that the mage has a partial membership of the highEnergy and lowEnergy sets of approx. 0.8 and 0.6, respectively.

[10] A magician or sorcerer

With the mapped values now at hand, fuzzy rules [Cha04] can be applied. These rules can tie together and create new membership values for other sets. As an example, regard the statement

> *"If I am close to the enemy, and I have high energy, then I should use a magic spell"*

With degree of membership values calculated for highEnergy and enemy closeness, we can calculate the value for using a magical spell.

Now we need to turn the values back into useful data. This is done by using a "defuzzification" technique [Buc05], a bit less obvious than the membership functions that helped us "fuzzify" the input values in the first place. This process involves turning a set of membership values into a single output value, which then can be used to decide which state to transition to.

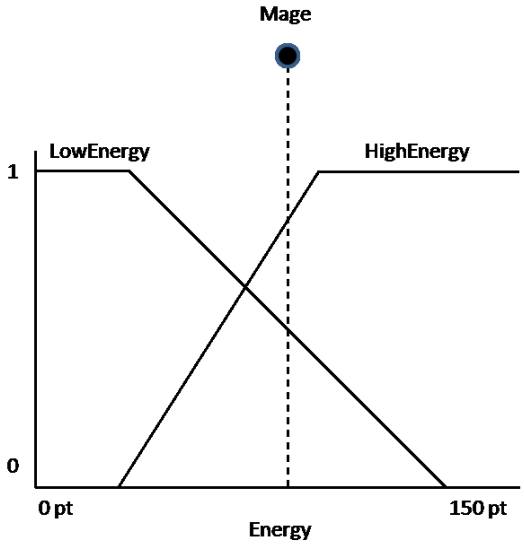

Figure 6: Fuzzy logic membership functions for the energy amount of a mage

4.1.4 Blackboard Architecture

Blackboard Architecture [IB02] is not a decision making technique in itself, but more of a concept that can be used to combine and choose what to do, based on a multitude of different decision makers. It makes use of the following three parts; experts, a blackboard and an arbiter.

The experts can be of any type, using any internal architecture, e.g., Decision Trees (4.1.1), FSMs (4.1.2) or Bayesian Networks (4.2.7). An example of experts in an RTS game (see section 6) communicating with the help of a blackboard solution is shown in Figure 7 on the following page.

Experts can ask the arbiter for read and write access to the blackboard, which is a shared part of the memory. How variables and data are stored and represented depends on the particular implementation. Blackboards come in two flavors, either static or dynamic, where a static blackboard only shares a

static amount of predertermined data, whereas a dynamic blackboard does not have this constraint [Ork02].

The arbiter is the piece of code that decides which expert should be allowed to use the blackboard, as only one expert should be modifying it at a single point in time. Each of the experts, and the blackboard itself, may be implemented as separate threads for Multiprocessor Machines [Bay08].

The selection algorithm used by the arbiter varies between implementations, but a simple solution is to let every expert express their interest in working on and modifying the blackboard by providing a numerical "interest value", and then selecting the expert with the highest indication of interest [Mil06].

The blackboard architecture procedure can be summarized by the following three steps

- Each expert takes a look at the blackboard and indicates its interest
- Based on some selection algorithm, the arbiter chooses an expert
- The expert may modify the blackboard and when done, release control

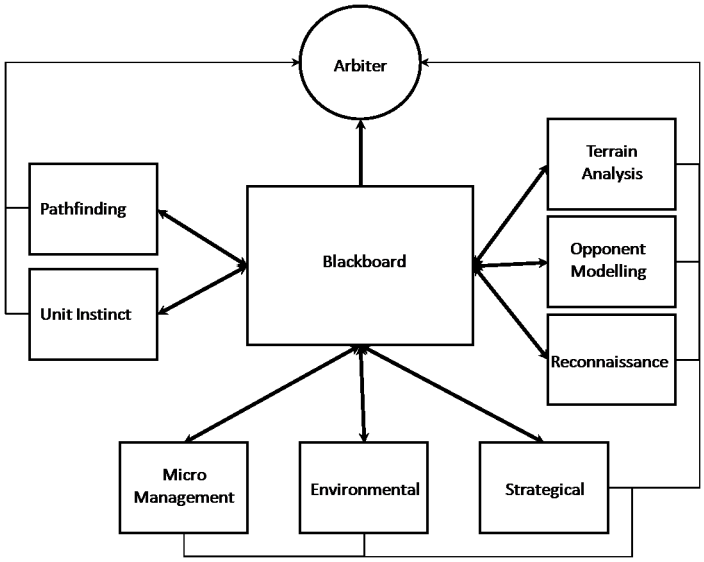

Figure 7: An example of a blackboard architecture for the AI levels discussed in section 6 on page 36

4.2 Learning

> *Learning is an active process. We learn by doing.. Only knowledge that is used sticks in your mind.*
>
> — *Dale Carnegie*

Letting two unchangeable predefined CCPs compete in a number of games against each other would most likely end up in a set of identical battles, maybe with a few exceptions due to random ingame events.

If we as humans were facing either of these CCP opponents, it is likely to believe that we could lose a few times. This will last only for a short period of time, as we would start adapting our game style and strategies due to our ability to recognize patterns, learn, infer and predict, even over the course of a few games.

A general problem with learning algorithms is avoiding over-learning. Over-learning is the concept of being exposed so much to certain events, that the learner ends up learning only responses to those events [Mil06].

4.2.1 Supervised Learning

Supervised learning [RN03], [Kot06] is a machine learning technique where a function is learned from inputs and outputs. The purpose of this function is to be able to predict the correct output to a given input, after having seen a number of training examples.

This can be compared to the way we humans learn when studying for example mathematics. We open our book, find a problem we want to solve, puzzle for a couple of minutes, and then come up with a solution. We can then check if our solution was correct, either by looking in the solution section of the book, or by asking the teacher.

Supervised learning is often used in Neural networks, by using backpropagation in feedforwarded networks. See section 4.2.6 on page 26.

4.2.2 Unsupervised Learning

In unsupervised learning [GH99], the output is not available for the specific input, so we cannot say that a given output is correct or not. Instead, input patterns are learned, so that when faced with some new input, it will be determined to belong to a class.

This can be compared to the way we humans learn to separate between apples and oranges. Even before we know their names we are able to differentiate between them, by recognizing that apples belong to a given group and oranges to another.

Clustering is an important unsupervised learning problem and deals with recognizing a structure in an unlabelled collection of data and organizing objects into groups with members who are somewhat similar. See [Fun01] for more on clustering.

4.2.3 Reinforcement Learning

> *I am always ready to learn although I do not always like being taught.*
>
> — *Winston Churchill*

Another learning type is reinforcement learning [RN03], which tries to maximize the expected total reward. An agent uses trial and error by selecting actions to try out, sometimes random ones, sometimes the best currently known. For each chosen action, a corresponding reward is given. A high reward means that the action led to something positive and a low value means something negative.

See section 3 on page 14 for a more detailed explanation of reinforcement learning.

4.2.4 Online and Offline Learning

Independent of the learning techniques used, learning can be applied either online or offline. Online learning is the concept of performing learning while the player is playing the game. This may allow the player to adapt better to the opponent and pose a greater challenge. The more a player plays, the better the predictions can be made by a computer player.

Offline learning is done between games, or possibly in between levels if the game possesses such a concept. Most games which incorporate some sort of learning uses an offline approach [Mil06]. This is mostly due to the cost in terms of computational power of these algorithms.

4.2.5 Genetic Algorithms

> *It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.*
>
> — *Charles Darwin*

Genetic algorithms try to simulate the process we see in nature, more commonly known as evolution[11]. It involves a set of "chromosomes", representing a possible solution to a problem, that undergo mutations, replication and competition.

The chromosomes are generally encoded as a sequence of binary numbers, making it easy to manipulate them. Starting out with a set of initially random chromosomes, we calculate a fitness score using a fitness function. The least fit chromosomes are not allowed to reproduce, and some of the more fit ones may actually be allowed to create more than one offspring.

Based on the values given by the fitness function, we select two and two chromosomes from the remaining gene pool. With a fairly high probability, we use a crossover operator to create two new chromosomes from the "parent chromosomes". The new ones are added and the old ones are removed. An example of creating such chromosomes is illustrated in Figure 8 on the next page, where we have chromosomes consisting of 10 genes, each represented by a two digit binary value.

Then, with a small probability, we mutate some of the chromosomes. Mutation operators come in many flavors, but a simple way to do this is by selecting two random genes in a single chromosome and switch their positions.

The process of killing, mutating and reproducing is done when either a fit solution is found or a given amount of time has elapsed. Choosing the probability parameters and algorithm related constants, like the chromosome length and the population size, may vary from problem to problem and follow no definite rules.

In [MP04] genetic algorithms were successfully used as an offline learning technique to find strategies to beat two different types of "rushing opponent players", in the game Wargus (see section 5.3.4). A complete game-AI script was encoded as the chromosome, where each gene represented a single rule. Different genes corresponded to different rules, like building genes, research genes, economy genes and combat genes. In addtion, four different genetic operators were implemented and used to create new chromosomes.

See [Buc02] and [RN03] for more on genetic algorithms.

[11] See [Tal09] for a collection of resources on evolution

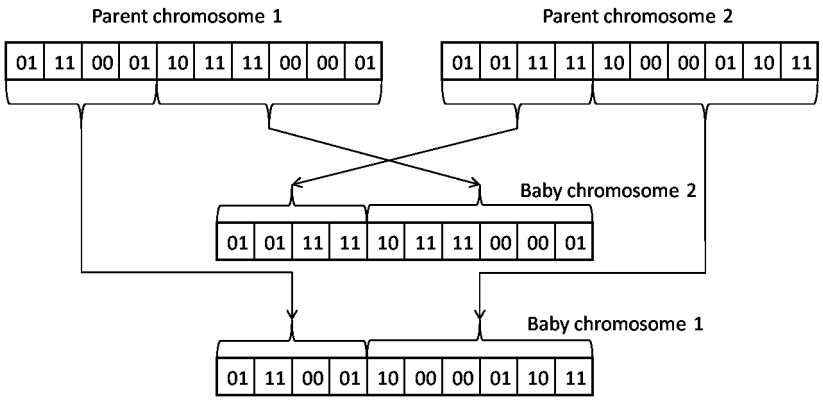

Figure 8: Creation of new chromosomes in a genetic algorithm with the crossover parameter set to 4

4.2.6 Neural Networks

Neural networks [RN03], [Buc02] try to simulate the structure and functionality of biological neural networks, by using small building blocks called artificial neurons. These neurons are connected together in a network by directed links. The number of neurons used in a given network may vary from problem to problem, but generally the more neurons, the slower and more complex the network gets.

Each neuron has a number of inputs, an activation function and an output value. Input and output values are usually either 1 or 0, symbolizing that the neuron "fires" or not, respectively.

Each of the inputs a_j have a corresponding weight $W_{j,i}$ associated to it, which are responsible for the overall activity of a given neuron. For each neuron, the following sum is calculated

$$s_i = \sum_{j=0}^{n} W_{j,i} a_j$$

and is handed over to the activation function. The activation function then decides whether the neuron should fire or not, e.g. if the calculated activation value is over some given threshold.

The activation value can be forwarded to new neurons, located in a so called hidden layers. In such a network, output values from neurons in one level are fed as input values into the neurons of the next level.

With backpropagation [Ber05] and supervised learning, we may train our network to recognize patterns. In a sequence, we show a number of patterns to the network. For a given pattern, we calculate the output value, σ, for the whole network. Then, with the use of a given target output value, τ, we can calculate an error between σ and τ.

With this error at hand, we can adjust the input weights of the output layer, so the next time a similar pattern is presented, the output is a bit closer to the

target output. The same process is then propagated backwards through the network, level by level, as shown in Figure 9.

When errors reach an acceptable level, the network is considered fully trained.

In [Smi01] neural networks were used to improve a CCP in a simple RTS game. After the neural network had been successfully trained, it showed that tank units had higher weight values than factories, giving a priority to build tanks instead of factories. Additionally, units were learned to move close in on the enemy forces. But right before they got close enough for a confrontation to escalate, they were retreated. The result was scattered enemy forces.

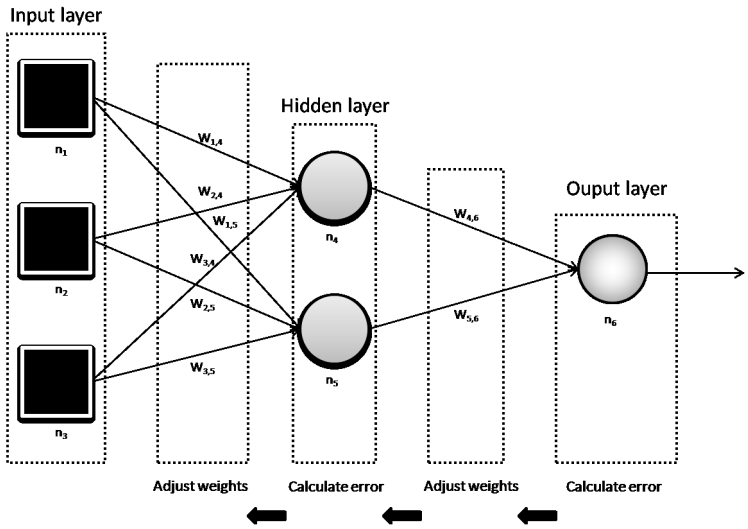

Figure 9: A simple feed forward neural network with backpropagation

4.2.7 Bayesian Networks

Bayesian networks [RN03] are DAGs, directed acyclic graphs, with nodes representing random variables, used in domains that are dominated by uncertainty.

The archs illustrate the relationship between the variables, so if we have an arrow from node X to node Y, we say that X is the parent of Y, and that X has a direct influence on Y.

We can use Baye's rule [Ric94]

$$p(A|B) = \frac{p(A,B)}{p(B)} = \frac{p(B|A)p(A)}{p(B)}$$

to find the probability of being in a state, given knowledge about the parents. That is, if some of the variables are unknown, they can be inferred. Each node X_i has a conditional probability distribution, given by $\mathrm{P}(X_i \mid \mathrm{Parents}(X_i))$. If we have k parental nodes, the CPT[12] will contain 2^k independently specifiable probabilities.

[12]Conditional probability tables

Scouting an enemy base, as important as it is in an RTS game, can be a case of bayesian networks. When certain buildings are identified, the likelyhood of other buildings may be inferred, based upon that player's previous actions [Bae06]

In [Wal06], Bayesian networks were used to create a resource management AI. By introducing the random variables WhichUnitIsNeeded, UnitOrWorker, SoldierOrBarrack and TankOrFactory, their corresponding conditional probability tables were trained and calculated. Using different learning algorithms for training the tables, some of the experiments showed promising results, outperforming a manually created AI.

4.3 RTS Related Research

The current AI performance in RTS games is quite poor [Bur04]. However, RTS related topics are being researched and advanced on multiple fronts. In addition, new games are regularly released, providing both new and improved solutions. We will here present some of the recent studies that have been made in the RTS domain.

4.3.1 Pathfinding

In [EF08], the efficiency and suitability for various pathfinding algorithms was investigated in the narrow domain of mobile system RTS games. These devices are severly limited in terms of processing power and storage capabilities and therefore pose a great design problem for RTS game developers

The triangle-based TRA* algorithm was implemented on a Nintendo DS and alongside the Minimal Memory solution, they both showed promising results. See section 6.1.1 on page 36 for an explanation of pathfinding in RTS games.

4.3.2 Terrain Analysis

An AI with situation aware agents, being able to form a playable strategy for the TA Spring project (5.3.3) was developed in [Ste06]. With the use of threat maps, i.e. influence maps able to describe how much attack damage the opponent player can do to the surrounding tiles, the AI was able to improve its overall game performance by moving units more strategically.

The AI also uses a simplified version of the K-means algorithm to create an abstraction of where the opponent bases are located. As well as using path planning to calculate a detailed attack route, it also handles micro management on a fairly simple level (6.2.4). See section 6.2.1 on page 38 for more on terrain analysis.

4.3.3 Opponent Modelling

In [FSS07], a hierarchical approach to modelling the opponent's strategy was developed for the TA Spring project (5.3.3). Various strategies were put into subcategories of either aggressive or defensive. With a set of confidence values updated based on observable game information during the game, the AI had an estimate of the probability that an opponent player was using a specific strategy.

Experimental results showed that the AI was able to accurately distinguish between aggressive and defensive players, and between subcategories in later stages of the game. See section 6.2.2 on page 38 for more on opponent modelling.

4.3.4 Rushing Strategy

[Li08] proposes a way to learn and improve a rush strategy by using an advanced point system. RTS games usually calculate a set of "points" during the game, shown to the player when the game has ended. These points reflect how well the player has performed, in terms of units destroyed, structures razed and the total income.

Decision making will be influenced by the calculated points and form a strategy. As a strategy with higher amount of points is assumed to have a greater likelyhood of winning, the AI tries to maximize the total amount of points over a series of games. See section 2.4 on page 9 for a brief explanation of rushing in RTS games.

Part IV

Analysis

5 Analysis of the Problem

> *The question of whether a computer can think is no more interesting than the question of whether a submarine can swim*
>
> — *Dijkstra*

This section provides as a guide through the problem emphasized in this thesis. We will first state the definition of the problem we attempt to solve. We then discuss some of the similar work that have been done and address why this problem is worth answering. The section ends with a discussion of frameworks where we present some of the alternatives and select the most appropriate one.

5.1 Stating The Problem

Our problem definition for this research is as follows

Can reinforcement learning be used to improve the decision making of a computer controllable player AI in an RTS game, so as to adapt to the opponent and learn winning strategies? How will exploration strategies and different opponents influence the learning performance?

5.2 Similar Work and Approaches

In section 4 we presented some general techniques found in game AI, as well as RTS related research made by others. This type of research often concentrate on parts of the AI that does not involve learning, like pathfinding, opponent modelling and terrain analysis (6). These particular fields may help improve the overall functionality of AI players, but for our problem they are irrelevant as they do not directly influence the learning part of the AI.

We therefore take a look at the research made on learning in RTS games. A bayesian network, an advanced point system and a genetic evolutionary approach to implementing an RTS AI all showed promising results and that learning in RTS games can be achieved. However, the advanced point system utilizes "game points", a measure of how good the various entities are. These points are decided by the game designer and does not have to reflect the actual value they serve in the game. The concept is thus limited to only consider those decisions that are connected to these points.

Reinforcement learning in RTS games is not a common tool of the trade. This may be because of the complexity these games introduce in terms of entities, positions and properties with respect to how reinforcement learning models the problem domain. We therefore wish to expand on knowledge by advancing a known position in a new direction. We do this by both applying known techniques, like Q-learning and reinforcement learning related exploration strategies (8.4), and unknown techniques, like concentrating on build orders (6.3) and unique heuristic algorithms (8.4.4), to a new matter of inquiry.

5.3 Selecting Framework

In section 5.1 we stated that we will develop a CCP AI with learning capabilities. If the AI is to be useful, or able to do anything at all, it must be implemented

in some kind of framework, or at least communicate with some RTS game. As there are several options for such a framework, we will here briefly discuss four of the most promising possibilities. For each framework, we will summarizing the main points both for and against using that particular type further in this project, as our working platform.

5.3.1 Develop a new RTS game

Released games are usually closed when it comes to the source code, meaning that you as a customer only have access to the executable. This makes it extremely hard, or even impossible, for an external developer to create an AI for that particular game. If the game comes bundled with an AI, you as a human can play against it, but creating your own AI and hooking it into the game is generally not possible, as there are no APIs[13]. This opens the door for creating an RTS game from scratch.

There are some positive aspects of devloping a game from scratch. We are fully in control of the game details, e.g. units, structures and resources. How a single map is represented and the general game flow can be constructed just the way we want it. It allows for flexibility, in that both the AI and the game engine can be constructed in such a way that they can interact with each other, using our own APIs and protocols. We can freely choose the supported operating systems, preferable programming languages, development IDEs, and hardware requirements.

On a positive note, graphics could be kept at a low level, if desired, as it is not very important for the overall results we seek. Learning either OpenGL or DirectX would require using an unnecessary amount of time when the project is already under hard time constraints. The development would be extremely time consuming as there are a vast number of details that need to be implemented so the game can behave similar to other well known RTS games, i.e. be a fully functional RTS game according to the definition in section 2.2 on page 7.

5.3.2 ORTS

ORTS, or Open Real Time Strategy, is a free open source game engine for RTS games. The framework is aimed at researchers as a test-bed for their research, in problem domains such as pathfinding, dealing with imperfect information and planning[Bur09].

Most AIs are implemented as a part of the game engine and thus operates within a given time slice. This slice may lie between computations that are a necessity to keep the performance at a high level. If the AI computations are processor-heavy, it may therefore reduce the game experience for any human players. In fact, this is not entirely fair, or even "real time" at all, as the CCP AI essentially pauses the game every time it operates. In ORTS, the game engine is located on a server, while both human and AI players must run on a client, forcing anyone in the game to operate under the same conditions.

The AI can be developed separate and without any particular knowledge of the game engine, communicating through APIs. ORTS even allows for super-computers to be connected to the game server.

[13] Application programming interfaces

On a positive note, any part of the game can be defined and modified, including the graphics. However, time must be invested in designing a game or modifying an existing open source version, running on the ORTS platform, as users define their own games in ORTS.

In section 6 on page 36 we discussed some of the problem domains a CCP AI can or must cover. With the ORTS framework we would need to implement several AI features that are beyond the scope of this thesis to be able to investigate our own problem domain regarding build orders (6.3).

5.3.3 TA Spring Project

The TA Spring Project is another free open source game engine, that was originally intended to bring the experience of the widely popular RTS game, Total Annihilation[Cav97], back to life. Today, the spring engine works as a pure game engine, supporting a great variety of game modules [Yan09].

With a physics engine allowing deformable terrain, reflective bump mapped water and realistic weapon trajectories, the Spring project is with no doubt the free framework alternative with most breathtaking graphics.

It supports development of fairly general AI components through their APIs, allowing them to be competent in multiple game mods on top of the game engine.

The framework was used as a test-platform in [Ste06]. Additional time was devoted to the development, so almost all game features could be supported. Like in the ORTS framework, the AI must support more features than necessary to function properly. The project is already on a tight schedule, so avoiding time consuming solutions is of great benefit.

5.3.4 Stratagus

Stratagus is the third free open source game engine possibility. The engine supports a variety of games, but they all inherit the concepts of the popular game Warcraft II[Ent09d]. One game in particular that inherits this concept is the open source game module Wargus[Tea07b]. Wargus is currently the only fully completed game module for Stratagus, and is a clone of the original Warcraft II game.

Using the original graphic contents of Warcraft II, Wargus both looks and feels just like the original, providing us with a framework as close to a commercial RTS game as possible.

Stratagus game modules are scripted in Lua, making it very simple to either modify contents of a game, or add features to it.

Wargus comes with an incorporated opponent CCP AI, defined as a simple script with order instructions. This AI makes use of a large AI framework inside the stratagus game engine. It is to some extent a simple "goal-driven" AI, executing the orders described in the script. As an example, if the script indicates that the AI should build a barracks, a structure production order of one barracks is issued to the AI framework order queue. The environmental level AI (6.2.6) makes new orders, so that enough resources are collected to construct the building. With such an AI framework at hand, we can hook into some of the already existing functionality, leaving us able to focus on our main problem, the build order concept as stated in 5.1 on page 31.

As opposed to the ORTS project, the AI is given a time slice every game cycle, in between rendering graphics and calculating all the game state details. In essence, the game is paused when the AI is making its moves, and thus violates to a certain degree with the concept of reacting and making decisions "real time".

5.3.5 Framework Conclusion

We have presented four different choices for creating or selecting a framework to implement our AI in. Even though we get the greatest sense of control and flexibility by creating a game engine from scratch, implementing all the needed functionalities to meet our definition in section 2.2, is by far the most time consuming alternative.

Both ORTS and TA Spring Project are a bit more "developer friendly" than Stratagus, having separated the AI part from the game engine itself. In addition, the real time concept is more fair in these frameworks than in Stratagus.

However, this problem does not really concern us. In section 5.1 on page 31 we stated that reinforcement learning will be the concept we use to investigate the thesis problem. The reinforcement learning algorithm is very processor friendly, so the effects of letting the AI use its time slice in a "paused game setting" is essentially the same as if each AI were connected to a server engine.

Stratagus, with its great AI framework, offers a great advantage over the other frameworks. Being able to use parts of the Environmental Level AI, in addition to the game's own Unit Behaviour and Pathfinding Level AI, makes it unnecessary to develop these parts.

Additional benefits of using Stratagus is that it has a very clean and well documented code and can be compiled and debugged under Microsoft Visual Studios™.

We will therefore be using Stratagus with the game module Wargus installed, throughout this project. Figure 10 on the following page shows a screenshot from each of the three open source frameworks discussed.

Figure 10: Ingame screenshots of the three most suitable frameworks. Top left: TA Spring Project, top right: ORTS, bottom: Stratagus with Wargus.

6 Analysis of the RTS Game

> *Intelligence is: (a) the most complex phenomenon in the Universe; or (b) a profoundly simple process. The answer, of course, is (c) both of the above. It's another one of those great dualities that make life interesting.*
>
> — *Ray Kurzweil*

When discussing the use of artificial intelligence in RTS games, it may not be entirely clear which aspects of the game the AI should be dealing with. In other words, what exactly should the AI do and why?

Even though different RTS games vary greatly, there are always some commanalities. In this section we will present a brief analysis on a possible representation of a CCP and discuss each individual part.

The fully functional AI can be split up into a multitude of smaller pieces, here referred to as level AIs, that together produce what we generally consider as a CCP. Each of the levels represent a task we humans, or the game engine itself, normally carries out while playing. Even though we might not think about every single one of them in detail, we most likely do give each one a tiny bit of attention, at least on a subconcious level.

The levels can be grouped to form a hierachy, as seen in Figure 13 on page 45. We will briefly discuss what these domains represent and how each of the levels work. The section ends with an explanation of the concept and importance of build orders.

6.1 The Game Engine Domain AI

This domain covers all the tasks related to the game engine, i.e. every aspect of the AI that a human player under normal circumstances cannot control or modify. Every unit in an RTS game has its own unique AI controlling it.

Whenever the player, either human or CCP, orders a unit to do some task, what really happens is that you tell the unit AI what you want it to do. The AI takes over the control, making sure the order is executed, or if something comes up, notifies you of it and aborts the order. The game engine AI domain consists of two different AI levels, the pathfinding level AI and the unit instinct level AI.

6.1.1 Pathfinding

When you order a unit to move from localtion A to location B, what do you really do? As a player, you usually select the unit you want to move, and right click on your mouse somewhere on the map, and the unit starts to move. This is where the pathfinding level AI comes in. By right clicking, or maybe by selecting a move-button on the GUI, you tell the unit that you want it to move, and where. Now it is fully up to the unit AI to bring it there.

An example of a problem the pathfinding level AI has to address is shown in Figure 11 on the following page. Here we have a simple unit located at (1). This unit is currently selected, and we order it to move to location (2), which would lead to a "move unit order" for the pathfinding level AI. There are a multitude

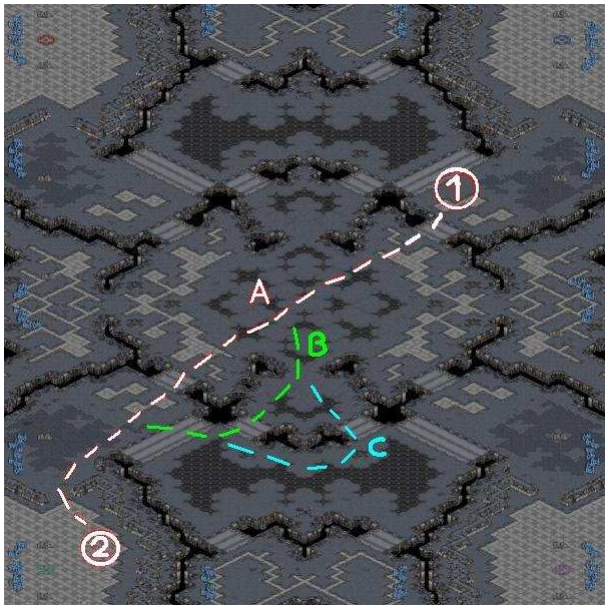

Figure 11: The Starcraft map Othello, illustrating the pathfinding level AI

of paths to choose from, where 3 of the most reasonable ways are shown, as A, B and C.

Which of the possible paths the unit will choose relies on the implementation of this particular level AI. For more information on pathfinding in RTS games see [EF08].

6.1.2 Unit Instinct

Unit instinct level AI, or unit behaviour level AI, is a level that controls how a given unit behaves or reacts when something or someone interacts with it. Consider the situation where a worker has recently been purchased and not yet assigned to any work, known as idling. An enemy unit is approaching quickly. This may seem like a typical scenario where the player needs to stay alert in order to save the worker from the opposing threat. Let us assume that the player is currently caught up in a big battle somewhere else on the map, and is not able to multitask fast enough to give this poor worker attention.

When the enemy initiates an attack on the defenseless worker, the unit instinct AI forces the unit to react. It is reasonable to think of the reaction as an instinct, since what the unit does often corresponds to its most reasonable move. A worker, even though capable of inflicting some damage, is very weak, so when attacked, it reacts by fleeing.

However, where a worker may try to run away from the opponent force, a swordsman would react by automatically engaging in battle. Units are not the only ones to have AIs, buildings may have one of their own. A defensive structure might have attack abilities, and whenever an enemy enters its attack range, the AI initiates attacking automatically.

The other aspect of the unit instinct level is maintaining and executing

given orders. If a worker has been assigned by the player to harvest gold, it may involve several actions, like moving from its current location to the gold mine, enter the gold mine and return the gold to the nearest headquarter. These actions are executed without the interference from a human player.

6.2 The Player Domain AI

This domain includes all decision making, strategy planning, high level movement of units and placement of structures and defences. Like the game engine domain, the player domain can be split up into multiple levels. Whereas the pathfinding and unit instinct level AIs were separated, many of the following levels will either overlap, or need to communicate with each other to some extent.

6.2.1 Terrain Analysis

Most games come with several terrain types, whether it is mountains, forests, lakes or even space stations. However, different terrain types may have different attributes, which can be used by a player to tip the game in his favor. An example is the higher ground concept in Starcraft [Ent09c].

Let us illustrate this concept by an example. In Figure 12 on the next page we have three sieged tanks (artillery) on the ledge of some higher ground, firing off at a pack of ranged combat units, positioned at a lower ground. Units at lower ground only have 70% chance of hitting units on higher ground. In addition, the higher ground remains unexplored, or covered by fog of war until they either move up or are attacked, leaving them in a less optimal situation.

Another part of the terrain analysis level AI is the concept of building influence and threat maps. These are essentially tables that describe how big of a threat a given unit is surrounded by. Such maps can be used together with path planning to avoid enemy units, defensive structures or even increase the chances of success when attacking. See [Ste06] for more on influence and threat maps.

6.2.2 Opponent Modelling

> *To know your Enemy, you must become your Enemy*
>
> — *Sun Tzu*

There might be some truth to the quote above. To be successfull in an RTS game, or maybe even a real life war, you must know what your enemy is up to. In RTS games, it is common to differentiate between defensive and aggressive playstyles, as well as different preferences when it comes to game strategies.

In a game like Starcraft, a player could go for an early defensive build, maximizing on building a steady economy, while fending off possible threats with defensive structures. Another possibility is to neglect the economy part, and spend the resources you have on a swift and aggressive "rush", where you aquire some basic warriors and attack as soon as possible (2.4).

With little information at hand, due to game restrictions such as fog of war and area covered darkness, a player should try to build a model, or at the very least guess what the opponent is doing. Since these games often have units that

Figure 12: Ingame screenshot of Starcraft, illustrating the need for Terrain Analysis as part of the Player Domain AI

counter each other (see Figure 2 on page 11), you do not want to end up buying units that are easilly countered by your opponent's army.

See [FSS07] for more on opponent modelling in RTS games.

6.2.3 Reconnaissance

Where the terrain analysis level AI is all about using map intelligence and various terrain aspects to better position your units, the reconnaissance level AI focuses on how to acquire such information, so called scouting. As a player in an RTS game, you are always faced with imperfect information, mostly due to the fog of war. Moving your units, that be workers, warriors, or designated scouting units around the map to gather information, is essential to winning the game.

This level AI needs to address questions like, which type of unit, how many units to send, where to send them and when or how often. Sacrificing a single unit by sending it straight into your opponent's base, may seem like a bad idea, after all, you end up one unit less powerful. However, you gain great insight into what your opponent is currently planning on and the gathered information is needed to maximize the effects of the opponent modelling and terrain analysis level AIs, as well as the possibility of discovering new resources scattered around the map.

6.2.4 Micro Management

Moving units in RTS games is a simple and intuitive task. You select the unit you want to move and either right click on the map or press some kind of move button on your keyboard. The pathfinding level AI will now move the unit from A to B.

When engaged in battle however, how you control your units may not be

quite as simple. First of all, you generally want to apply to the Rock-Paper-Scissors concept, as discussed in section 2.4 on page 11, by using units that act as good counters against your enemy units.

Let us say you already have a nice mixture of archers, swordsmen and cavalry in your army. Having your archers lined up in the front seems like quite a bad idea, as they are ranged and extremely vulnerable to close-ranged units. The clue is to position your units in a way that is coherent with the mixture of unit types both you and your opponent have.

Secondly, even though each unit may have a theoretical fixed position in the larger army, the concept of moving each unit individually to improve its position can often lead to more inflicted damage to your opponent, or even reducing the damage made on your own troops.

6.2.5 Strategical

For the CCP to become successfull, or maybe even to be able work at all, all the levels need to be glued together somehow. If they all operate on their own, one AI level could try to send a unit out as a scout, while another wants that unit to harvest resources. The strategical level AI tends to be somewhat of a coordinator, and makes the overall decisions on how the game is to be played.

Every game you play, you decide on some strategies to deploy. Even if you are at a completely basic level in this game, you still deploy strategies, they might just not be very good. Deciding on which strategies to use, when to use them and assign orders to the other level AIs are tasks of this particular level.

We can split the strategical level AI into multiple sub-levels, depending on the complexity of the game, and the number of units and structures that may be used simultaniously. We will briefly discuss a possible set of such sub-levels, high, medium and low.

High Level: Decides the overall strategy of the game. The selected strategy could be of many types, like early game rushes, defensive and economic build orders, low economy - massive army attacks, or maybe even a mixture of several strategies. Communication with the various level AIs can provide great information to choosing a more profitable strategy. These strategies are exactly those the opponent modelling level AI is trying to model. Strategy can be deployed by assigning medium level AIs to solve different parts of the tasks to be done.

Medium Level: Commands a group of units and structures and performs the given tasks handed over by the higher level AI. Examples of medium level AIs could be an organizer of an upcoming attack. Each unit is controlled by the micro management level AI and the unit instict level AI, but the AI deciding when and where to attack, is the medium level AI.

Low Level: Some games operate with very few units, like Warcraft 3, which has a maximum of 80 units at one single time, while others may have no such limit at all. If there is a huge amount of units, the medium level AI may assign work to even smaller parts of the group. This could come in very handy in a large army, consisting of archers, swordsmen, cavalry and pikemen. Instead of the medium level AI having to assign individual

orders for each unit, each type of unit could be assigned to a low level AI, for even more precise positioning.

6.2.6 Environmental

Some of the most important aspects of the game are those regarding the environment, that is, units, buildings and resource management. Where should we build the structures, which buildings and in what order should we build them, which units to purchase, where to purchase them and how to spend our money wisely? These are all questions that the environmental level AI addresses. The level can be thought of as consisting of the following three parts:

Resource Management: In some way, in every RTS game, you need to collect resources. Whether this is gold from a gold mine, gas from a refinery or tiberium from a deadly field of minerals. Nonetheless, units will be needed to gather resources, and how to acquire a steady income of the various types of resources will be a fundamental problem to solve. Apart from collecting resources, managing them is also essential. Whenever a unit is to be purchased, a technology upgrade researched or some building constructed, communication with this level is needed.

Unit Production: The responsibility of purchasing units lies within this level. When a strategical high level AI has decided upon some strategy, this level can decide which units to produce.

Construction: As for unit production, buildings are constructed according to some higher level strategy. Which buildings to build and where to build them are questions this level needs to answer.

6.3 Build Orders

A build order is a set of instructions describing the buildings, units and upgrades to purchase in the early stages of a game. How many steps a build order describes varies, both between games and between build orders themselves. The term "strategy" is commonly used to describe the overall game plan in an abstract way, like "rush" or "fast expansion". Where build orders describe the exact order and number of entities to produce, the strategy may describe additional features. Since the AI developed in this thesis only uses trained build orders for decision making, we regard build orders to be the same as strategies.

A build order does not provide information on where to build, which buildings to use for unit production, how to react to enemy behaviour, or how to collect resources. It only illustrates in what order the various entities should be purchased, and to some extent, when they should be made.

For a human player, learning a particular build order requires memorization and a lot of practice. Usually, the build orders are created either by professional players, or by observing and plotting down strategies that seem to work well over time. If two players are equally skilled, and player 1 builds whatever comes to his mind and player 2 follows a well founded build order, the second player will have a huge advantage. Having a particular build order in mind when starting a game is therefore key to success.

Listing 1: Simple Wargus Build Order

```
1 1/0 Great hall
2 1/1 Farm
3 1/5 Peon
4 4/5 Barracks
5 4/5 Farm
6 5/9 Grunt
7 7/9 Lumber mill
8 Etc.
```

Let us illustrate this by a simple, and most likely far from optimal build order in Wargus. The build order can be seen in Listing 1. There are two numbers preceeding each instruction, where the first one denotes the unit demand, and the second one the unit supply.

In RTS games, you need to maintain the "supply limit". This is a balance of how many units you are able to feed, or house, at a given moment in time. You usually get a higher supply by building farms, or equivalent buildings. Each time a unit is purchased, the demand raises. Most games operate with units that demand one supply, but games like Starcraft even have units that requires 2, 4 or even 6 supply.

Whenever your demand reaches the supply limit, you must purchase additional farms to be able to produce more units. Being able to build farms before you actually reach the supply limit enables you to keep producing units at a steady rate, intead of having to wait until a particular building in construction is finished. You can also have a negative supply limit if some of your farms get destroyed. You will still be able to keep the units you have, but you must get back to a positive supply to buy additional ones.

Looking at the Wargus build order again, we see that our first instruction is to build a great hall. This would enable us to gather enough resources for our second structure, a farm and an additional peon (worker). Next up is two more, here shortened and assumed to be peons, as the next instruction is performed at a demand of four. See section 7.3 on page 52 for more game specific details.

It is possible, but less common, to include conditions in a build order. See Listing 2 for an excerpt of a fairly advanced build order in Starcraft.

6.4 Nash Equilibrium

In game theory, Nash equilibrium is a concept where two or more players do not benefit of changing their strategies while the others keep their unchanged, with strategies of the opponent players taken into account. In other words, player 1 in a two player game is in Nash equilibrium if player 1 is making the best decision possible, taking into account the decision of player 2, and player 2 is making its best decision, taking the first player's decision into account.

Applied to our RTS problem, two Thanatos players would be in Nash equilibrium if neither one would diverge from its build order, taking the other one's into account. This is somewhat of a problem since Thanatos never takes opponent build orders into account when making decisions.

For the sake of the argument, let us say we have two Thanatos players

Listing 2: Advanced Startcraft TvZ Build Order

```
9/10 Supply Depot
11/18 Barrarcks
12/18 Send unit as a scout
if : Enemy expansion is scouted before demand hits 15
        15/18 Comand Center
        15/18 Supply Depot
        15/18 SCV
        16/18 Start Marine Production
else
        15/18 Supply Depot
        15/18 1 Marine
        16/18 SCV
        19/26 Command Center
        19/26 Start Marine Production
Etc.
```

playing against each other and that they are both able to take the opponent's build order into account when making decisions. If player 1 knows that player 2 is planning on a fast low economy rush, it knows that the opponent forces will be using some time to traverse the map. In the meantime, player 1 can get a couple of extra workers up before the defensive units are trained. Being able to hold back the attack, player 1 would have a huge advantage with additional workers and therefore additional resources and probably win the game.

Player 2 would however not select this strategy if it knew that player 1 would execute the mentioned counter strategy, as there would be even more beneficial strategies to choose from.

We thus see that strategies in RTS games behave similar to the rock-paper-scissors game described in section 2.4. If we know that our opponent uses a specific strategy, we can always select between a set of counter strategies, which would make our opponent change its strategy, and so on. Thanatos can therefore not be in a Nash equilibrium state.

6.5 Discussion

Section 6.1 and 6.2 shows how the CCP AI can be organized as a set of different levels. Not every level needs to be covered in order for an RTS AI to be successfull or interesting as an opponent. However, the more levels, the better the chance of being a challenge to the human player and give an illusion of intelligent behaviour.

So which of the levels are of most interest when constructing a CCP and which can be avoided without limiting it to such a degree that it cannot even play a standard game? Looking back at the game engine domain, we know it is higly game specific, as it addresses tasks like pathfinding and unit behaviour. This is generally not of a player's concern, so disregarding these levels will not limit the AI.

This leaves us with the player domain. Both opponent modelling and reconnaissance are quite important aspects of the game. Without opponent modelling, you are essentially forced to play a generic game strategy-wise, as you cannot adjust your own strategies to your opponents. Without reconnaissance, you are playing completely in the dark, with little or no map information at all.

Without covering these topics, a CCP will be quite limited as opposed to its full potential, but at the same time, it is only reduced to a bad player, much like a human player, new to RTS games, still unfamiliar with different game specific strategies and the concept of scouting.

The terrain analysis is very much like the reconnaissance in that having the level implemented may give the CCP a more fundamental and complete performance ingame. In the same way, not having it implemented makes it less optimal and definitely more prone to human exploitation. When that is said, the CCP can fully operate without knowing its environment, or to use hills and terrain to its advantage.

Micro management can be a very game specific task to handle. In one game an archer may be able to run fast between shots, and should be microed in between to improve life expectancy, while in some other game this would only slow it down. So implementing a micro management level AI requires an in-depth study of the game details, and would be both time consuming and as for the other topics discussed so far, not required to make a CCP work in a basic fashion.

The two levels we have not discussed so far are the strategical and environmental levels. For the strategical component, it is definately needed, as it is both the coordinater and the decision-maker that "make things happen". High level strategies could be as simple as to build a fixed amount of certain units or maybe even complete randomness. It all boils down to how advanced we want the CCP to appear.

The same goes for the environmental level. Without it, we cannot produce anything or have an understanding of resource management. Without units or structures, there is essentially no game, so that would be quite unfortunate. This level can be as complex as a cellular automata, or as simple as to produce one building of each type, in a given order.

Conclusion We have seen the different parts of the CCP and the tasks they need to address. As for creating a simple CCP we could manage with only the strategical and environmental levels, but as we add more of the other ones, both the complexity and the performance rises quickly. How complex we desire the CCP to be depends on time, resources, game specifics and consumer demands.

In this project, our attention will be directed to the build order concept, as it simplifies the strategical and environmental level domains and will yield a fairly simple CCP, which will be able to play games against an opponent. Focusing on the build order, we can easily study how the CCP adjusts to its opponent over a multitude of games. Since the build order only addresses a certain amount of steps into the game, this also helps simplify the vast amount of possibilities, and narrows down the complexity for a project like this.

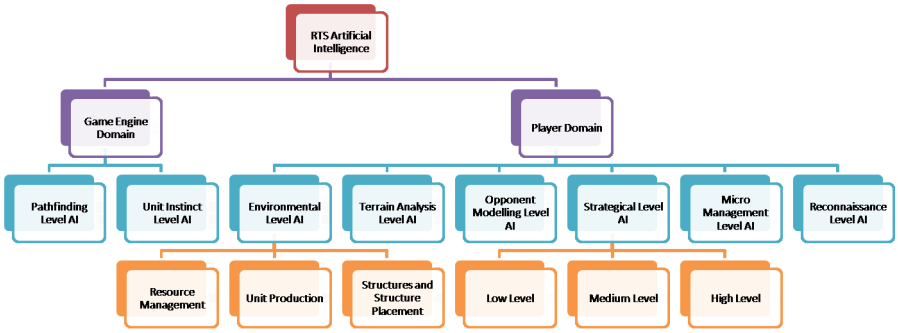

Figure 13: The various AI levels that constitute the fully functional computer controllable AI

Part V

Implementation and Results

7 Adapting The Framework

The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.

— *George Bernard Shaw*

In section 5.3 on page 31 we discussed possible frameworks and concluded to use the open source project Stratagus, with the game module Wargus as a plug-in.

In this section we describe the modifications that will be made and which parts of the framework that we use together with the Thanatos AI. A brief explanation of how a game unfolds and some needed game restrictions will also be presented.

7.1 Application Model

The entire application consists of six individual parts, see Figure 14 on the following page. We will develop a GUI control panel, which will be used to simplify the process of running a series of different simulation experiments (See section 9 on page 71), as well as provide both online and offline simulation specific details and statistics.

Thanatos and a set of Scripted AI opponents are the CCPs that we will use when running our simulations. They will be implemented as a part of the game engine, as they both use and expand on the current functionality provided by the framework.

With its learning capabilities, Thanatos needs a place to store the information learned at any point in time. The database will store this information, and provide access both to the learning AI experts and to the control panel.

Stratagus with Wargus, as presented in section 5.3.4 on page 33, is the backbone of our application. Framework modifications will be presented in section 7.2 and game details in section 7.3. The current version of the original game engine source code can be found at [Tea07a] and the game plug-in at [Tea07b].

Following is a short explanation of each of the four parts we will develop to combine and use with the game engine framework.

GUI control panel The Graphical User Interface control panel will be the platform from which we will be working once the implementation phase is completed. Whenever we want to run a simulation, whether it is to view a single game as the result of a set of trained Q-tables, or to start a new one, consisting of a thousand games, we will do it from this panel. We can adjust the number of games to be run, tailor the learning parameters and select the desired AI opponent.

The control panel will be a standalone process, written in Visual Basic.NET. When a simulation is started, the game engine process will be executed from the control panel and parameters will be passed on as command line arguments to define the simulation behaviour. While the simulation is in progress, the control panel and the game engine will be communicating through a TCP/IP connection. This way, we can adjust the game speed to our needs, i.e. a slow

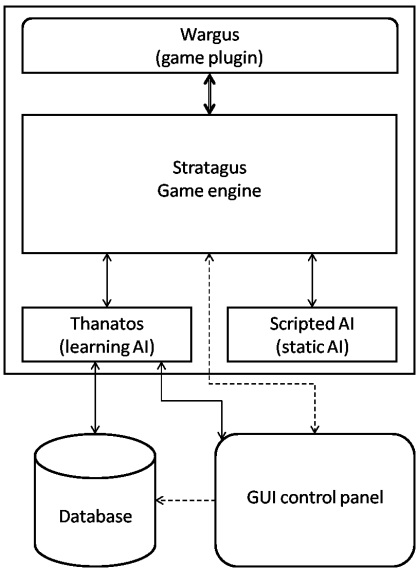

Figure 14: Illustration of the constituent parts of the application, including the Stratagus framework

speed if we want to observe the game as it is supposed to be played, or a high speed to make the simulation run as fast as possible.

The game engine will transfer state transition and action choices back to the control panel, so it can display the current expert states and the actual decisions made by Thanatos to the user.

The control panel will also help translate the trained Q-tables, state transitions and game data into meaningful results. See appendix B on page 96 for a detailed description of the GUI control panel functionality.

Database The database is merely a collection of a few simple files containing many numbers. As described in section 8.1.2 on page 58, each expert (see section 8.1) will have its own Q-table file in the database, used for learning purposes. Thanatos AI experts will have write access to the Q-tables, whereas the control panel only needs read access.

In addition, we will keep track of all game results, i.e. wins and losses, and store every state transition made by an expert in the database as well, as two additional, but separate files. The control panel can then read these files and display how a single expert's learning progresses, as well as produce statistics and understandable results. These files will be written to by the control panel, not by the game engine.

Thanatos AI Thanatos is the the center of attention in this thesis. Its primary objective is to make in-game build order decisions, and be able to improve these over a series of games. With different rewarding strategies (8.4) and a multitude of parameters to tweak and modify, we will study how the AI performs against various opponents and under different conditions.

Thanatos works as the decision making module, hooked into the rest of

the Stratagus AI game engine (7.2.1), using framework parts to solve simple general purpose tasks, like collecting resources, finding free locations to build on, attacking the enemy and general unit pathfinding.

The implenetation details is described in section 8 on page 57 and some of the most important framework modifications will be discussed in section 7.2.

Scripted AI To create a regular one-on-one RTS environment, we will need to have an opponent for the Thanatos AI. In addition to playing against an instance of itself, we will be using a static player. This is a player who makes the same moves every game.

As for Thanatos, the scripted AI will be implemented using the important parts of the game engine framework. However, its decisions will be predefined and unchangable. A script will be described as a build order (6.3), and different scripts can therefore be used to train Thanatos in different environments. Framework AI details will be described in section 7.2.1 and implementation specifics in section 8.3.3 on page 65.

7.2 The Modified Game Engine

The Stratagus game engine supports a fairly simple AI with scripted behaviour, much like some of the opponents we will be using against Thanatos, described in section 8.3.3. In this section we will briefly describe how the environmental and the strategical level work and follow up with some of the details of what has been modified to our needs.

7.2.1 The Stratagus AI

The static AI does not learn, analyze or take anything observed during a game into consideration. It merely executes a set of predefined actions. The game engine can thus be said to only support the two AI levels; environmental and strategical, in addition to the game engine related levels; pathfinding and unit instincts.

In Figure 15 we can see how both Thanatos and the scripted AIs are extended to the player domain, as well as being closely tied to the game engine domain.

7.2.2 Environmental level AI

There are two main queues in the game engine, used to keep track of player given orders; the production queue and the research queue. The production queue will store structure and unit orders, whereas the research queue handles all the upgrade requests. Every game cycle these queues are inspected to see if either more resources are needed or if an actual item can be purchased or constructed. Each player has its own object instance of these important queues.

If a fairly expensive item has been placed in one of the queues and additional resources are needed, the environmental level AI constructs a resource request. Like the production and research queues, resource requests are also inspected every game cycle. Whenever resource requests are issued, free or even busy workers will be assigned to harvest the amount needed of the specified resource type. When a worker has been assigned to harvest resources, the pathfinding and the unit instinct level AIs will control its behaviour (6.1).

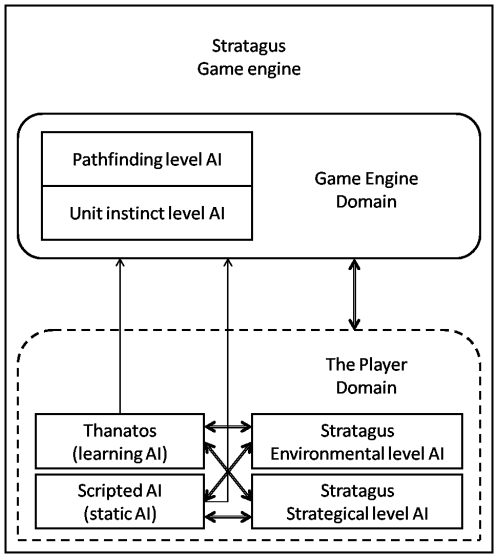

Figure 15: Internal view of how the AIs are connected to the Stratagus game engine

When enough resources for a structure has been gathered, a free building location needs to be found. Some buildings, like the great hall (see section 7.3 on page 52), should generally be built as close to a gold mine as possible. If a gold mine cannot be found, it should be close to wood, and always at a great distance from your opponents. The game engine comes with a set of functions that handle this appropriately. See section 6.2.6 for more on the environmental level AI.

7.2.3 Strategical level AI

The overall strategy of scripted AIs comes in the form of build orders (6.3). To create a "rushing AI", you would need to create a set of structure and unit requests for this particular behaviour, very much like we do in section 8.3.3.

In addition to the regular build order commands, the scripted Stratagus AI will also describe which force a particular warfare unit belongs to and when to issue an attack. As for the environmental level AI, the strategical level will check if a new attack command should be issued, every game cycle.

See section 6.2.5 for more on the strategical level AI.

7.2.4 The Modifications

Both Thanatos and the scripted AI use quite a lot of the features provided by the Stratagus framework. We will modify

However, not all of them are compatible without modification, either because the feature is superfluous and complicates the entire game, or because it violates with some of our implementation strategies. Following is the most important changes that we have made to the game engine.

Adjusting the queues There are two problems regarding the production and the research queues. The first is that they are constructed in such a way that cheaper requests may get precedence if there is a lack of resources for the more expensive ones, even though the expensive ones were added to the queue first. In other words, the queue does not function as a FIFO list. We would like it to be a fair FIFO list, since we want Thanatos to learn exact build orders, not just a set of buildings required throughout the game.

The second problem is that farms, that are needed to increase supply on high demand, are produced on a "supply-is-needed" basis. In other words, the Environmental level AI will each game cycle check if the demand is higher than or equal to the supply, and issue a new farm order. This also poses a problem for Thanatos, as we want it to learn when to construct farms, not just build it as a static consequence of having a supply barely greater than the demand, as this does not have to be the optimal way to produce farms.

The queues have therefore been slightly modified to address these issues.

Repair and Rebuild Workers have the ability to repair damaged buildings for the cost of a small amount of resources. The AI checks, as for everything else, each game cycle for needed repairs, by calling a function called AiCheckRepair(). As repairing buildings results in longer and more advanced games, we avoid calling this function. In fact, repairing a building may turn out to be poor usage of the available workers, at least when there are few of them.

The Stratagus AI is implemented in such a way, that whenever a unit or structure is lost, the AI tries to reproduce it by pushing a new queue order. This violates with the concept of building only the items described in the build order, as a build order generally does not describe the items you should have, but the items you should produce. This also poses a problem for Thanatos. As state transitions occurs when an item has been completed, this would make Thanatos transition every time an item was rebuilt, and this without ever making the choice to actually build them. This could easily lead to a crash due to the limited state space we operate with.

Both the repair and the rebuild concepts have been disabled for both Thanatos and our static scripts, as we want them all to operate on the same terms.

Warfare As described briefly in section 7.2, warfare units are put into groups called forces. If we were to create a force, we would decide how many units of each kind we need and when to attack. This is very straightforward for a static build order, but for Thanatos however, this has no meaning at all.

Since Thanatos does not have any predefined understanding of the units that will be trained during a game, knowing when a force has been completed, or even define a force at all, leads to some fundamental problems.

We have therefore redefined the force concept slightly, so Thanatos can be able to engage in warfare on the same terms as its opponents. The regular approach is to define a force of units, wait for the units to be produced, and then attack when an attack order is issued. Instead of this, we let every produced warfare unit, i.e. grunts, axethrowers and catapults (see 7.3) be assigned to a unique force.

Instead of waiting for an attack order, which Thanatos knows none of, the attack will be issued immediately after the training phase has been completed.

In other words, a warfare unit will initiate an attack right away after it has been trained.

The modified force concept will be used by Thanatos, as well as all of our scripted players.

General A few other more general modifications have been made. We will here briefly point out some of the most important ones.

Length: A standard game ends when either player has been defeated. We added the possibility to end a game when a given number of game cycles have passed. This can be used to train Thanatos in various ways, and ends a game if neither player is able to finish it.

Speed: Instead of being able to only play one game at a time, at normal game speed, we have added the possibility to run multiple games in a row, without restarting the game engine, in addition to bring the game speed up to a maximum.

Player: In a normal setting, you see your own units ingame, and control these as you wish. Since there is no need for human intervention after a game has started, the "player view" has been modified so that the user can observe both AI players, in addition to being unable to tamper with the game.

Technical: Command line arguments have been modified and communication support with the GUI control panel through winsock connections have been added.

7.3 Game Specifications

If you want to play a standard Wargus game, you have to choose between two different races when you start the game. Either you choose to be the "not so unknown" race of humans, or you choose the orcs, much alike the creatures popularized through J.R.R Tolkien's "Lord of the rings".

Technically, there are a few differences between the two races at an advanced level, but for this thesis, only the most basic units, structures and upgrades will be used, and the races can therefore be viewed as identical. We restrict the race selection to orcs only, for all of our AI players.

The internal Stratagus AI disregards the fog of war concept and thus has full knowledge and map information. This is practically the same as cheating, but since it allows a much simpler AI and the fact that we are not really concerned how our AI performs against a human player, Thanatos will also be implemented with this advantage. Since all players now have access to all of the same game information, the players are in fact playing on the same terms, i.e. a fair game.

Resources The game originally has three distinct types of resources you need to collect; gold, wood and oil. Gold can be harvested in gold mines, by sending a worker to a mine, and then return back to the nearest great hall with the collected gold. The same goes for wood, except that it is chopped from trees scattered around the map, and that it is slower to harvest than gold, and can be delivered to lumber mills as well as great halls. Oil however, can only be

collected at sea, and is used for more advanced purposes. We therefore exclude the use and collection of oil and warfare on water in our simulations.

There are some harvesting bonuses that can be achieved throughout a game if certain buildings are either built or upgraded. If a lumber mill is constructed, the worker will return 125 pieces of wood instead of the regular 100, every time the worker returns with the goods. This does not increase the tree removal rate, it just gives a small bonus when delivering the wood. You can view it as an improvement of processing the wood at the main base.

The two other bonuses comes in form of either receiving 110 or 120 gold pieces for every 100 you collect in the gold mine. These can be achieved by upgrading the great hall into either a stronghold or a fortress. However, upgrading the great hall is an advanced feature, requiring oil, and thus makes these bonuses unavailable for our AIs.

Map Design The game comes with a variety of different maps, each with a set of unique properties. Maps are usually selected according to personal taste and balance. However, some maps can be extremely biased or unfair to some of the players, either because there are less space at the initial starting location to be built on, or there are longer distances to be travelled to find resources.

For our simulations, we want both Thanatos and its opponent, whether this is a static script or Thanatos itself, to have the same opportunities, i.e. a fair game. For this we have created a small, simple and symmetrical map, illustrated in Figure 16. The black and the white circles represent the player starting locations, i.e. where the worker is placed when the game is started. The yellow figures are gold mines and the green rectangles are trees, ready to be harvested.

This is a rich environment in resources, with more wood than could possibly be collected in a single game, additional gold mines so expansions can be utilized, and lots of space available for significant battles. The map is also fairly small, only 64×64 tiles, so games will be shorter than on a larger map.

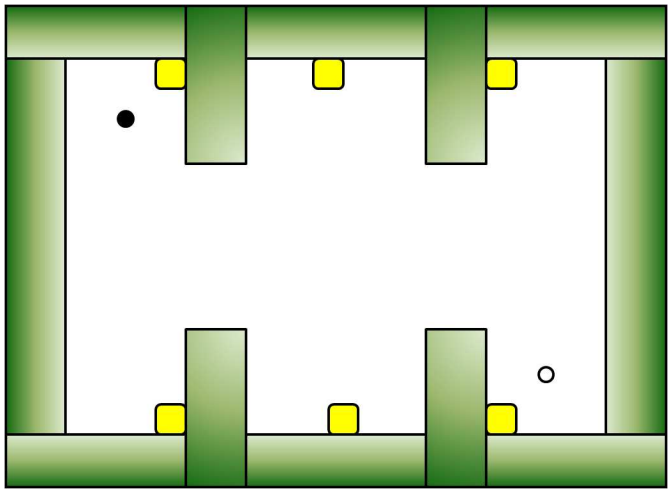

Figure 16: The game map used in all of our simulations

Structures The game supports a multitude of different structures, as can be seen in [Ent09b], each with a set of unique properties. Out of this myriad of buildings, we have decided to only support the five most basic ones. This is partially to keep the AI fairly simple, and partially to prevent enormous state spaces (8.1.2).

The supported structures are depicted in Figure 17. Each structure is unique in terms of resource cost and the amount of damage it can take from an opponent. Following is a short description of the five structures.

Pig Farm: The pig farm increases the available supply and is thus needed to be able to increase the amount of units when the demand is greater than or equal to the current supply. For each pig farm an additional 4 units can be produced before yet another one is needed.

Great Hall: This is the headquarter building and is critical for a player to be successfull, as this is where the gold is delivered by workers. The great hall can produce additional workers and raises the supply by one. Whenever one of our AIs have at least one great hall and construct another, this is regarded as expanding. This means that the new great hall will be placed at a strategical location, usually close to an unused gold mine, at a great distance from the opponent player.

Barracks: This is the training camp for warfare units. Either one of the three supported units; "grunts", "axethrowers" or "catapults" can be trained in this building. Axethrowers and catapults require additional structures to be purchasable from the barracks.

Lumber mill: Unlocks the possibility to produce axethrowers from the barracks. The axethrowers can get their attack damage upgraded in this structure as well.

Blacksmith: Together with the lumber mill, the blacksmith unlocks the possibility to produce catapults from the barracks. The blacksmith also serves as an upgrade station. Grunt armor and attack damage can be upgraded, as well as the attack damage for catapults.

Units We will support the four most basic units, illustrated in Figure 18. As for the structures, these are unique in terms of resource cost and hit points. In addition, the units have different attack ranges and inflict different amounts of damage to opponent forces. They also apply well to the "rock, paper, scissors" concept discussed in section 2.4 on page 11. Following is a short summary of how the different unit types behave.

Peon: The orc workhorse which continuously collects resources or constructs buildings. They do have a basic weapon, but it will not be used for our simulations, they will primarily be used to collect and construct.

Grunt: A well-armored close combat fighter with both armor and attack upgrade possibilities.

Axethrower: A light-armored unit with the ability to strike from a distance.

Name	Figure	Gold Cost	Wood Cost	Hit Points	Supply
Pig Farm		500	250	400	4
Great Hall		1200	800	1200	1
Barracks		700	400	800	0
Lumber Mill		600	450	600	0
Blacksmith		800	450	775	0

Figure 17: Supported structures and their properties

Catapult: A slow long ranged unit able to inflict a high amount of damage.

Each unit has two types of damage that is inflicted on a single blow; basic damage and piercing damage, where piercing damage describe the unit's ability to pierce through or bypass enemy armor.

The damage is inflicted according to the following formula:

$$M = \begin{cases} (B - T) + P & \text{if T} < \text{B} \\ P & \text{if T} \geq \text{B} \end{cases} \tag{2a}$$

$$A = \beta M \tag{2b}$$

where $\beta \in [0.5, 1]$ is a random number, B is the basic damage, P is the piercing damage, M is the maximum damage, T is the target armor and A is the actual damage inflicted.

Looking at the table in Figure 18, we use the second row to calculate the actual damage a grunt would inflict on an opponent grunt. For this particular scenario we have $T = 2$, $B = 6$ and $P = 3$. From equation (2a) we get

$$M = (B - T) + P = (6 - 2) + 3 = 7$$

Equation (2b) gives us the actual damage $A \in [\frac{7}{2}, 7]$. Since all structures have an armor of 20, this formula shows why catapults may be an effective choice in destroying buildings.

Upgrades We have chosen to support the four most basic upgrades in the game, making it possible to upgrade all three of the warfare units, enhancing their abilities. They can be seen in Figure 19.

Each upgrade has two levels that can be researched. Each level adds a bonus according to the corresponding table entry in the "Bonus Per Upgrade" column

Name	Figure	Gold Cost	Wood Cost	Hit Points	Basic Damage	Piercing Damage	Armor	Range
Peon		400	0	30	3	2	0	1
Grunt		600	0	60	6	3	2	1
Axethrower		500	50	40	3	6	0	4
Catapult		900	300	110	80	0	0	8

Figure 18: Supported units and their properties

in Figure 19. Upgraded attack damage is always added to the piercing damage of the affected unit, making the unit able to do more damage regardless of the enemy armor.

Name	Level 1	Gold Cost	Wood Cost	Level 2	Gold Cost	Wood Cost	Bonus Per Upgrade	Affects	Researched in
Battle Axe		500	100		1500	300	+2 damage	Grunts	Blacksmith
Armor		300	300		900	500	+2 armor	Grunts	Blacksmith
Catapult		1500	0		4000	0	+15 damage	Catapults	Blacksmith
Throwing Axe		300	300		900	500	+1 damage	Axethrowers	Lumber Mill

Figure 19: Supported upgrades and their properties

8 AI Implementation

You cannot teach a man anything. You can only help him discover it within himself.

— *Galileo Galilei*

In this section we will describe the details of how Thanatos will be implemented. First we will discuss how it will make its decisions, how states and actions are represented and how much storage space is required for the Q-tables.

Next we will discuss and present a reward delaying algorithm, designed to compensate for the time it takes to complete an action after it has been selected. This is followed by a discussion on build orders, where we will describe the purpose of each scripted opponent, as well as explain how it plays. We end the section by explaining the exploration and rewarding strategies we will be using in our experiments (see section 9). Excerpts from some of the implementations can be seen in appendix D.

8.1 The Experts

We will be using the term "expert" to describe an abstract part of the code that represents a human expert who makes decisions within its particular field of expertise.

There are three sets of supported actions, each describing a unique functionality that Thanatos supports

- structure construction
- unit training
- upgrade research

as described in section 7.3. Thanatos needs a mechanism to choose between these actions. For this we will use a set of experts, being able to learn using Q-learning, as described in section 3.2. Each expert thus operates and learns entirely within its own domain.

Each expert will be given a unique Q-table, enabling it to learn which actions are better than the other ones in any given situation. Section 8.1.2 describe the details of each expert's action space, A_e.

As each expert operates only within its own narrow domain, we need something to combine and control all of the decisions made. For this we will be using a slightly modified version of a blackboard architecture (see section 4.1.4).

8.1.1 Blackboard Architecture

A blackboard architecture is usually a shared part of memory where various experts can gain knowledge about some problem, write down new information, and the most important action among all experts is selected.

In our problem we have multiple experts, each solving a small problem, each with an opinion on what is supposedly the current best action to perform. We

thus use a blackboard architecture to combine the decisions of the different experts.

Instead of letting a single expert look at the blackboard and indicate its interest, we let the blackboard be represented by a single Q-table indicating which expert to select. This way, only one expert choice will be selected when an action is to be performed. Valid arbiter actions and states are described in section 8.1.2.

When a game starts and a Thanatos player is allowed to run, the arbiter routine for that particular player will be started by the game engine. Performable actions are selected by the structure, unit and upgrade experts, so the arbiter is responsible for selecting one of these. To do so, it does a lookup in its own Q-table and makes a decision. The selected action is either the best learned so far, or a random one, depending on the exploration strategy used, which we will introduce and discuss further in section 8.4.

Each action has a corresponding number, so we can select a random one by drawing a random number between 1 and the number of valid actions the expert currently has. If the best action is to be selected, the entire row in the Q-table is traversed, locating the cell with the highest Q-value, indicating the best action learned so far.

Now that an expert has been selected, the decision routine for this particular expert is called. As for the arbiter, a Q-table lookup is made and an action selected, using the same exploration strategy.

The selected expert hands the action over to the arbiter, which in turn pushes the action into either the production or the research queue in the game engine (see section 7.2). The first action has now been selected and the environmental level AI will make sure that sufficient resources are gathered and the action performed.

When the action has been completed, i.e. a structure constructed, a unit trained or an upgrade researched, the game engine signals the corresponding player's arbiter that a new action should be selected. This process continues until the game ends. An illustration of the decision making process can be seen in Figure 20.

8.1.2 Action and State Representation

Before we can implement the experts, we must decide on the action and state spaces of each expert.

We know that the experts have their individual set of supported entities. Selecting a particular entity for production or research will be regarded as our actions. The exception is for the arbiter, where an action will be selecting a particular expert. We thus write the general action space as

$$A_e = \Big\{a_1, a_2, ..., a_n\Big\} \tag{3}$$

This set denotes all the possible actions a given expert can choose amongst.

Next we need to address how a state should be represented. The variables we have available are the set of possible actions. Whenever we select an action we would like a state transition to occur, as this signals an update of the Q-table

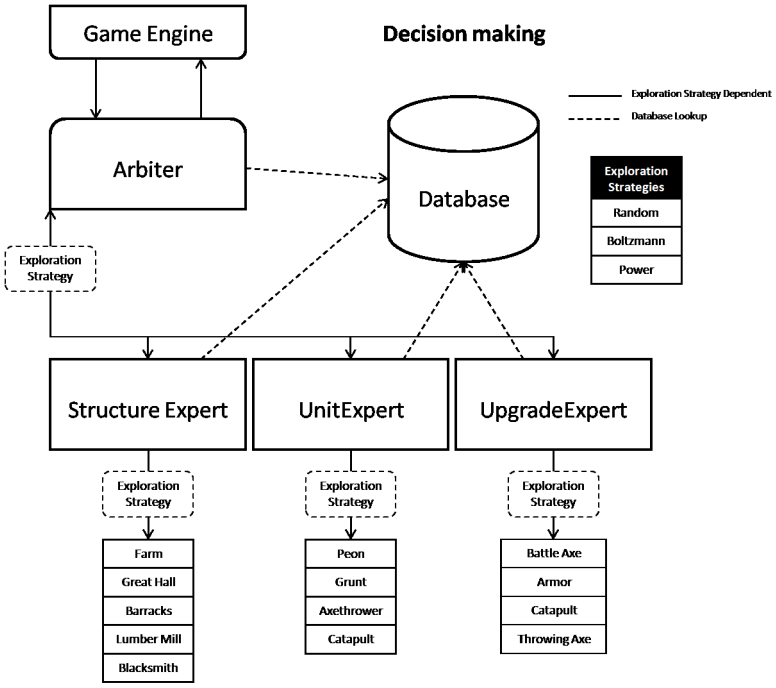

Figure 20: Illustration of how the various experts are connected together

and affects the ongoing learning. The difference between two states is therefore the different number of actions that have been selected so far.

Put another way, the number of times each action has been selected denotes the current state. A state is therefore represented by a simple vector of length $|A_e|$, where each entry describes the number of times the corresponding action has been performed. We get the state space

$$S_e = \left\{ (a_1, a_2, ..., a_n) \, | a_i \in \{0, 1, ..., k_e\} \right\} \tag{4}$$

where k_e is a constant, defined uniquely for each expert, denoting the maximum number of allowed selections of any action. We discuss what values of k_e we will be using, in section 8.1.3 on the next page.

Say that the unit expert learns that a worker is the best action to choose, when in the initial state $(0, 0, 0, 0)$, and that the next one is a grunt. These actions are not necessarily performed in sequence, as the arbiter may have learned that after the first choice of the unit expert the next action should be a structure expert. We are therefore able to learn both in what order to select experts, and in what order structures, units and upgrades should be purchased, resulting in a correct build order.

To clarify this we present an example as seen in Figure 21. Here we represent the order of best actions learned in each Q-table as a list. We see that the first three units to train is a grunt, a peon and a catapult. However, the arbiter has learned that a farm should be constructed in between the grunt and the peon,

resulting in the build order; grunt, farm, peon, catapult. An upgrade at the end of the build order has been added to the example illustration.

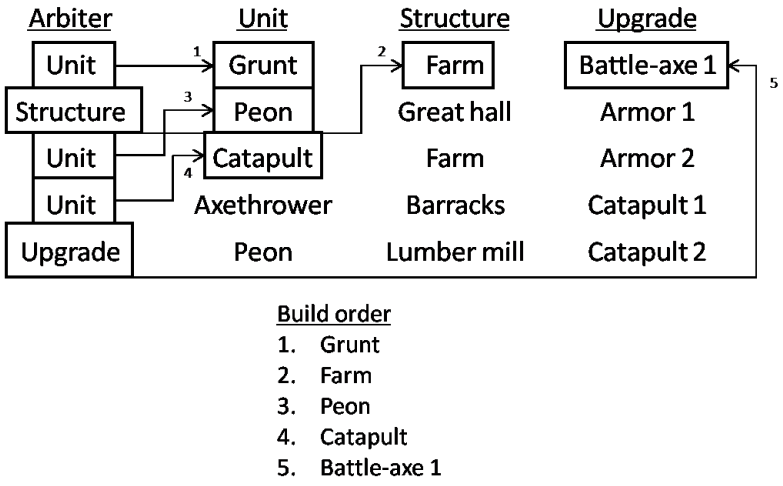

Figure 21: Example of how a build order is constructed from the learned actions of the set of Q-tables. The tables are here illustrated as lists, containing the order of best learned actions.

8.1.3 Deciding Constraints

Equation (4) shows that the state space of a particular expert is bound by the value of k_e and $|A_e|$. These constants describe how many choices of a particular action we support, as we operate with finite stace spaces in this thesis.

In order to decide on these limits, we should investigate the required memory and storage space needed using Q-learning, as explained in section 3.2, with the state and action model we described in section 8.1.2. We have the following equations

$$S_e = (k_e + 1)^{|A_e|} \tag{5a}$$

$$Q = S_e|A_e| \tag{5b}$$

Equation (5a) tells us that the size of the state space, is given by the maximum number of supported actions and the number of times any action can be chosen. Since having selected no action is possible, like in the initial state, we add one to the base of the equation to include these states.

Equation (5b) describes how many values the actual Q-table will contain. In fact, it will be implemented as a double precision matrix with S_e rows and $|A_e|$ columns. Assuming that each Q-value is stored as a number in the range $[-3, 3]$, like we do in section 9.2.3 on page 77, we can estimate the upper limit of the needed database disk space.

With a maximum number of 6 decimals, including the dot, the base value, a possible sign and a whitespace to separate the q-values, we get 10 characters, or 10 bytes per q-value. This gives us

$$D_e = 10Q$$

where D_e is the diskspace required in bytes for this particular expert. However, each row is ended with a newline, resulting in an additional $2S_e$ bytes. We have also calculated one whitespace too much per Q-value, as we only have whitespaces between the actions, not after the last one in a given row. We thus get the maximum number of bytes to be

$$D_e = 10Q + 2S_e - S_e = 10Q + S_e = S_e\big(10|A_e| + 1\big)$$

With this in mind, we can now discuss and choose the values we use for k_e. Let us first take a look at the structure expert. With five structures supported, the needed space will grow extremely fast for increasing values of k_e. We should therefore select a fairly low value. For the structure expert, it is very uncommon to construct more than 4 or 5 of most structure types.

With $k_e = 4$ we get $D_e = 159375$ bytes, or close to 156 kilobytes, which is not a problem.

For the unit expert we must allow a fair number of units to be built. If we limit each unit type to 7, i.e. setting $k_e = 7$, we get $D_e = 167936$ bytes, or 164 kilobytes. It is unlikely that more than 28 units, that is 7 of each of the four unit types, will be purchased during our limited games, indicating that 7 should be a solid choice for k_e. Again, the space required is not a problem.

The upgrade expert does actually have a built-in limit of 2 levels per upgrade type. This means that we get $k_e = 2$ and $D_e = 3321$ bytes, or about 3 kilobytes.

As with the other experts, the arbiter must also have a finite state space. By limiting the maximum number of arbiter actions, we are really limiting the amount of choices each of the other experts can perform during a game. This value should therefore be somewhat higher than the others. With $k_e = 16$, each expert will be able to perform an action up to 16 times each game. 16 actions for each expert allows all upgrades to be researched and leaves enough room to constantly produce units and structures throughout the game. This gives us $D_e = 126976$ bytes, or 124 kilobytes of required storage space.

With the selected values, we see that we actually need less than half a megabyte of diskspace to store our Q-tables. Since the storage representation is somewhat simpler during runtime, the required memory is actually less than the needed hard drive storage space.

It is worth pointing out that there are a lot of states that never will be reached, due to game restrictions, such as the impossibility of training a unit before a corresponding structure has been built, as addressed in section 8.1.4. The actual used disk space will therefore be somewhat lower than our estimates.

8.1.4 Invalid actions

When an expert faces a decision, some of them may not actually be valid. By this we mean that the particular action cannot be purchased or researched, due to game restrictions. An example of such an action is to research an upgrade before any structures have been constructed. As an upgrade is always researched in either a lumber mill or a blacksmith, the arbiter should not be able to select the upgrade expert before at least one of these structures have been built.

A set of functions will be called during the arbiter's decision making, ensuring that at least one action is valid for the expert selected. The functions will be called criteriasMetStructures(), criteriasMetUnitExpert() and criteriasMetUpgradeExpert(), and returns true if the corresponding expert has at least one legal action to perform. In other words, the arbiter is only allowed to select the unit expert if a unit can actually be trained.

For the structure expert, resources could have been considered a restriction, but since resource gathering is controlled by the environmental level AI (see section 6.2.6 and 7.2), and performed on a "gather-when-needed" basis, we will not put any restrictions on structure selection. This can result in the AI being "stuck" if it tries to construct a building with insufficient funds and no income. This is not a problem since a loss is then inevitable, resulting in a punishment which is learned as bad behaviour.

For unit production, the current supply must be higher than the demand in order to train additional units. Corresponding structures for both units and upgrades must be constructed in order for their experts to be selected.

8.1.5 Winning the game

A game is won when all your opponent units and structures have been destroyed. In the same way, you lose the game if all of yours are destroyed. In addition, a game could be stopped if it reaches a given game cycle. This could happen by playing against some of the scripts (see section 8.3.3), since they never attack their opponent. It could also happen with two instances of Thanatos, where neither one gets to train warfare units to win the game with. Therefore Thanatos will lose the game if the time runs out, making it possible for both players to lose at the same time if there are two Thanatos instances playing against each other. However, a script will always receive a win if the time runs out. The game cycle cap can be specified in the GUI control panel (see appendix B on page 96).

8.2 Delaying the reward

Decisions are made throughout a game, but when does a state transition occur and when do we actually update the Q-tables?

A general approach to a Q-learning problem would be to select an action and immediately transition to the next state, thus implying that the action has been successfully performed. In an RTS game however, this is not as trivial. Whenever an action has been selected, the chosen entity will be put into a queue, becaue resources need to be gathered; and at some point later in the game the action is completed.

Before describing the algorithm we have designed, we will discuss some of the problems it handles.

Let us say we start a game and choose to construct a blacksmith. In doing so, we eliminate the possibility of affording a great hall, thus making an income impossible throughout the whole game. The blacksmith is completed rather fast, and a new building is selected, let us say a great hall. Since we cannot afford it, the AI will be "stuck", and the opponent will either defeat us, or the time will run out. Either way, we will lose the game.

Now the question is, should we give a negative reward for the action selection of the blacksmith, or the effort of trying to build the great hall? We could reward the great hall action with a negative value, indicating that this was a bad choice in the state of one blacksmith. A state transition to the blacksmith state in a later game would thus pick up on some of this negative reward and indicate that the blacksmith is a bad choice in the initial state.

Instead we could reward the last successfully completed action directly. This is more reasonable as it was an action that clearly affected the game and the reward is immediate.

To be able to punish building the blacksmith, we will introduce two concepts; saving and delaying. Whenever we save or delay something, we are actually storing away both a state and an action, to be used for later purposes. In our example, in order to punish correctly, we need to update the Q-table row corresponding to the state we were in, and the column corresponding to the selected action. In other words, we need to update $Q[0][4]$, as a blacksmith is the structure expert's action number 4.

State 0 and action 4 is thus "saved" right after the decision has been made. The blacksmith is then put into the production queue (see section 7.2.1)and built as soon as possible. Now we check if there is a delayed action pending, meaning that we should do a Q-value update for the particular stored action. As we have only saved state 0 and action 4 at this point in time, we have no delayed values yet.

We then set the delayed values to be the saved ones, transition to the next state, and make a new decision, i.e. the great hall. New values gets saved, but the delayed ones remain from the first decision. Since the game has now ended, due to time running out, we check if there is a set of delayed values to be distributed. We recall that the delayed variables were set to the saved ones from the first decision, making state 0 and action 4 accessible. We can now update the Q-table according to equation (1).

Whenever we update the Q-table, we do so with a given reward. As we cannot tell how well the AI is playing until the game is over and the outcome is available, we must delay the reward or punishment. We do this by doing a regular Q-value update, whenever a delayed reward is defined at the completion of an entity, with the reward set to 0. This way we propagate the other learned values accordingly, but do not interfere and say that a new action has proved to be either good or bad.

When the game has ended, we check the results and hand out correct rewards to the last completed action. A pseudo code can be seen in Algorithm 2.

8.3 Build Order Representation

We will be using build orders (6.3) to describe the decisions made by a trained Thanatos AI, as well as the full behaviour of the scripted opponents. In this section we will present the use of these build orders and the opponent script designs.

We can think of build orders as used in two different contexts; as a result of Q-learning decisions and as an instructional script during game. The first one was discussed in section 8.1.2 where we illustrated an example in Figure 21.

Algorithm 2 Pseudo code of the delayed reinforcement learning algorithm used in Thanatos

Require: All expert tables and states initialized to 0 and savedAction/delayedAction set to not defined.

```
1: for all i = 0 to GameIterations do
2:     repeat
3:         selectedAction = getAction(explorationStrategy)
4:         productionQueue.add(selectedAction)
5:         save(selectedAction)
6:         waitForCompletion(selectedAction)
7:         if isDefined(delayedAction) then
8:             distributeDelayedReward(delayedAction)
9:             delete delayedAction
10:        end if
11:        delayedAction = savedAction
12:        transitionState(getNextState(selectedAction))
13:    until Game ends
14:    if isDefined(delayedAction) then
15:        distributeDelayedReward(delayedAction)
16:        delete delayedAction
17:    end if
18:    reset(allExperts)
19:    setNotDefined(savedAction)
20:    setNotDefined(delayedAction)
21: end for
```

Here we viewed the order of best learned actions by all the experts as the trained build order. The other use of a build order is to describe a set of instructions to follow when playing a particular game.

8.3.1 The Syntax

Since the arbiter selection of an expert always leads to a request of either a structure, unit or an upgrade, a build order should consist of a set of instructions of these types, leaving the arbiter expert selection out of the picture. Following is a short description of the three supported instructions.

Build: Adds the specified building to the production queue (7.2.1), which is then built on some location found by internal game engine functions.

Train: Adds the specified unit to the production queue, which is then trained at a free corresponding structure.

Upgrade: Adds the specified upgrade to the request queue, which is then researched at a free corresponding structure.

A simple excerpt of a possible trained Thanatos build order output is illustrated in Listing 3.

Listing 3: Example of a simple build order

```
1 Build   unit-great-hall
2 Build   unit-pig-farm
3 Train   unit-peon
4 Build   unit-orc-barracks
5 Train   unit-grunt
6 Build   unit-orc-blacksmith
7 Upgrade upgrade-catapult1
```

8.3.2 Thanatos

In learning mode, Thanatos selects its decisions based on the current Q-values with respect to the selected exploration strategy (8.4). Whenever we turn the learning mode off, or run the "Calculate Build Order" functionality in the control panel application, Thanatos can be regarded as playing by a simple script, where each instruction is given by the best Q-values currently available.

When calculating a particular build order, a regular game will be played, but instead of using some exploration strategy, the best Q-values will always decide the actions to perform. The selected actions will be outputted to a file, with the syntax described in section 8.3.1.

8.3.3 Scripted Opponents

We have designed three scripted opponents for Thanatos, each inspired by a unique core strategy. The scripts are fully described by a corresponding build order and can be found on the attached DVD in the *src/master/scripts/ai* directory, or seen with the syntax described, in section 8.3.1 in appendix C.

The actual lua-file scripts differ somewhat from the build order syntax we present in this thesis. Each script defines an array containing a set of functions, where each such function calls either AiNeed(), AiResearch() or AiSleep(). AiNeed() is a substitute for both the Build and Train instructions, AiResearch() a substitute for the Upgrade instruction, and AiSleep() tells the AI to idle for a given amount of game cycles. Since scripts are read by the game engine in such a way that multiple actions can be performed at the same time, we use idling to force the scripted AI to do only one action at a time, making the game fair against Thanatos.

The scripts can be categorized as following.

Passive The passive player represents a peaceful community that will never engage in battle even though it might not prove to be a winning strategy. It builds a few workers, researches a couple of upgrades and expands to a second gold mine, doing nothing apart from gathering resources.

Offensive The offensive player is very similar to the passive one, but delays the upgrades until the expanding phase is completed. When the first upgrade is finished, the attack will commence. Two grunts, one axethrower and a catapult will be sent, one by one, towards the enemy base. If the enemy is not prepared,

defeat may be inevitable, but if the attack is stopped, the offensive player will be left defenseless.

Defensive The defensive player is identical to the offensive player, but instead of launching an attack at the opponent, the warfare units are kept in the base for defensive purposes.

8.4 Exploration and Reward Strategies

When training the Thanatos AI we want to try out different exploration and rewarding strategies to see how the learning performance can be influenced. In addition to a few standard schemes, we have developed an alternative exploration strategy called power exploration and a heuristic algorithm which uses game variables to adjust the given rewards. The following subsections describe the techniques that will be used throughout our simulation experiments in section 9.

8.4.1 Constant Randomness

We draw a uniform random number $r \in [0, 1)$ and test whether $r < \rho$ for some constant $0 \leq \rho \leq 1$. If the test returns true, we draw a random action, otherwise we select the best action possible in the current state. This can be illustrated by the following equation

$$P(a|s) = \begin{cases} \text{Random action} & \text{if } r < \rho \\ \text{Best action} & \text{if } r \geq \rho \end{cases} \tag{6}$$

This strategy does not take into account whether we have trained the AI for an hour or a millennia, nor how well it has been trained. It merely describes a simple probability of choosing a completely random action, in any state the expert may be in.

8.4.2 Boltzmann Exploration

Unlike constant randomness, Boltzmann Exploration takes previous training into consideration. The idea is to explore more in the early stages of the simulation by selecting actions randomly, and later on use the accumulated knowledge to select more of the promising actions.

In a given state s, each action $a \in A_n$ is given a probability according to

$$P\left(a|s\right) = \frac{\exp^{\frac{Q(s,a)}{T}}}{\sum_{j=0}^{n} \exp^{\frac{Q(s,a_j)}{T}}} \tag{7}$$

where $Q(s, a)$ is the current Q-value of selecting action a from state s and T is the temperature given by

$$T\left(g\right) = \exp^{-gT_s} \tag{8}$$

where T_s is a scalar constant and g is the current game number in the given simulation.

When more and more games have finished, (8) gives us smaller and smaller temperatures, which yields higher exponential values in (7). A larger Q-value means a bigger slice from the probability cake. Thus, the boltzmann strategy selects more and more solid actions the later we get into the simulation.

8.4.3 Power Exploration

Even though Boltzmann exploration proclaims a change of pace regarding random actions throughout a simulation, it does so heavily dependent on the actual trained Q-values. Power exploration provides a means of having a similar behaviour without taking the Q-values into account. We can achieve this by multiplying ρ in (6) with a variable $\Omega(g)$, that changes over the course of a simulation. We have

$$\Omega(g) = 1 - \left(\frac{g}{G}\right)^{\frac{1}{\sigma}} \tag{9a}$$

$$P(a|s) = \begin{cases} \text{Random action} & \text{if } r < \rho \cdot \Omega(g) \\ \text{Best action} & \text{if } r \geq \rho \cdot \Omega(g) \end{cases} \tag{9b}$$

where g is the current game number, G is the total number of games in a simulation and σ describes the duration of high probability randomness.

$\Omega(g)$ will be calculated at the beginning of each game and will determine how big the probability for drawing a random action at any given state in that particular game is.

The constant σ determines how $\Omega(g)$ varies throughout a single simulation, and as can be seen in Figure 22, higher values of σ results in an earlier drop in terms of the random action probability. In other words, if we want to keep exploring for quite some time, and then switch to the most suitable actions, we would select a fairly low value for σ.

A disadvantage with this strategy however, is that it requires prior knowledge of the total amount of games to be run. This makes it less trivial to extend an existing dataset with more games if such a situation is desired.

8.4.4 Heuristics

The rewarding strategy discussed in section 8.2 is quite fair since it does not take into consideration what expert human players or the RTS community consider is a good or a bad play. However, rewarding only wins and losses leaves for the possibility of having the AI play parts of a game extremely well, only to be beaten by an even better opponent, and thus getting a negative reinforcement. The overall reaction might thus be that this was a bad strategy, which might not be the case. We therefore wish to introduce a more advanced rewarding policy.

We will be using two game variables that we believe are related to how good a specific build order is; the game cycle number and the total income.

With the game cycle number, we are really referring to the current game cycle when a game has ended, i.e. the ending cycle. We will construct the heuristic algorithm in such a way that we give greater rewards for a shorter game.

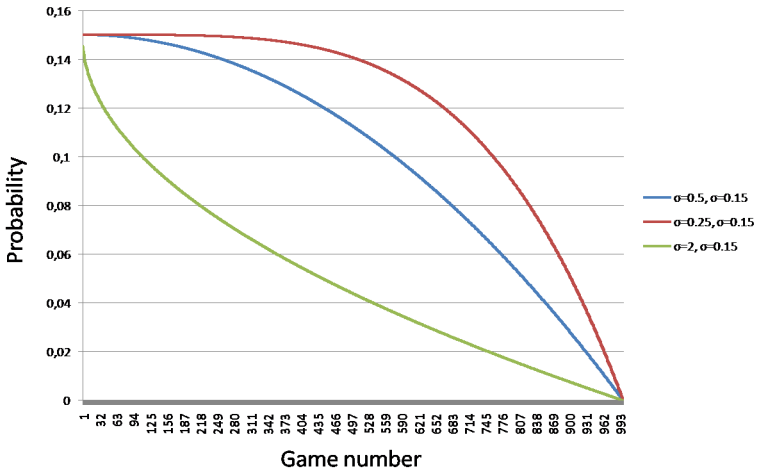

Figure 22: Illustrates power exploration with different values of σ

The other variable is the total income. This is the total amount of gold and wood harvested throughout the game, added together. On a very general level, the higher the amount of resources you can harvest throughout a game, the more promising your strategy is. Therefore, the reward will be greater the more resources you collect.

Our policy will be constructed based on a few observations and assumptions.

- It is unlikely for the AI to win a game within the first 10000 game cycles. This is due to the fact that at least two buildings are needed to train a unit, the unit must be trained, the map must be traversed by the attacker, and buildings razed for the player to be able to win. Giving a reward this early should therefore not be of concern. However, the reward will be capped in case of extremely early wins, or an enormous amount of resources harvested.
- An estimated reasonable length of a single game is 50000 game cycles. This should be more than enough time for either a learning Thanatos AI player, or a static script to develop a steady economy, build a reasonably large force, and finish off the game. We will be using 50000 as the game length throughout our simulations in section 9.
- A fairly strong strategy should enable a player to beat the opponent early, and thus 25000 will be set as a turning point for the policy. After 25000 game cycles, rewards will be lowered at a faster pace to enforce a more rapid strategy.
- The game result should still be taken into consideration, however less than the plain win/loss reward policy. A fraction representing how much the win/loss results should contribute to the reward must be specified when running the algorithm.

We estimated how much income could be harvested throughout a single game by

running an experiment and inspecting the results. We trained the Thanatos AI against itself with (6) for 500 games in a slightly modified environment. Instead of ending the game when a player got defeated, we let it play out the entire 50000 game cycles before the game was stopped. We outputted the amount of harvested resources every 2500 game cycles to a file, over a course of 100 additional games.

With the resource data at hand, we made a plot in MATLAB, describing the average amount of harvested resources at a given point in time. We then used regression to find a function that fitted our plot well. The entire algorithm was split into two separate parts, so the distribution of rewards could reflect greater values in faster games with solid economy, and lower values in long and poor games. By multiplying the second algorithm part, the one describing the latter game cycles, with an exponential value dependent on the specific ending cycle, we force the values to drop fairly quick as the game length tends to get higher.

The algorithm can be viewed as a set of equations

$$\tau = \begin{cases} \alpha \cdot \text{reward} & \text{if game won} \\ -\alpha \cdot \text{reward} & \text{if game lost} \end{cases} \tag{10a}$$

$$f(t) = \frac{I_t(G) + I_t(W)}{8.29 \cdot 10^{-6} t^2 + 0.21t + 1} \tag{10b}$$

$$F(t) = \begin{cases} 1 & \text{if } f(t) \geq 1 \\ f(t) & \text{if } f(t) < 1 \end{cases} \tag{10c}$$

$$R(t) = \begin{cases} 2\left(1-\alpha\right)\left(F(t) - 0.5\right) + \tau & \text{if } t < t_0 \\ 2\left(1-\alpha\right)\left(F(t)e^{-\left(\frac{t-t_0}{16000}\right)^2} - 0.5\right) + \tau & t \geq t_0 \end{cases} \tag{10d}$$

where $\alpha \in [0, 1]$ denotes how much the plain win/loss strategy should be calculated into the final reward. $I_t(G)$ and $I_t(W)$ are the total amount of gold and wood harvested when the game ends, respectively. $f(t)$ describes the regression function found and $F(t)$ maps the larger values onto the range $[0, 1]$. $R(t)$ denotes the calculated reward, taking on different equations when the ending cycle t is either greater than or lower than $t_0 = 25000$. We subtract 0.5 in equation (10d) to get a reward symmetrical around 0, and multiply by 2 to adjust it to the range $[-1, 1]$. Since α denotes how many percent of the total reward consists of the win/loss policy, $1 - \alpha$ denotes the percentage of the heuristic policy.

Figure 23 illustrates two example incomes that could occur during a simulation. If the game ends at a particular time (see the x-axis) you receive the reward given along the y-axis. From this example we also see that if the game ends before game cycle 25000, with an income of 10500 or above, the reward reaches the maximum limit at 0.5, given an $\alpha = 0.5$.

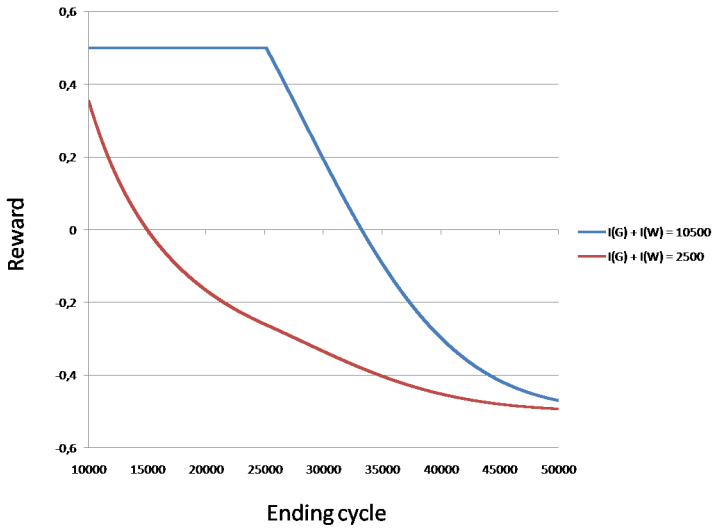

Figure 23: Illustrates the heuristic algorithm cap and how the reward may vary with respect to the total resource income and the ending cycles

9 Experiments

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

— *Albert Einstein*

In this section we will be running a series of experiments to see how well Thanatos is able to learn using the different exploration and rewarding strategies proposed in section 8.4. In addition, some of the trained sets will be played against each other to see which one has the most efficient build order.

Table 1 illustrates the set of experiments we will be running. These can be located in the *src/datasets* directory on the attached DVD and loaded with the "File→Open Simulation" functionality in the GUI control panel. For each experiment we will briefly point out the main aspects of the resulting build orders by disussing how the AI performs during runtime against its opponent. The experiments can be run and build orders derived by using the "Calculate Build Order" functionality with the corresponding experiment. Therefore, apart from one exception, the build orders will not be illustrated in this thesis.

Parameter choices will be pointed out, followed by a short note on what we should expect to see. The results will be discussed and a conclusion presented in section 10 on page 91. See appendix B for information on what the various graphs illustrate and how to use the software.

Table 1: The experiments performed

Experiment 1	Constant Randomness with $\rho = 0.15$
Experiment 2	Constant Randomness with $\rho = 0.5$
Experiment 3	Boltzmann Exploration with $T = 0.002$
Experiment 4	Power Exploration with $\sigma = 0.5$
Experiment 5	Power Exploration with $\sigma = 2$
Experiment 6	Heuristic Constant Randomness with $\rho = 0.15$
Experiment 7	Heuristic Boltzmann Exploration with $T = 0.002$
Experiment 8	Heuristic Power Exploration with $\sigma = 0.5$
Experiment 9	Thanatos vs Passive Script
Experiment 10	Thanatos vs Offensive Script
Experiment 11	Thanatos vs Defensive Script

9.1 Simulation and Learning Parameters

Since there are several strategy independent parameters that must be specified in the control panel, we will decide on a fixed set of values we will be using throughout our experiments. This will also give us a far more generalized foundation for comparing the observed results.

The first parameter we will decide is the number of games in a single simulation. Given that a fair amount of new states are explored throughout the entire simulation, a higher number will lead to a bigger exploration percentage of the actual state space (8.1.3). However, increasing the number of games also inceases the required runtime.

As games that progresses into a stage where resource gathering is established tend to traverse quite a lot of states, covering the entire state space should not be of our greatest concern. To some extent, it is the actions in the beginning of a game that are of most importance to develop a successfull build order. A thousand games per simulation should thus be a nice fit for our research. With an average game length of about 20 seconds on a standard computer, it will take approx. 5-6 hours to run a single simulation, which is quite reasonable for our time restrictions.

The average game length, mentioned above, depends heavilly on how long a particular game is. The maximum number of game cycles is our measure of how long a game can last before it is considered a draw, and either one or both of the players are rewarded with a loss (8.1.5).

As constructing a single structure can take more than 1000 game cycles alone, traversing the map about 500, and the fact that gathering wood is extremely slow, we select a fairly high game cycle cap. We will therefore set this value to 50000 and reward all Thanatos players with a loss if the game lasts this long.

As introduced in section 3.2, we need to select values for α and γ. [Mil06] proposes $\gamma = 0.75$ as an initial value based on the author's experience. This seems like a reasonable value as this uses quite a lot of recent information, but at the same time, as much as $0.75^{10} = 0.05$, or 5%, of the rewards 10 steps ahead will contribute to the current reward.

For the discount rate we also use the value 0.75. This should make for a nice balance between the newly acquired and the already learned information.

The last general parameters we need to specify are the rewards given at the end of each game. When a bad action has been performed, we would like the other possible unexplored actions to be regarded as better choices for the next game, rather than the bad one. As Q-values are initialized to 0, we can achieve this by punishing losses with a negative value and rewarding wins with a symmetrical positive value.

Since the Boltzmann exploration algorithm, expressed as equation (7), uses the exponential function on the Q-values, we should select fairly low rewards to avoid overflow or other representation problems. A punishment of -1 for a loss and a reward of 1 for a win thus seem like reasonable values.

We will be using 50 for both the "performance skip" and the "trend value". This practically means that when we plot our resulting graphs, we disregard the first 50 games in a particular simulation. This is due to the large amount of randomness that occurs during the first games, which makes the graph look less representative for the actual results. It also makes the trend graph illustrate the average win percentage of the last 50 games, which gives us a closer look at how the strategies evolve.

9.2 Thanatos vs Thanatos

In this section we will be running experiments with two instances of Thanatos playing against each other. We achieve this by selecting "Thanatos vs Thanatos" in the AI type dropdownbox in the GUI control panel.

Because of map symmetry and equal exploration strategies for both players, the game should be fair. However, ingame unit damage is influenced by randomness, and combined with the strategy dependent randomness, we expect to see player 1 emerging as the winner in some experiments, and player 2 in others.

We also expect to see multiple changes in the "current winning player" trends. In other words, when a player has learned a strategy enabling it to beat the opponent, it receives positive rewards, even though it may be trying out some new less optimal states. In the meantime, the other player only receives negative feedback, and thus can be thought of as searching for a counter-strategy. At some point, despite positive rewards being given to the winning player, the decisions tend to get somewhat random, since every move results in a win. The losing player can thus be able to surpass the winning one and turn the tables.

Despite oscillating winning trends, we should be able to see the overall build order strategies improving throughout the simulation.

9.2.1 Q-value Variation

To see how the learning progresses and that the AI is actually learning, we will look at a few Q-value variation graphs, as displayed by the GUI control panel. These graphs show how the Q-value of a single transition varies over the course of a simulation and can thus explain how well a state transition was thought to be at a particular point in time. We will be using Q-value variation graphs corresponding to experiment 4 (9.2.4).

Figure 24 displays the five possible structures that can be constructed when a game has just started and the structure expert is in its initial state. As we can see, the Q-values are always less than zero. This is due to the fact that the AI is losing more than it is winning (Figure 32), and thus the value closest to 0 is regarded as the best action currently possible.

We see that for most of the simulation, and especially after game 500, action 2, i.e. transitioning to state 5, is the best alternative. In other words, constructing a great hall is considered to be the best move in the beginning of a game. Constructing anything else than a great hall as the first building will lead to either a unfortunate or totally hopeless economical situation, as gold cannot be harvested, resulting in an inevitable loss.

Figure 25 illustrates the next possible set of structures that can be built. In this case, building another great hall is less optimal than building most other buildings. This is because it is quite expensive and almost all of the initial resources have been used. With only one worker available, harvesting resources is a very slow process, especially gathering additional wood. Instead, it is reasonable to believe that a farm is more optimal, as it makes training more workers possible. This is exactly what we see in the Q-value variation graphs, and thus the second step in the build order seems to be trained successfully.

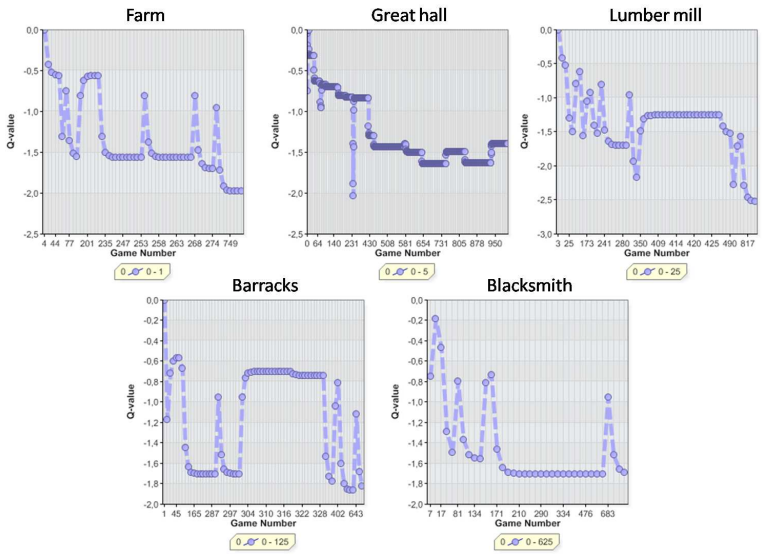

Figure 24: Q-value variation for the structure expert, showing the five possible actions in the initial state

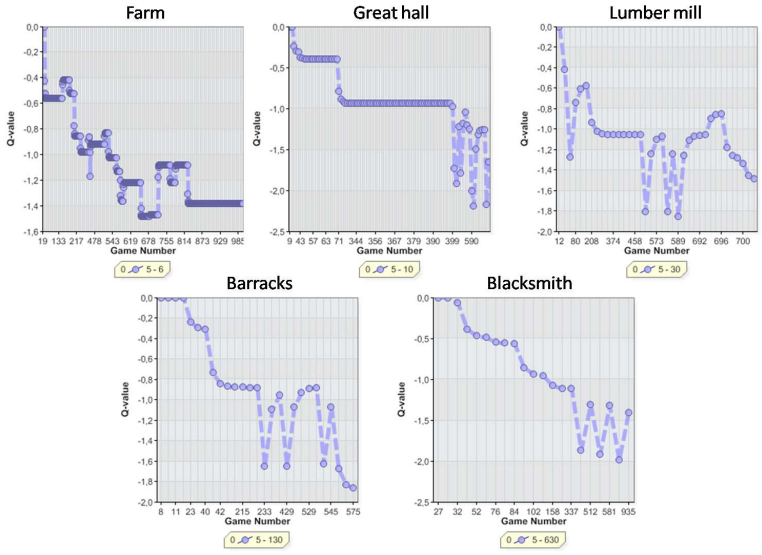

Figure 25: Q-Value variation for the structure expert, showing the five possible actions in state 5, one large hall constructed

9.2.2 Constant Randomness

When running experiments with constant randomness we would like to see how a low and a fairly high value of ρ may influence the learning. In experiment 1 we will use $\rho = 0.15$. In any given state, this indicates a 15 % probability of selecting an action by chance.

As the randomness is constant, we would expect several cases of "ups and downs" in the trend graph. Since games where the first action is not a great hall result in a guaranteed loss, we have the probability

$$P(\overline{\text{great hall}}) = 4 \cdot \frac{\rho}{5} = 4 \cdot \frac{0.15}{5} = 0.12$$

of losing because of this first incorrect move. In other words, about every 8th game is a certain loss for both players.

For $\rho = 0.5$ we would expect even greater oscillations. Here we have the probability

$$P(\overline{\text{great hall}}) = 4 \cdot \frac{\rho}{5} = 4 \cdot \frac{0.5}{5} = 0.4$$

of constructing something else than a great hall as the first structure, and thus a certain loss. We should be able to see some improvement throughout the simulation, but due to the heavy amount of random states traversed, it is possible that Q-values get so much random influence that the learning is occasionally wiped out and started all over again.

Experiment 1 In Figure 26 we see that both players have about the same distribution of wins and losses. Player 1 has a few more wins, which can also be seen from the leftmost chart in Figure 27.

Viewing the learning performance, we see quite a lot of "ups and downs" as predicted. We see that player 1 achieves a rather strong strategy which ensures a fair amount of wins early on. Player 2 is then able to counter before the situation turns around again.

As the simulation progresses we see that the overall build orders tend to improve to a win percentage of about 35 %. The trend illustrates two large winning streaks for both players towards the end of the simulation. This may indicate that it gets tougher and tougher to find a good counter strategy, meaning that the learned build orders are now fairly good, or possibly that some of the first moves have been "unlearned" for the losing player.

Looking at the resulting build orders for player 1 we see that it is a winning strategy, however a weak one. The strategy is slow, choosing to build several expansions, which are poorly utilized, and upgrades of units that are not even trained. Player 2 is currently on a hard losing streak, having canceled out even its first actions, leading to definite losses and search for a winning strategy in the upcoming games. This thus leaves player 1 in a loop of easy wins where each random move is considered a positive one, even though it is not, as discussed in the latter parts of section 9.1.

Experiment 2 Figure 28 shows that almost twice as many games as in experiment 1 are lost due to time running out. This happens either when a financial dead end is met, a passive strategy has emerged or both players are using

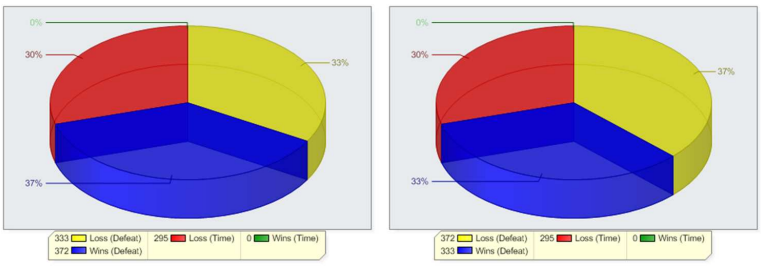

Figure 26: Experiment 1 - Win/Loss ratio for Constant Randomness with $\rho = 0.15$

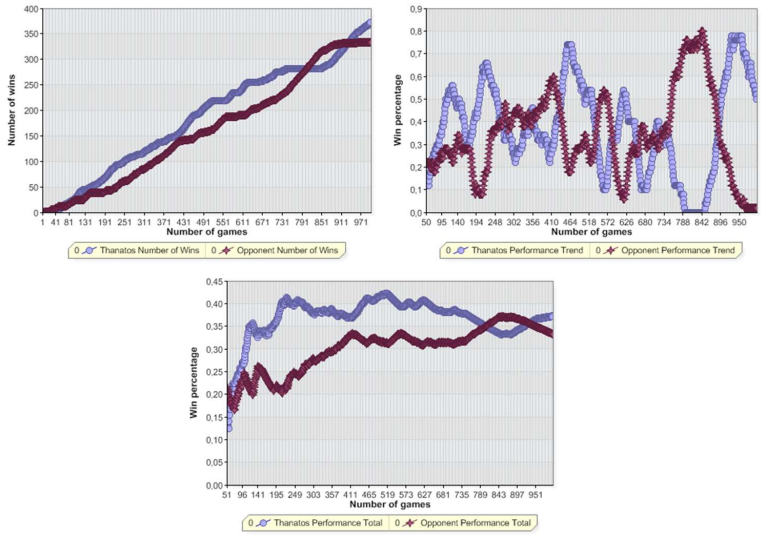

Figure 27: Experiment 1 - Performance graphs for Constant Randomness with $\rho = 0.15$

strong strategies, with neither one able to defeat the other in time. Looking at the charts in Figure 29, we see that the oscillating trends are characterized by randomness as expected.

The overall winning percentage climbs to about 22 %, which is clearly lower than in experiment 1. The percentage starts to flatten out after about 300 games and does not increase significantly after this. This effect may arise because of the many random actions selected, canceling the learning as discussed briefly in the beginning of section 9.2.2.

As in experiment 1, the trained build orders are not particularly great, seen from a human's point of view. Player 2 is trying to construct something else than the great hall as its first move, resulting in a certain defeat. Player 1 on the other hand gets a solid economy up and running, but does so very slowly. It actually constructs two additional great halls before anything else. This is very slow and makes the player extremely vulnerable against early attacks, as discussed briefly in section 9.2.1. However, it constructs a warrior and is able to win before the game ends.

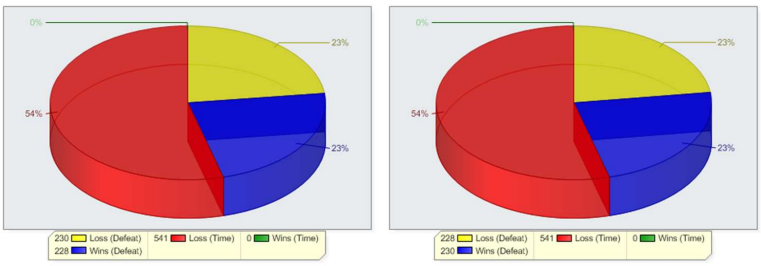

Figure 28: Experiment 2 - Win/Loss ratio for Constant Randomness with $\rho = 0.5$

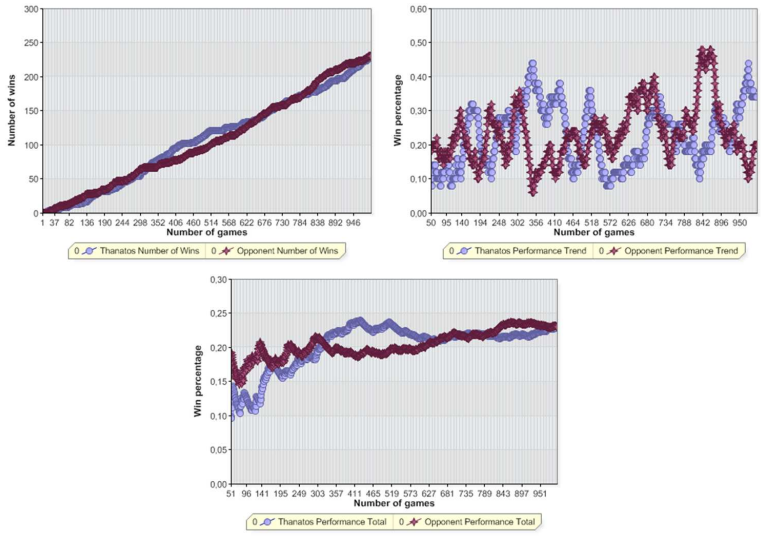

Figure 29: Experiment 2 - Performance graphs for Constant Randomness with $\rho = 0.5$

9.2.3 Boltzmann Exploration

Before we run simulations using bolztmann exploration, we need to decide the parameter T_s. Because of the exponential functions used in equation (7), described in section 8.4.2, higher Q-values lead to exponentially higher probabilities. The temperature controls how large these values can grow, so we need to investigate how it changes over time.

By experimenting with some values, we find that the temperature scalar $T_s = 0.002$ is a rather good choice. Looking at the possible minimum and maximum values for the varying temperature, using equation (8), we get the following

$$T(g_{\min}) = T(0) = e^{-0 \cdot 0.002} = e^0 = 1$$

$$T(g_{\max}) = T(1000) = e^{-1000 \cdot 0.002} = e^{-2} \approx 0.135$$

We thus see that the temperature used in equation (7) lies within the range $[0.135, 1]$. By using 1 and -1 as reward and punishment, respectively, we assume

that a single Q-value never lies outside the range $[-3, 3]$. From Figure 24 and 25 we see that the Q-values mostly lie within $[-2, 0]$, but at some occasions drop as low as -2.5, making our assumption reasonable.

With respect to equation (7), we should never get values larger than $e^{\frac{3}{0.135}} \approx e^{22}$. This is an enormous, but feasible number.

The selected T_s should be able to give us fairly strong probabilities for better actions in the later games of the simulation. We would therefore expect to see less oscillations in the winning strategies, the further we get into the simulation. If a player is currently on a winning streak, breaking this habit will be difficult for the opponent, as little or no new states are explored.

Experiment 3 Figure 30 and 31 show us that player 2 has had a better run than player 1, winning nearly twice as many games. The first 300 games seem to be quite chaotic, which is reasonable to expect since the Q-tables are untrained, and since the simulation should yield more exploration.

At approx. game 350, the oscillating wavelengths seem to get somewhat larger for a period, most likely due to the fact that the temperature, now at $e^{-350 \cdot 0.002} = e^{-0.7} \approx 0.496$, makes actions with greater Q-values a lot more probable. There is still enough randomness involved to defeat a winning strategy, or allow for a losing strategy to improve. However, at about game 700 this seems to end, resulting in a victory for player 2.

Looking at the resulting build orders, we see that player 1 cannot establish a steady income since the first structure built is not a great hall. This is not very surprising since it has been losing the last 100 games or so, resulting in negative feedback to many of the traversed states, propagating the negative feedback to the initial states.

Player 2 however was on a winning streak with a pretty solid strategy when the exploration gradually stopped. In fact, the strategy is a very effective, "one grunt rush" attack based on a low income economy.

The overall learning thus seems to be somewhat better than for constant randomness, especially for the winning player, but at the consequence of the other player being extremely bad.

Training Thanatos with boltzmann exploration thus seems to end up with a player that more or less can consistantly beat its opponent, given that one of the players actually managed to learn a winning strategy as the temperature dropped.

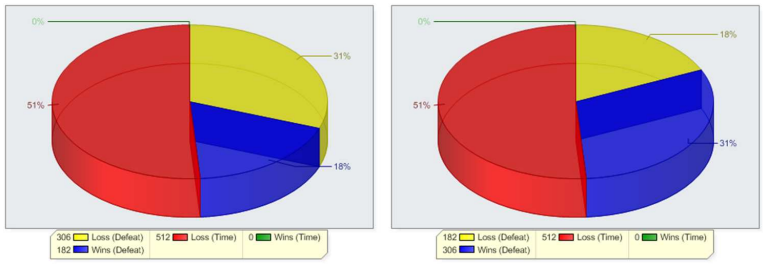

Figure 30: Experiment 3 - Win/Loss ratio for Boltzmann Exploration with $T_s = 0.002$

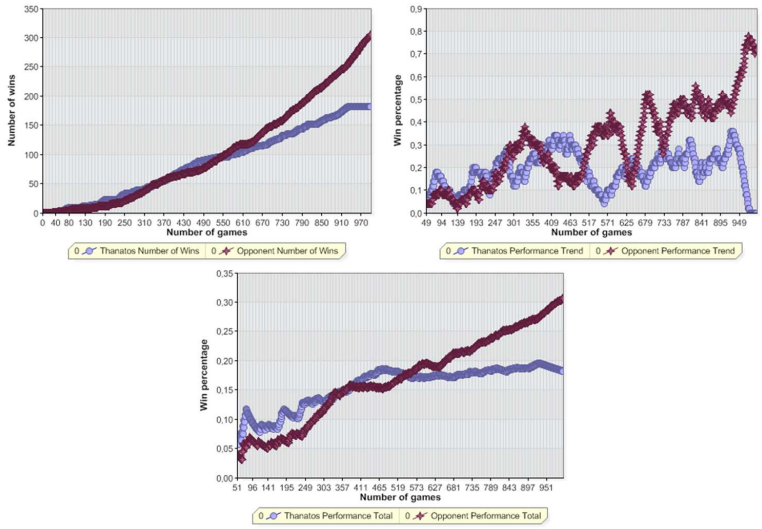

Figure 31: Experiment 3 - Performance graphs for Boltzmann Exploration with $T_s = 0.002$

9.2.4 Power Exploration

For power exploration, we will see how two of the values for σ plotted in Figure 22 in section 8.4.3 on page 67 affects the learning. The first experiment will use $\sigma = 0.5$, which slowly decreases the value of $\rho \cdot \Omega(g)$ in the early phase of the simulation, and speeds up a bit when about 500 games have been played. The second one will use $\sigma = 2$, which drops really fast in the beginning, slowing somewhat down in the latter parts of the simulation. In addition, we will use $\rho = 0.15$, as the two first experiments indicated that this was a rather good choice.

For experiment 4, equation (9a) thus gives us

$$\Omega(g) = 1 - \left(\frac{g}{G}\right)^2$$

As for experiment 3, with boltzmann exploration, we should see less random actions toward the end, but much less so with power exploration. In the early parts of the simulation, we should see a similar behaviour to the results in experiment 1, as the probability of doing random actions are kept fairly equal. This should involve quite a lot of random oscillating, but should nonetheless start to cool down after about 500 games.

For the second experiment, equation (9a) becomes

$$\Omega(g) = 1 - \sqrt{\frac{g}{G}}$$

Even though the probability drops quite fast, we should still see a lot of randomness occur in the early phase of the simulation. This is due to the fact that the amount of unexplored states is vast. Additionally, since most games in the beginning will be losses, the actions not yet tried will be regarded as better.

The overall learning does not have to be any less fruitful than in experiment 4, but due to the significant drop in the random exploration probability, we expect to see a somewhat less optimal build order.

Experiment 4 In Figure 32 we see that both players win a fair amount of games. They experience many losses in the beginning, when suddenly player 1 finds a somewhat good strategy, sending it on a short winning streak. Then player 2 discovers a counter strategy, and some wild oscillations follow as seen in Figure 33.

As expected, at around game 500 it seems like the randomness has dropped significantly, disregarding what seems like a chaotic part at around game 750. There are still ups and downs, which indicate that both players still are able to adapt their strategies to overcome the opponent.

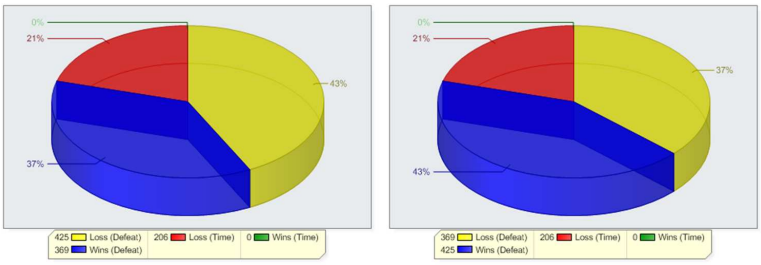

Figure 32: Experiment 4 - Win/Loss ratio for Power Exploration with $\sigma = 0.5$

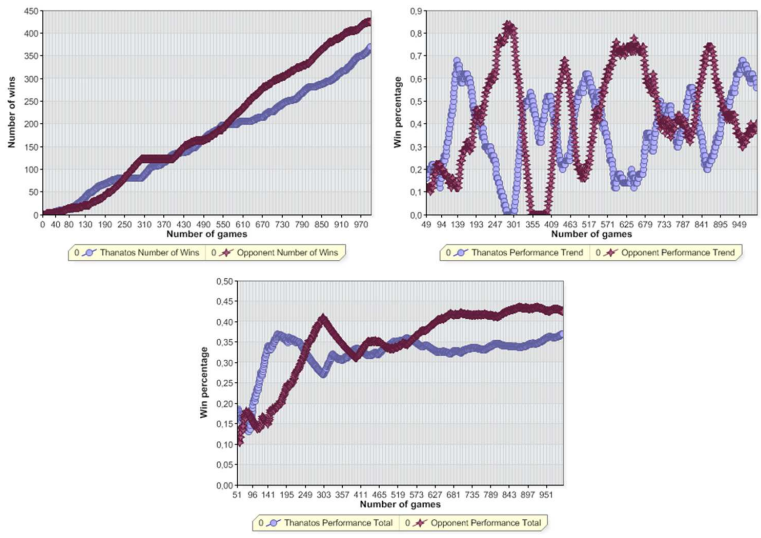

Figure 33: Experiment 4 - Performance graphs for Power Exploration with $\sigma = 0.5$

Unlike boltzmann exploration, where adapting a strategy towards the end of a simulation is extremely unlikely due to the high exponential values in equation (7), random actions are still chosen on a regular basis with power exploration.

Listing 4: Experiment 4: Excerpt of the resulting build orders

1	BUILD unit-great-hall	BUILD unit-great-hall
2	BUILD unit-pig-farm	BUILD unit-pig-farm
3	TRAIN unit-peon	TRAIN unit-peon
4	TRAIN unit-peon	TRAIN unit-peon
5	TRAIN unit-peon	TRAIN unit-peon
6	TRAIN unit-peon	TRAIN unit-peon
7	BUILD unit-pig-farm	BUILD unit-orc-barracks
8	TRAIN unit-peon	BUILD unit-pig-farm
9	TRAIN unit-peon	TRAIN unit-grunt
10	TRAIN unit-peon	TRAIN unit-grunt
11	BUILD unit-orc-barracks	TRAIN unit-grunt
12	TRAIN unit-grunt	TRAIN unit-grunt

This indicates a better chance of improving a strategy further. Since both players have learned a rather strong strategy, in addition to the small chance of exploring random actions, it is less likely than in other cases that one player "unlearns" its curernt strategy, and thus sends its opponent on a winning streak.

In figure 33 we can actually see that the players seem to have alternating winning streaks that gets shorter and shorter, closing in on a winning percentage of about 0.5 each. The total performance is still climbing when the simulation ends, indicating that the players actually have a potential to learn more.

For this experiment it is worth illustrating the produced build orders as they show very similar strategies for both players. In Listing 4 we illustrate the first 12 actions, with the leftmost being the losing player and the rightmost the winning one.

As we can see, the first 6 steps are identical. Actually, these steps are the exact same a human player would normally use. What happens next is critical and determines the outcome of the game. For what we have called the losing player, we see that an additional farm and more peons (workers) are purchased, before the barracks and a warfare unit is finally acquired.

The winning player jumps directly to the warfare units and is thus able to produce an earlier grunt than its opponent. The losing player is actually able to train its grunt in time of the attack, but the additional grunts of the winning player are overwhelming and it suffers a loss.

Experiment 5 In figure 34 we see that player 2 actually has a very high amount of wins, especially compared to the other experiments we have done so far. By inspecting Figure 35 we clearly see that the randomness we predicted early on is present, and that towards the end we have hard-to-beat strategies. Eventually the winning strategy of player 2 gets too strong, or possibly the strategy of player 1 gets too weak. This combined with the fact that few random choices are made makes it incredibly hard for player 1 to find a counter strategy, thus resulting in permanent loss.

When looking at the build orders, we see exactly this. Player 1 has a corrupt build order making every game a simple win for player 2. The build order player 2 has learned is actually not particularly strong either, as its main focus lies on

expanding and gathering resources. A small attack force is launched at the opponent and ends the game close to the game cycle cap. It seems as this value for power exploration has the same destiny as experiment 1, i.e. a fair chance of one player winning a huge amount of games in a row, when random exploration is still able to influence the choices. This way, it is actually possible that the winning build order gets less optimal with an additional amount of games to be played.

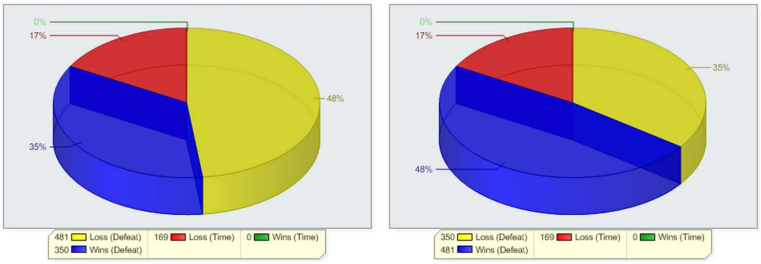

Figure 34: Experiment 5 - Win/Loss ratio for Power Exploration with $\sigma = 2$

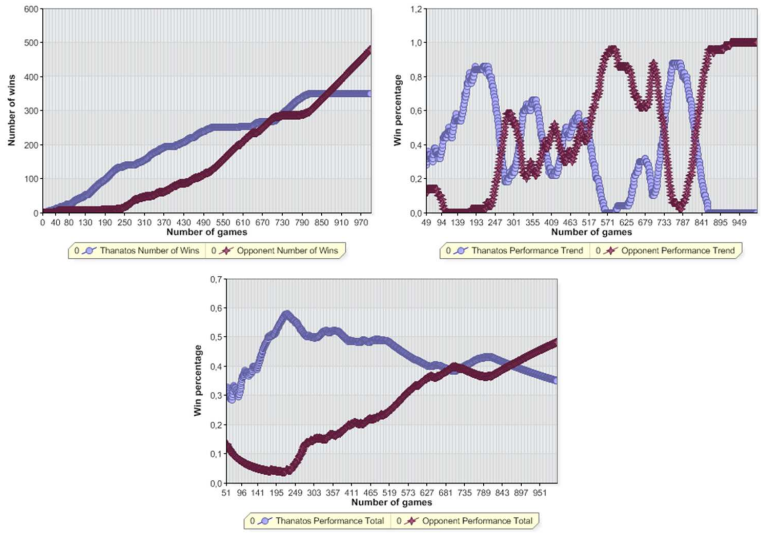

Figure 35: Experiment 5 - Performance graphs for Power Exploration with $\sigma = 2$

9.2.5 Heuristics

In these experiments we will use the heuristic algorithm which was explained in section 8.4.4. We will be using $\alpha = 0.5$, so 50 % of the reward is based on the result of the game, and 50 % on income and game length.

We will run one experiment for each of the three exploration strategies and see whether the learning has been improved or not. Parameters will be set according to the experiments already performed. In other words, we will be using the following parameters; $\rho = 0.15$, $T_s = 0.002$ and $\sigma = 0.5$.

We do not expect the learning patterns to be significantly different than in the original experiments since the exploration strategies are kept identical. However, overall learned strategies should tend to finish games faster and be influenced by a higher income of resources and thus result in more powerful strategies.

The performance graphs of all heuristic experiments are gathered and illustrated in Figure 36.

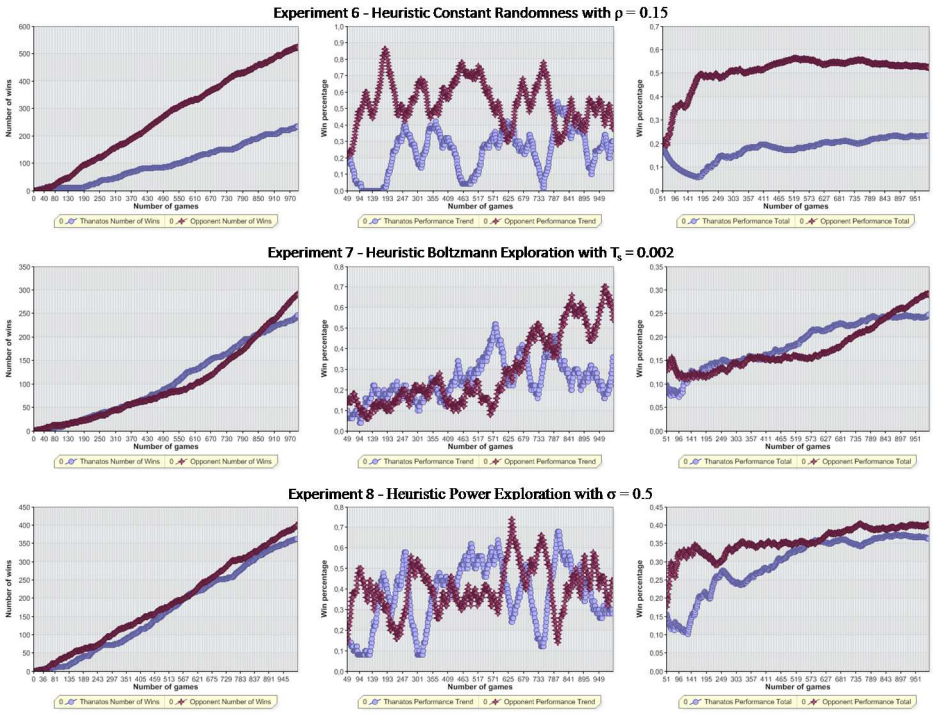

Figure 36: Experiment 6, 7 and 8 - Performance graphs for Heuristic experiments

Experiment 6 Looking at the graphs, we see that player 2 is quite dominant throughout the entire simulation. Player 1 is able to turn the tide every now and then, but a counter strategy is always found by player 2 before player 1 is able to turn the tables.

Even though the graphs are somewhat different than in experiment 1, we must recall that early games are entirely random, and with $\rho = 0.15$ there is still quite a lot of randomness involved.

As there is a huge number of ways to win a single game, and actions are explored on a random basis, the first winning strategy may differ greatly from simulation to simulation. It is therefore quite possible to see entirely different winning patterns, as some strategies may be hard to counter right away, others not. However, the overall learning ability of the resulting build orders is restricted by the given exploration strategy and its parameters.

Looking at the resulting build orders we see that player 1 is fairly slow, as it tries to expand right after building the first great hall. Player 2 has a very good economic start, using resources on structures and upgrades instead of warfare. After a while it attacks with a minor force which is repelled by a weak defense from player 1, before a huge final attack begins and the game ends right before the game cycle cap is reached.

Both players thus seem to have learned a rather strong build order, especially when compared to experiment 1 where player 2 was not even able to build the first structure correctly. This indicates that the heuristic algorithm may have rewarded the income of resources a bit too much compared to the game length.

Experiment 7 The boltzmann graphs are very similar to those in experiment 3, except for somewhat more chaos during the first 400 games.

Looking at the build orders we see that as for experiment 6, both players are able to establish an opening game with a great hall, a farm and several workers. The rest of the game plays out very similar to experiment 4, power exploration with $\sigma = 0.5$. Player 2 goes for an early barrack and follows up with a massive grunt attack, whereas player 1 constructs a lumber mill.

Player 1 can thus be thought of as being influenced mostly by the rewards for a high income, and player 2 by the rewards for ending a game swiftly. The heuristic algorithm thus seems to have improved the resulting build orders to some extent.

Experiment 8 Again the graphs are very similar to the original experiment, as seen in experiment 4. The total winning percentage climbs in a similar fashion, despite fairly large differences in the current winning trends.

As for experiment 6 and 7, both players start out with a great hall, a farm and some workers. Using heuristics seems to result in less "unlearned" build orders, which implies a better environment to further improve the learned strategies in. This may be because of the reward scheme of the heuristics, making rewards close to 0 possible, even when losing a game. Receiving such high rewards when losing a game results in less new explored actions than normal, and therefore a smaller chance of unlearning the better actions.

As for experiment 4, this simulation shows a very promising build order. A big attack is launched by player 2 early on and ends the game at about game cycle 25 000. Even though it is quite strong, this particular build order is not as efficient as the one in experiment 4.

9.3 Tournament of the trained

In section 9.2 we performed a series of experiments showing that the learned build orders tend to vary a lot with the different strategies and their corresponding parameters. We discussed how the strategies affected the learning, and briefly analyzed how the resulting build orders performed after the training phase. Even though we have established a few favourite build orders, we cannot tell which strategy is currently the best one. We will therefore perform another experiment, using the trained build orders in a single elimination tournament

style competition[14] to see how well they perform against each other.

We do this by pairing up the winning player from each experiment, ginving us four initial matches. The winner advances to the next round, facing an opponent from one of the other matches. The winner of the second round goes to the finale where a winner will be decided.

The Q-tables used in this experiment are located in the *src/datasets/tournament* directory on the attached DVD. To run this experiment, the files for a particular matchup must be copied to the GUI control panel folder and loaded using the "Reload Simulation" function. The game can then be started by pressing the "Calculate Build Order" button.

Figure 37 shows the tournament lineup, where a blue player indicates the winner of that particular match.

Not surprisingly the constant randomness player, with $\rho = 0.15$, wins the first match, but loses against the fast "power rusher" from game 2, in the first semi final.

The Boltzmann trained player defeats the slow constant heuristic player, facing its heuristic counterpart in the second semi final. This proves to be a very close match, where the heuristic player is barely able to win the game.

In the finals, both players have aggressive strategies, with very solid inital build order instructions. Even though the heuristic boltzmann player shows some strength, the power exploration trained player is attacking too fast and too many units. Seeing how close the best build orders are, it is fair to believe that another set of simulations could have gone in the opposite direction.

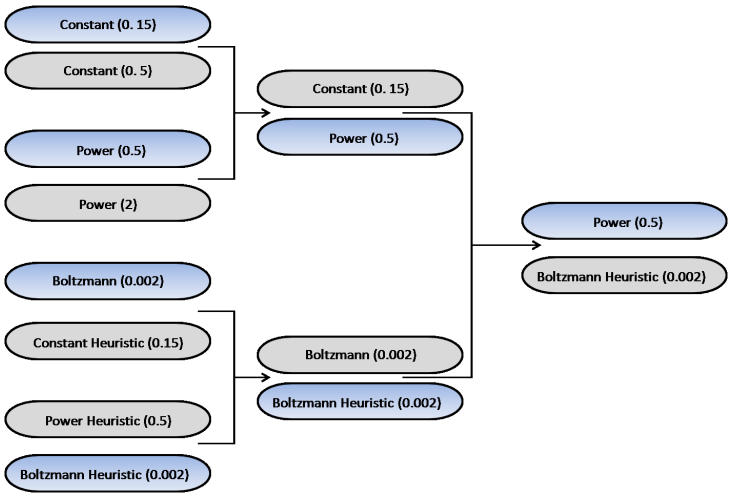

Figure 37: Build order tournament lineup, showing winners and losers for each match in the competition

[14] A type of elimination tournament where the loser of each match is eliminated from the competition.

9.4 Thanatos vs Static Opponents

The last experiments we will perform are against a set of predefined scripts. These were described in section 8.3.3 and their full build orders can be seen in appendix C.

With a slow but solid build order emerging in experiment 1, we will be using constant randomness with $\rho = 0.15$ as the exploration strategy for these experiments as well. This way we can see if playing against a fixed opponent results in a better strategy. As for the stronger exploration strategies, like heuristic boltzmann and power exploration, we assume that against a static opponent unlearning will be less of a problem. This should result in stronger strategies, making it hard to compare with the earlier obtained results.

The winning percentage of player 1 and 2 in these experiments are symmetrical due to the definition of wins and losses against scripts. Thanatos will be playing as player 1 and can be seen as the blue player in the graphs illustrated in the following experiments.

Against the passive opponent we expect to see a winning strategy emerge. How great this strategy will be is harder to predict, as actions will be rewarded with a positive feedback if just a single warfare unit is produced. This must however be done early enough to win the game before the time runs out. This could result in some quite random, low efficiency build orders able to win.

Facing the offensive opponent should force Thanatos to produce units early on to be able to withstand the aggressive opener. If this can be learned, a winning build order should also be possible to achieve, as there is plenty of time to finish off the battle when the offensive script's attack has been stopped. Further improvements could even lead to an earlier and more aggressive opener from Thanatos, resulting in a win before the opponent attacks.

The defensive opponent will be hard do defeat, as four units may be located in the same area at the same time if the game progresses too far. Since only one unit is sent to attack at a time it will most likely be overpowered and rendered useless by the opponent. The unit mix of two grunts, one axethrower and a catapult makes it hard to get close enough to deal any damage, unless Thanatos uses a catapult itself. If a winning strategy is to be learned in this experiment, it should emerge as a rushing strategy where the attack is started before the defensive units have been trained.

Experiment 9 As we can see from the trend graph in Figure 38 a winning strategy is already found at game 50, sending Thanatos on a big winning streak. Every now and then we see that it loses a few games however. Looking at the pie chart, we see that every game lost is due to time running out. This is because the passive script never trains any units that can be used to defeat Thanatos with. The losses must therefore be a result of either trying out random actions early in the game, leading to financial dead ends as discussed earlier in this section, or due to unlearning.

The huge losing streak at about game 650 is clearly not a product of having "bad luck" with initial guesses. This is most likely a result of having several of

the first few critical steps unlearned and being forced to relearn these to get back to a winning strategy.

Running a game with the trained set we see that Thanatos has learned a winning strategy, actually somewhat better than predicted in terms of speed and income.

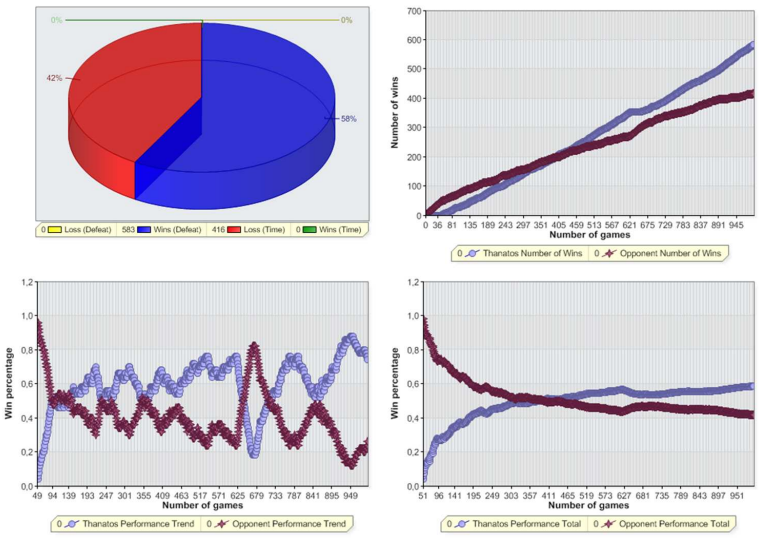

Figure 38: Experiment 9 - Graphs against a Passive script

Experiment 10 Even though it seems like the offensive script is running over Thanatos, winning an incredible 62 % of the games, the resulting build order is actually a winning one. In the trend graph we can actually see that Thanatos has just started winning after being on a long losing streak, when the simulation ends. If we were to stop the simulation only a few games earlier we could have experienced a losing build order instead.

Thanatos is actually able to learn a swift rushing strategy, as discussed earlier in this section, disabling the offensive script before it starts training warfare units itself. The opener is very much like some of the other strong ones we have seen with a great hall, a farm, some workers, and then a barracks and warfare units used to defeat the opponent.

Some games were lost due to time, meaning that neither player were able to finish the game, as seen in Figure 39. We could have rewarded these games with a positive reward instead of punishing them, as holding off the opponent should be considered to be a solid result against an offensive opponent.

Experiment 11 In Figure 40 we see that Thanatos has a serious problem winning. As much as 72 % of the games are lost, and only one winning streak is particularly noticeable. Even though the actual win percentage seems to drop somewhat over time, we are lucky enough to be on a winning streak when the simulation ended. A similar build order to the one seen in experiment 10 has emerged and thus finishes off the defensive script before it manages to purchase any warfare units at all.

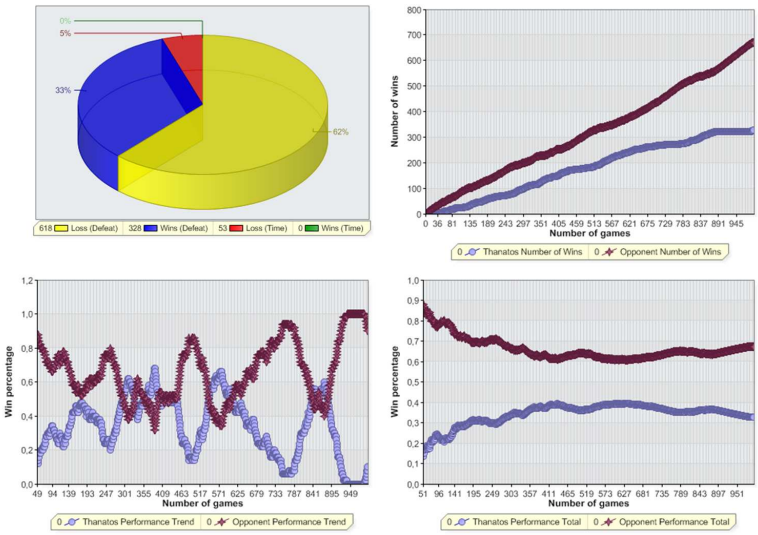

Figure 39: Experiment 10 - Graphs against an Offensive script

It is reasonable to think that due to periodically events of unlearning, closely connected to the constant randomness exploration strategy and the difficulty of finding winning strategies against this particular opponent, it will be hard for Thanatos to converge on a winning strategy.

We performed this experiment a second time to look for clues to support our claim that this experiment has so far given us a rather "random" outcome. The new results can be seen in Figure 41 and clearly show both what we initially predicted to see in this experiment and a clear indication that the first build order we got is not representative against this particular opponent.

The new build order shows that Thanatos has a rather slow development, thus letting the script train its defensive army. Every attack launched is stopped by the script, and thus leaves in a loss by time. This simulation has the absolutely highest amount of losses we have encountered this far.

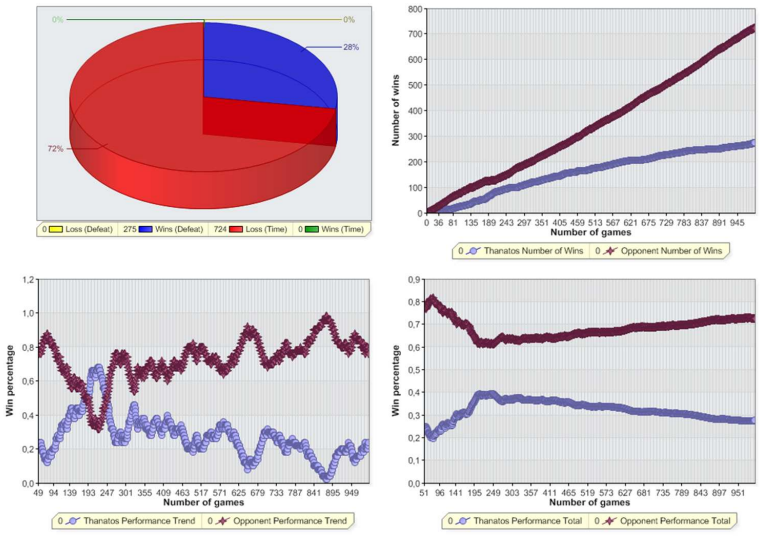

Figure 40: Experiment 11a - Graphs against a Defensive script

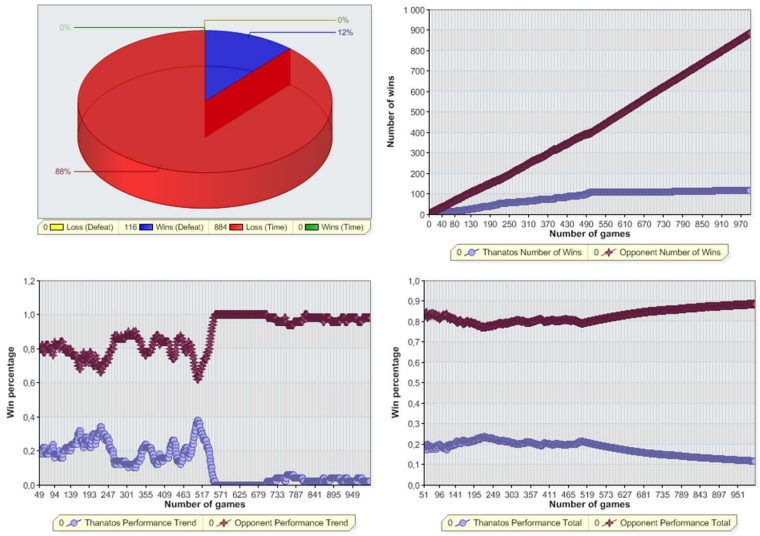

Figure 41: Experiment 11b - Graphs against a Defensive script, second run

Part VI

Outcomes

10 Conclusion

A conclusion is the place where you got tired of thinking.

— *Arthur Bloch*

Looking at the results presented in section 9, we see that they match our expectations well (1.3). With the approach we have proposed in this thesis, we see that Thanatos was able to improve its strategies throughout the simulation in every single experiment. Apart from the imbalanced game against the defensive script (9.4), Thanatos was able to learn winning strategies in every experiment as well. From this we may conclude that reinforcement learning can be used to learn and improve strategies for a CCP AI in an RTS game (5.1).

The experiments illustrate that various exploration strategies and different choices of parameters have an impact on how well Thanatos is able to learn and how successfull the resulting build orders can become. Power exploration resulted in the strongest build orders, defeating the others in a tournament between the trained AIs (9.3). Together with Boltzmann exploration, they both show that adjusting the amount of exploration throughout the simulation yields particularly stronger build orders than keeping the exploration constant, like in the constant randomness experiments.

The experiments further indicate that a paricular player's strategy improves more against an offensive opponent than a passive one. This effect can be seen in the experiments where Thanatos played against the static scripts. Against the offensive player, a strong and effective strategy was learned, whereas against the passive one, a less effective one emerged. Additionally, the defensive script experiment illustrates that the learning may fail if the opponent is too strong. In this experiment it was actually a case of "unfairness", since the defensive script gathers several units at the same location, whereas Thanatos applies to the "one-unit-at-a-time-attacks" concept.

The biggest problem for Thanatos was the occasional event of unlearning. When one player had to start learning almost from scratch, the other player could win the game with almost any strategy. Winning every game implies that every move is good, resulting in random behaviour and hence "unlearning". The heuristic algorithm helped to avoid this in experiment 6-8. In addition, heuristics improved the build orders to become stronger than the initial ones. The exception was for power exploration, where both were about equally strong. We therefore conclude that heuristics may improve the overall results when training the AI.

11 Future Research

Don't worry about people stealing your ideas. If your ideas are any good, you'll have to ram them down people's throats.

— *Howard Aiken*

With the limited time at hand, there were some experiments that could have been interesting to run and investigate, but never got to see the light of day. In this section we discuss some of these, as well as propose additional improvements that can be done to Thanatos, and briefly introduce a promising machine learning field with a possible interesting future in the RTS game domain.

11.1 Further experimentation with Thanatos

In section 9 we ran a number of simulations to see how the various exploration strategies affect the learning of Thanatos. We decided on a specific value for both γ and α, to reduce the number of experiments and make the comparison of those that were ran somewhat easier.

Even though the values we selected proved to be rather good, we have not yet seen if the results could be further improved. It is also interesting to see if some values actually lead to less fruiful results or even makes the overall learning of the AI break down. If the entire concept got expanded, these values may have to be adjusted, and thus a study of how they affect the learning would therefore be of great benefit.

Varying γ and α during a single simulation could also be a resourceful experiment, e.g. let α start with a value of 0.75 and be lowered to as little as 0.1 towards the end. This would enable Thanatos to use less newly acquired information the further the simulation gets, which possibly could increase the learning potential.

We restricted the game to 50000 game cycles, which made fairly long games possible, as well as speedy games with rush strategies. Pushing the game cycle down could possibly affect the emerged strategies to be even more rush based. Running longer games is also interesting, as this could make the learning somewhat harder, due to the vast amount of possible strategies that could lead to a victory. Unlearning could therefore be a bigger problem than what we experienced in our simulations.

11.2 Improving Thanatos

In section 8.1.2 we described how we represent and store state-action tuples to the hard drive. We also pointed out that there may actually turn out to be quite a lot of states that are never traversed, no matter how many games we run. Because of this, our $m \times n$ matrix thus stores a lot of numbers that are never modified.

Reducing the array may not be possible, as this would severly complicate any interaction with the Q-tables, or leave room for less states than needed. Additionally, it will not be easy to predict which states are unreachable during design time, as this is closely tied to the selected parameters, as well as possible random ingame events.

Instead of using a standard two-dimensional array, another way of storing and accessing the Q-tables could be used. One possibility that comes to mind is vectors, where traversed states could be added as they were explored.

In section 8.4.4 we developed a heuristic algorithm that we hoped would be able to improve the learned build orders of Thanatos. It provides a redefined reward model that includes both the resource income and the length of the game to reward better playstyles.

The RTS concept is a lot more complex than just resources and the game length, so including other relevant factors to address how well a particular strategy have been is something that could further improve the potential of Thanatos. Examples of such could be resources spent, structures not utilized and entities lost and destroyed.

Looking at the state space presented in section 8.1.2, we see that with additional supported features, the space required increases dramatically. Since the game used in this thesis have been simplified to some extent, and newer games tend to get more and more complex, it would be reasonable to investigate possibilities of handling larger state spaces.

In [RN03], an idea called function approximation is sketched, and discussed further in [IS05]. This concept learns a representation of the Q-function, as a linear combination of features that describe states. Adapting Thanatos to use such a concept could allow it to address more complex problems, like losing and rebuilding units, resource management, structure placement and general added game complexity.

11.3 HTM Networks

Instead of adapting the reinforcement learning related parts of Thanatos, new technologies and theories could be introduced and tested as well. A new uprising and promising field is HTM, or Hierarchical Temporal Memory. The idea is not a new one in itself, but is a combination of already existing ones. It is a machine learning model, sharing properties with bayesian (4.2.7) and neural networks (4.2.6), that attempts to model some of the properties of the neocortex, where a single node can represent a large amount of neurons and a trained network can be used for prediction.

> *HTM works best for problems where the data to be modeled are generated by a hierarchy of causes that change over time. Here, a cause is the object that caused the HTM input data. The problem should have both a spatial and a temporal component. [Num]*

If modelled in an appropriate way, using HTM for decision making and emerging build orders in RTS games could be both interesting and shown to be quite promising.

Part VII

Appendices

Appendix A

RTS Terminology

Explains some of the not so common words used in the thesis.

Attack damage The amount of damage dealt when striking an enemy. Section 7.3 illustrates how this is calculated in Wargus with the two equations in (2).

Build Order A set of instructions describing the decision that should be made when playing a game. See section 6.3.

CCP Computer Controllable Player, also generally referred to as the "AI" of a game.

Expansion An additional base used for resource gathering.

Health Points Often shortened to "HP". It is a measure of how much damage a given unit can take before being destroyed.

Race A categorization of units, structures and upgrades with different characteristics that a player must select to play with. The races used in this thesis are discussed in section 7.3.

RTS Real Time Strategy. The genre is described in section 3.

Structure A name for stationary entities that are purchased during the game. Structures are essential as they are used to both produce units and research upgrades.

Tile The game world is often represented as a matrix of tiles. Each tile may have different properties. In chess, each square is a tile.

Unit The name used for mobile entities, just like pieces in a chess game.

Upgrade An upgrade may improve the abilities of a specific unit or structure type during the game.

Appendix B

Using the Software

The application developed for this thesis is located on the DVD attached to this thesis and needs Windows XP (or later) with .NET framework 2.0 (or later) installed to run. There are three directories in the root folder on the DVD; bin, doc and src.

bin: Contains a ready to run version of the application with a full version of dotNetCharting© enabled. To run simulations or to save configurations between sessions, the application should be copied to an area with appropriate access levels, as it needs write access.

doc: Contains two files describing how to compile and run the application and a brief overview of the original framework files that have been modified.

src: Contains the directories master and wc2 that together forms the Wargus game module and the modifications made to it. Additionally there are two Visual Basic 2005 projects, Stratagus and GUI control panel. The Stratagus project is the c++ game engine, including the Thanatos AI and the framework for the Scripted AIs. GUI control panel however is a project written in Visual Basic and is the application used to run all the simulations. The control panel source code comes with a development version of dotNetCharting©.

When the GUI control panel is used to launch a simulation for the first time, software firewalls may ask for permission to access the internet. This is due to the fact that the control panel and the Stratagus game engine uses winsock to communicate with each other. A TCP/IP connection on the loopback IP address 127.0.0.1 is established on the specified ports and is needed to successfully run a simulation.

When a simulation starts, the Stratagus window handle is passed to the control panel so it can attach itself to it, giving an illusion of running a single application, instead of the two separate processes that are actually running.

The application provides the following functionalities

Live Feed: Shows the current states for all experts, and posts a live message whenever an action has been chosen, a reward is given, or an actual state

transition has occured.

Win/Loss Ratio: Displays the amount of wins and losses as a nice pie chart. The chart distinguishes between wins and losses due to time and to being defeated by an opponent force.

Win/Loss Distribution: A simple graph that plots the amount of wins for each game in the simulation. The steepness of the graph at a particular point describes the winning streak.

Learning Performance: The trend graph plots the average win percentage for the last n games for every game, except the first x games. x is denoted by the performance skip value in the settings tab, and can be used to skip a given number of games, as the early stages of a simulation usually involves a lot of losses, and therefor yields meaningless results. The trend value can also be modified in the settings tab to adjust the trend we want to view. Default values are set to 50 games.

The total graph shows almost the same, except that instead of showing an average of the last n games, it shows the total win percentage, i.e. the number of wins / number of games for each game that has been played.

Distribution Charts: Each expert has a corresponding distribution chart that illustrates the actions that have been made in the current simulated game. If a unit of type "peon" has been built, the peon gauge will be increased by one. Some units may be destroyed during the game, so this can serve as a reminder of the actions that have been performed, as well as an easier way to monitor the actions if you are not familiar with the game entities.

Figure B.2 illustrates how the unit distribution may look while the Thanatos AI is attacking an opponent during a simulation.

Transition Value Variation: Every time a delayed reward is distributed, i.e. the Q-tables are updated, for a given expert, we save the states involved, as well as the new Q-value. The transitions are easilly kept in a treeview, so that the user can select which transitions to study with ease. When a specific transition is selected, all of the Q-values for that particular transition are plotted in a single graph, illustrating how well it has been throughout the simulation.

Q-Value overview: The Q-values are stored in a set of data files, but can here easilly be viewed more systematically. The entire state space is listed to the left, where a particular state can be selected, showing the current Q-values for all valid actions from that state. See Figure B.3 for a screenshot of this functionality.

Simulation: This starts the simulation with parameters according to the settings tab.

Calculate Build Order: This will run a single game, with the given AI type and produce build order outputs for all Thanatos players. These will be located in the GUI control panel folder when the game has finished.

Some of the functionalities mentioned above have the option of switching between the data sets of player 1 and player 2.

Settings: The settings panel have been divided into four parts; Learning, Strategies, Simulation Details and Path. The path is the simplest one. It just points to the Stratagus executable file, and must be provided by the user. When the correct path is set, the refresh button can be used to load the supported game types for the Thanatos AI. Then set two free ports on the loopback IP 127.0.0.1. The default ones should do just fine under normal circumstances. If you wish to run more than one simulation at the same time, each instance of the GUI control panel will need two unique ports to communicate on.

Cycle defines how many game cycles the game should last for and Games tell how many games the simulation should consist of. The Time elapsed checkbutton will make sure that Thanatos players will either win or lose if the game is ended due to reaching the last specified game cycle.

Learning and Strategies values can be adjusted as one would want, but you can only select one strategy at the time. Heuristics will count as a rewarding strategy and can thus be used at the same time as any exploration strategy.

The live feed functionality and the settings tab are illustrated in Figure B.1.

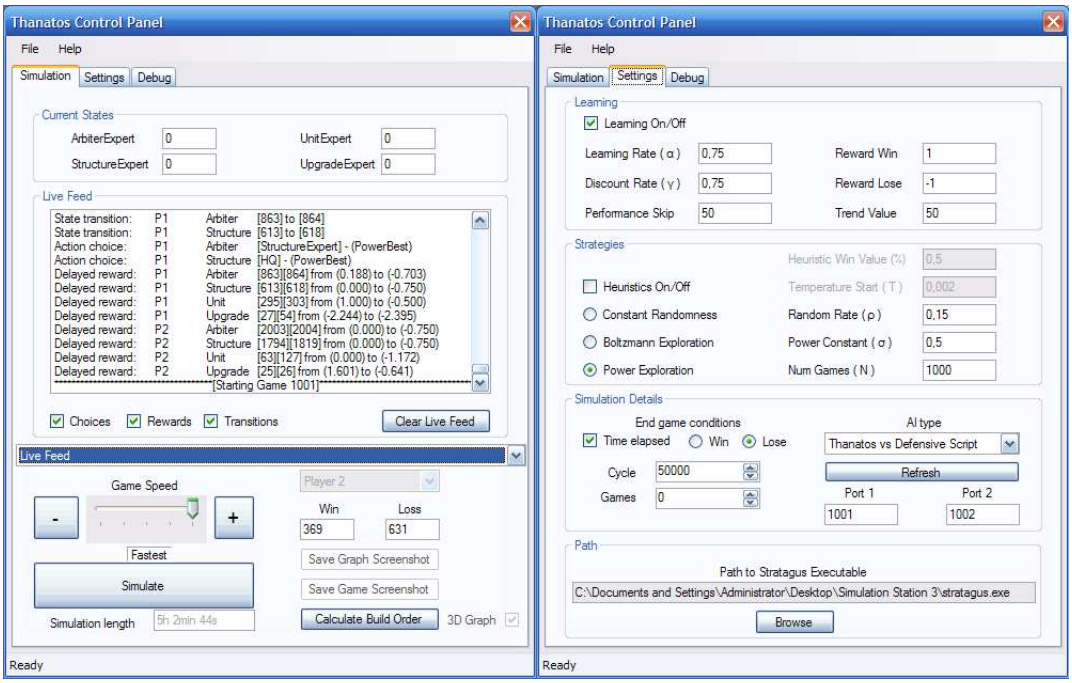

Figure B.1: Screenshot of the live feed functionality and the Settings tab in the GUI control panel

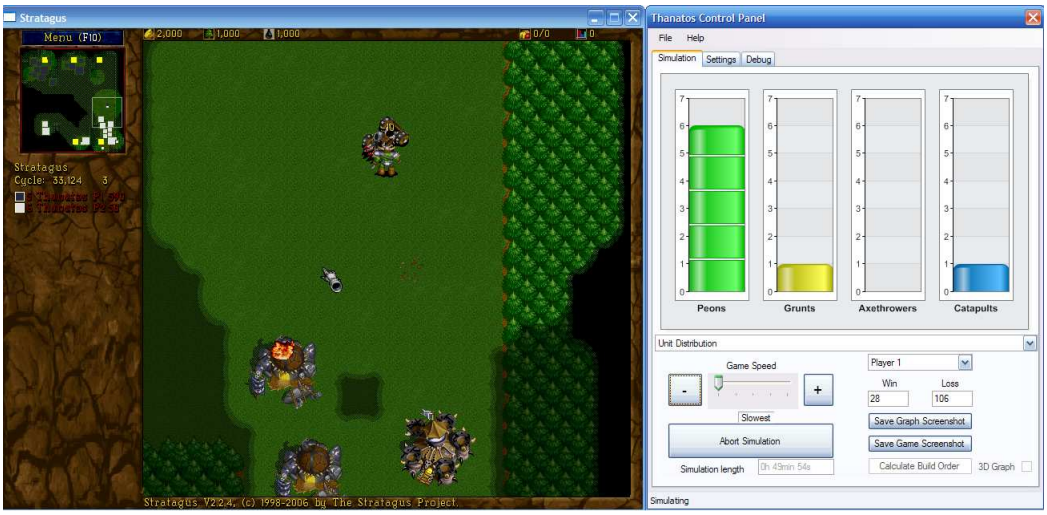

Figure B.2: Screenshot of the Unit distribution functioality while a simulation is running

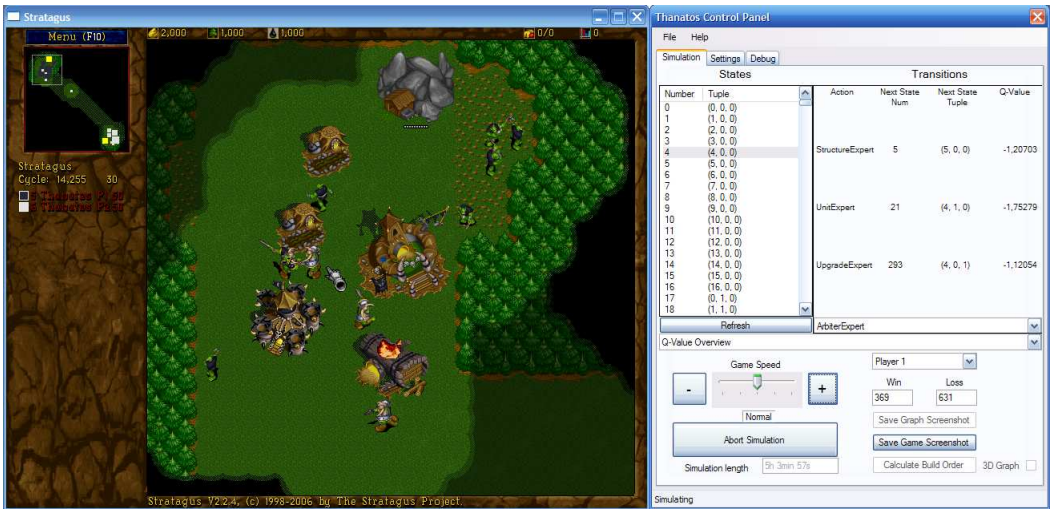

Figure B.3: Screenshot of the Q-value overview while a simulation is running

Appendix C

Static Build Orders

The following three static AI build orders were developed and tested against Thanatos. Be aware that the syntax differs between the illustrated build orders and the actual lua-code, as they are less presentable in their original fashion.

Two new commands are introduced. DefineForce(x) = unit and AttackForce(x) where DefineForce trains the specified unit and assigns it to force x, and AttackForce signals that the specified force should launch an attack at its opponent.

Listing C.1: Passive AI build order - passive_slow.lua

```
1  Build    unit-great-hall
2  Build    unit-pig-farm
3  Train    unit-peon
4  Train    unit-peon
5  Build    unit-troll-lumber-mill
6  Build    unit-pig-farm
7  Build    unit-orc-barracks
8  Train    unit-peon
9  Upgrade  upgrade-throwing-axe1
10 Build    unit-orc-blacksmith
11 Upgrade  upgrade-battle-axe1
12 Build    unit-great-hall
13 Train    unit-peon
14 Train    unit-peon
15 Build    unit-pig-farm
```

Listing C.2: Offensive AI build order - offensive.lua

```
1  Build              unit-great-hall
2  Build              unit-pig-farm
3  Train              unit-peon
4  Train              unit-peon
5  Build              unit-troll-lumber-mill
6  Build              unit-pig-farm
7  Build              unit-orc-barracks
8  Train              unit-peon
9  Build              unit-great-hall
10 Train              unit-peon
11 Train              unit-peon
12 Build              unit-pig-farm
13 Upgrade            upgrade-throwing-axe1
14 DefineForce(0) =   unit-grunt
15 AttackForce(0)
16 Build              unit-orc-blacksmith
17 Upgrade            upgrade-battle-axe1
18 DefineForce(1) =   unit-axethrower
19 AttackForce(1)
20 DefineForce(2) =   unit-catapult
21 AttackForce(2)
22 DefineForce(3) =   unit-grunt
23 AttackForce(3)
```

Listing C.3: Defensive AI build order - defensive.lua

```
1  Build              unit-great-hall
2  Build              unit-pig-farm
3  Train              unit-peon
4  Train              unit-peon
5  Build              unit-troll-lumber-mill
6  Build              unit-pig-farm
7  Build              unit-orc-barracks
8  Train              unit-peon
9  Build              unit-great-hall
10 Train              unit-peon
11 Train              unit-peon
12 Build              unit-pig-farm
13 Upgrade            upgrade-throwing-axe1
14 DefineForce(0) =   unit-grunt
15 Build              unit-orc-blacksmith
16 Upgrade            upgrade-battle-axe1
17 DefineForce(1) =   unit-axethrower
18 DefineForce(2) =   unit-catapult
19 DefineForce(3) =   unit-grunt
```

Appendix D

Example Code

The developed application source code can be found in the *src/Stratagus/src/masterAI* directory on the attached DVD. Here we present excerpts from some of the functionalities implemented, and a brief corresponding explanation is provided as well.

Listing D.1 shows the base class signature, used to derive all experts from. It defines the tools needed to save and delay actions, transition between states, and update the Q-table, as well as the actual Q-table itself.

An example of a class extending SuperExpert is shown in Listing D.2. This is the StructureExpert class, adding functions that are specific for this expert, like getRandomAction(), which returns a CUnitType object pointer. The upgrade expert on the other hand, would in this case return a CUpgrade object pointer instead.

```
1  template <class T>
   class SuperExpert {
3  private:
     T* currentState;
5    double currentReward;
     int delayedAction;
7    T *delayedState;
     T* savedState;
9    int savedAction;
   public:
11   SuperExpert(int rows, int cols, std::string filename);
     virtual ~SuperExpert();
13
     double **Q;
15   int nRows;
     int nCols;
17
     std::string filename;
19   void initQValues();
     void initQValues(std::string filename);
21   void saveQValues();

23   void setCurrentState(T *state) { this->currentState = state; }
     void setDelayedState(T *state) { this->delayedState = state; }
25   void setDelayedAction(int action) { this->delayedAction = action;
         }
```

```
    void setCurrentReward(double reward) { this->currentReward =
        reward; }
27  void setSavedAction(int action) { this->savedAction = action; }
    void setSavedState(T *state) { this->savedState = state; }
29
    T *getCurrentState() { return this->currentState; }
31  T *getDelayedState() { return this->delayedState; }
    T *getNextState(int action);
33  T *getSavedState() { return this->savedState; }
    int getDelayedAction() { return this->delayedAction; }
35  double getCurrentReward() { return this->currentReward; }
    bool isLastState() { return (nRows*nCols)-1 == currentState->
        getStateNumber(); }
37  int getSavedAction() { return this->savedAction; }

39  virtual std::string getExpertName() = 0;
    virtual int getExpertNumber() = 0;
41  virtual void resetCurrentState() = 0;
    virtual int getNumActions() = 0;
43
    void updateQ(double alpha, double gamma, double maxQ, double
        reward, T* state, int action, int playerNum);
45  double getMaxOfAllActions(T *nextState, bool &result);
    void transitionState(int action, int playerNum);
47  void delayAction(int action);
    void deleteDelayedReward();
49  void resetDelayedReward();
    void distributeDelayedReward(int playerNum);
51 };
```

Listing D.1: The SuperExpert class

```
1  #define MAX_STRUCTURE_ACTIONS (double)4
   #define NUM_STRUCTURE_TYPES 5
3
   class StructureExpert : public SuperExpert<StructureExpertState>
5  {
   private:
7    int actionsCounter[NUM_STRUCTURE_TYPES];
   public:
9    StructureExpert(std::string inputFile);
     ~StructureExpert();
11
     CUnitType *getRandomAction();
13   CUnitType *getBestBoltzmannAction(double temp);
     CUnitType *getBestAction();
15   CUnitType *getNextBuilding();

17   double getQValue(StructureExpertState *state, int action);

19   virtual std::string getExpertName();
     virtual int getExpertNumber();
21   virtual void resetCurrentState();
     virtual int getNumActions();
23
     bool isEnabledAction(int action);
25   void actionRequested(int action);
   };
```

Listing D.2: The StructureExpert class

As for Listing D.1, the code in Listing D.3 is used as a base class for expert specific state subclasses. It represents what is needed to define a specific state, except the action space, which is defined in each subclass.

Listing D.4 shows how the structure expert's states are defined. With five separate ints, the amount of actions performed can be easily be stored in the state object.

With the use of the SuperExpert and the SuperExpertState classes, it is trivial to add new actions, increase the supported state spaces or even add additional experts. However, the k_e values must be carefully re-selected if the AI model is expanded, as a single extra action may increase the required disk space tremendously.

```
class SuperExpertState {
private:
  int stateNumber;
public:
  virtual SuperExpertState *getNextState(int action) = 0;
  void setStateNumber(int number) { this->stateNumber = number; }
  int getStateNumber() { return this->stateNumber; }
};
```

Listing D.3: The SuperExpertState class

```
class StructureExpertState : public SuperExpertState
{
private:
  int nFarms;
  int nHQs;
  int nMills;
  int nRaxes;
  int nSmiths;
public:
  StructureExpertState(int farms, int hqs, int mills, int raxes,
      int smiths, int statenum);
  virtual ~StructureExpertState();
  virtual SuperExpertState *getNextState(int action);
};
```

Listing D.4: The StructureExpertState class

When a game is started, an object of the class CPlayer is created for each of the players. To be able to differentiate between a regular scripted AI and a Thanatos playing AI, we have added the MasterPlayerAi class to the file hierarchy and a pointer to such an object in the CPlayer class. This way, if the pointer is not set to NULL, we know that it is a Thanatos player, calling the decision making functions instead of running the predefined script.

The MasterPlayerAi class, as seen in Listing D.5, holds the expert object pointers, in addition to some helper functions. These are called whenever an entity has been completed, making a transition and the distributing of a reward easily accessible.

```
class MasterPlayerAi
{
public:
  MasterPlayerAi(CPlayer *player, int aiNum);
  virtual ~MasterPlayerAi();
```

```
7     CPlayer *player;
      int aiNum;
9
      ArbiterExpert *arbiterExpert;
11    StructureExpert *structureExpert;
      UnitExpert *unitExpert;
13    UpgradeExpert *upgradeExpert;

15    void buildingCompleted(CUnit *worker, CUnit *building);
      void newUnit(CUnit *unit);
17    void upgradeCompleted(CPlayer *player, const CUpgrade *upgrade);
  };
```

Listing D.5: The MasterPlayerAi class

Bibliography

[Ada02] Dan Adams. Warcraft iii: Reign of chaos. `http://pc.ign.com/objects/012/012906.html`, July 2002.

[Ada06] Dan Adams. The state of the rts. `http://pc.ign.com/articles/700/700747p1.html`, April 2006.

[Bae06] Christian Baekkelund. A brief comparison of machine learning methods. In *AI Game Programming Wisdom 3*, pages 617–631. Charles River Media, 2006.

[Bay08] Jessica D. Baylisse. Ai architectures for multiprocessor machines. In *AI Game Programming Wisdom 4*. Charles River Media, 2008.

[Bel07] Alex Bellos. Rise of the e-sports superstars. `http://news.bbc.co.uk/2/hi/programmes/click_online/6252524.stm`, June 2007.

[Ber05] Mariusz Bernacki. Principles of training multi-layer neural network using backpropagation. `http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html`, August 13 2005.

[Bro09] Jason Brownlee. Finite state machines (fsm). `http://ai-depot.com/FiniteStateMachines/FSM-Background.html`, August 13 2009.

[Buc02] Mat Buckland. *AI Techniques for game programming*. Stacy L. Hiquet, 2002.

[Buc05] Mat Buckland. *Programming Game AI by Example*. Wordware Publishing, Inc., 2005.

[Bur04] Michael Buro. Call for ai research in rts games. In *AAAI-04 Workshop on Challenges in Game AI*, pages 139–142, 2004.

[Bur09] Michael Buro. Orts - a free software rts game engine. `http://www.cs.ualberta.ca/~mburo/orts/`, 2009.

[Cav97] Cavedog. Total annihilation. `http://www.gamespot.com/pc/strategy/totalannihilation/index.html`, 1997.

[CF01] Brian Mac Namee Pádraig Cunningham Chris Fairclough, Michael Fagan. Research directions for ai in computer games. Technical report, Trinity College Dublin, Department of Computer Science, 2001. Department of Computer Science, Trinity College, Dublin 2, Ireland.

[Cha04] Alex J. Champandard. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. New Riders Games, 2004.

[Che08] Chessguru.net. Chessguru - chess rules. `http://www.chessguru.net/chess_rules/`, 2006-2008.

[EED] Yishay Mansour Eyal Even-Dar, Sham M. Kakade. Reinforcement learning in pomdps without resets.

[EF08] Samuel Erdtman and Johan Fylling. Pathfinding with hard constraints - mobile systems and real time strategy games combined. Master's thesis, Blekinge Institute of Technology, January 2008.

[Ent09a] Blizzard Entertainment. Blizzard entertainment. `http://www.blizzard.com/us/`, August 13 2009. ©2009 Blizzard Entertainment. All rights reserved.

[Ent09b] Blizzard Entertainment. Orc buildings. `http://classic.battle.net/war2/basic/obuildings.shtml`, October 2009.

[Ent09c] Blizzard Entertainment. Starcraft. `http://us.blizzard.com/en-us/games/sc/`, 2009.

[Ent09d] Blizzard Entertainment. Warcraft ii. `http://us.blizzard.com/en-us/games/legacy/`, 2009.

[fAI08] American Association for Artificial Intelligence. Decision trees. `http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/DecisionTrees`, September 03 2008.

[FH03] Dan Fu and Ryan Houlette. The ultimate guide to fsms in games. In *AI Game Programming Wisdom 2*. Charles River Media, 2003.

[Fle06] Lindsay Fleay. Boring theory. `http://www.rakrent.com/rtsc/html/boring.htm`, October 2006.

[Fle08] Lindsay Fleay. The full rts games list. `http://www.rakrent.com/rtsc/rtsc_gameslist2.htm`, December 2008.

[FSS07] Sander Bakkes Frederik Schadd and Pieter Spronck. Opponent modeling in real-time strategy games. In *GAMEON'2007*, pages 61–68, 2007.

[Fu03] Dan Fu. Constructing a decision tree based on past experience. In *AI Game Programming Wisdom 2*, pages 567–577. Charles River Media, 2003.

[Fun01] Glenn Fung. A comprehensive overview of basic clustering algorithms. Technical report, University of Winsconsin, Madison, WI, June 2001.

[Gera] Bruce Geryk. A history of real-time strategy games part i: 1989-1998. `http://www.gamespot.com/gamespot/features/all/real_time`.

[Gerb] Bruce Geryk. A history of real-time strategy games part ii: 1998-present. `http://www.gamespot.com/gamespot/features/all/realtime_pt2/index.html`.

[GH99] Terrence J. Sejnowski Geoffrey Hinton. *Unsupervised Learning: Foundations of Neural Computation*. MIT Press, 1999.

[IB02] Damian Isla and Bruce Blumberg. Blackboard architectures. In *AI Game Programming Wisdom*. Charles River Media, 2002.

[ICM] Inc. International Cyber Marketing. World cyber games. `http://www.wcg.com/6th/inside/wcgc/wcgc_structure.asp`.

[Inc08] Electronic Arts Inc. Sim city series. `http://simcitysocieties.ea.com/index.php`, 1989-2008.

[IS05] Marina Irodova and Robert H. Sloan. Reinforcement learning and function approximation. In *FLAIRS Conference 2005*, pages 455–460, 2005.

[Kot06] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Artificial Intelligence Review*, 26:159–190, 2006.

[Li08] Wenjie Li. Finding optimal rush attacks in real time strategy (rts) games. Master's thesis, University of Agder, 2008.

[Man02] John Manslow. Learning and adaptation. In *AI Game Programming Wisdom*. Charles River Media, 2002.

[McC00] Mason McCuskey. Fuzzy logic for video games. In *Game Programming Gems*. Charles River Media, 2000.

[Mic99] Microsoft. Age of empires 2. `http://www.microsoft.com/games/age2/`, 1999. © 1999 Microsoft Corporation. All rights reserved. Terms of Use.

[Mil06] Ian Millington. *Artificial Intelligence for games*. Morgan Kaufmann, 2006.

[Mil07] James Miles. Starcraft in korea. `http://blogs.warwick.ac.uk/jmiles/entry/starcraft_in_korea/`, June 2007.

[MP04] Pieter Spronck Marc Ponsen. Improving adaptive game ai with evolutionary learning. In *CGAIDE 2004*, pages 389–396, 2004.

[Num] Numenta. Htm technology overview. `http://www.numenta.com/about-numenta/numenta-technology.php`, 2008.

[Ork02] Jeff Orkin. Simple techniques for coordinated behavior. In *AI Game Programming Wisdom 2*. Charles River Media, 2002.

[Ric94] John A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 1994.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. Alan R. Apt, second edition, 2003.

[Sho06] Shawn Shoemaker. Rts citizen unit ai. In *AI Game Programming Wisdom 3*. Charles River Media, 2006.

[Smi01] Timothy Adam Smith. Neural networks in rts ai, October 2001. Bachelor thesis (Bachelor of Engineering Thesis) The University Of Queensland.

[Smi08] Edward Smith. Video game genres. http://www.intersites.co.uk/87004, 2008.

[Soc09] World RPS Society. Rock paper scissors. http://www.worldrps.com, 2009.

[Ste06] Sindre Berg Stene. Artificial intelligence techniques in real-time strategy games - achitecture and combat behavior. Master's thesis, Norwegian University of Science and Technology, September 2006.

[Tal09] TalkOrigins. Biology and evolutionary theory. http://www.talkorigins.org/origins/faqs-evolution.html, August 13 2009.

[Tea07a] Stratagus Team. Stratagus - a real time strategy engine. http://stratagus.sourceforge.net/, 2007. Stratagus is a free cross-platform real-time strategy gaming engine.

[Tea07b] The Wargus Team. Wargus. http://wargus.sourceforge.net, 2007. Wargus is a Warcraft2 Mod that allows you to play Warcraft2 with the Stratagus engine.

[Wal06] Anders Walther. Ai for real-time strategy games. Master's thesis, IT-University of Copenhagen, June 2006.

[Wal09] Mark H. Walker. Strategy gaming. http://archive.gamespy.com/articles/february02/strategy1/index.shtm, August 13 2009.

[Yan09] Swedish Yankspankers. Spring engine. http://springrts.com/, 2009.