A CBR/RL system for learning micromanagement in real-time strategy games

Martin Johansen Gunnerud

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Agnar Aamodt, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

RTS games pose many challenges from a computational reasoning point of view. Previous work in our group addressed the problem of improved micromanagement of units during combat, and a software environment within the game Warcraft III was developed for experiments. The system, based on case-based reasoning, proved successful and was able to improve its capability by storing new cases given by the human player.

In this master thesis the problem of automated learning during game playing shall be studied. A literature study shall be made in order to identify previous work done with automated learning of micromanagement in RTS games. A learning task will be identified, and the learning task will be implemented and tested in an RTS game environment.

Assignment given: 15. January 2009
Supervisor: Agnar Aamodt, IDI

Abstract

The gameplay of real-time strategy games can be divided into macromanagement and micromanagement. Several researchers have studied automated learning for macromanagement, using a case-based reasoning/reinforcement learning architecture to defeat both static and dynamic opponents. Unlike the previous research, we present the Unit Priority Artificial Intelligence (UPAI). UPAI is a case-based reasoning/reinforcement learning system for learning the micromanagement task of prioritizing which enemy units to attack in different game situations, through unsupervised learning from experience. We discuss different case representations, as well as the exploration vs exploitation aspect of reinforcement learning in UPAI. Our research demonstrates that UPAI can learn to improve its micromanagement decisions, by defeating both static and dynamic opponents in a micromanagement setting.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter is the introduction to the project, and contains three important sections; section 1.1 gives our motivation for choosing to study learning based on experience in RTS games, section 1.2 lists the goals of the project, and section 1.3 describes the overall structure of the report.

1.1 Motivation

The commercial video games available today usually advertise graphics and story as their main selling points. Nevertheless, AI is quickly becoming more important in video games. Considering the game F.E.A.R as an example. The game received excellent reviews, and the reviewers praised the smart behavior of the enemy soldiers [15]. However, the seemingly intelligent behavior of the soldiers was the result of their internal planning system, which was loosely based on the STRIPS planning system [13]. STRIPS was developed by Richard Fikes and Nils Nilsson in 1971, and successfully implemented in a game 34 years later. In the coming years, we believe that more and more commercial games will embrace AI techniques.

While playing against intelligent opponents utilizing planning techniques, these opponents should also be able to obtain new knowledge. In real-time strategy (RTS) games, the computer should be able to learn from past experiences, and predict the actions of the opposing player in order to provide a fun and challenging game for the human player. However, today's commercial RTS games suffer from bad AI [3]. The AI in these games usually has only one default strategy, which will be used independently of what the human player is doing. Stene [18] explains this poor behavior:

”In most RTS games the units have basic behavior, for example a unit

> that is fired upon will attempt to move close enough to the enemy unit to be able to attack it back. Human players learn over time how to control their different types of units in the most efficient way, but traditionally RTS game AI will rely on the basic behavior and not control units beyond telling them to where to attack and when to retreat."

This default, deterministic behavior makes it easy for human players to predict the actions taken by the AI, and defeat it. Previous work with learning in RTS games such as work by Aha [2], Buro [3] and Ram [14] have successfully found ways to improve the AI of RTS games.

However, most of the previous research on automated learning in RTS games have focused on the macromanagement aspect of RTS games [19]. Macromanagement include decisions such as building the right buildings and units, harvesting enough resources, and similar high-level actions. On the other hand, the micromanagement aspect of RTS games deal with how units should be moved in combat, which enemy units they should target, and when they should run away.

The motivation behind this project is the lack of research related to micromanagement learning from experience in RTS games. We believe that allowing the computer to learn and adapt to situations automatically could make RTS games more fun and challenging to play. Further, we wish to test whether a computer controlled AI player can learn to improve their micromanagement during gameplay by learning which order enemy units should be killed, based on the current state of the game.

1.2 Goals

The goal of this project was to study how learning from experience during game playing could improve the micromanagement of a computer controlled AI player. A study of previous research on this topic was conducted in order to get an understanding what work had already been done, and topics which had not been sufficiently researched yet. The result of previous research was used to design and implement a system in a chosen RTS game environment, which goal was learn the micromanagement task of choosing which enemy units to target in given situations in order to defeat an enemy player.

The system was implemented using the machine learning techniques case-based reasoning (CBR) and reinforcement learning (RL). CBR was used because it is the primary focus of our research group. RL was used because several previous works on learning in RTS games used either RL or a CBR/RL hybrid system for learning. Our system was based on some of the previous systems we studied, most noticeably CaT [2], described in section 4.3.1. Additionally, utility values were used to support the CBR/RL system. The goal of testing was that the system should improve itself while playing, and be able to defeat the AIs already implemented in the chosen game environment.

1.3 Structure of report

The report is structured to be read as a whole from beginning to end. Later chapters may refer to earlier chapters, hence it is not recommended to read the report in fragments from different chapters.

The report obviously starts with this introduction, followed by a brief introduction to CBR, RL and AI in RTS games in chapter 2. Next, in chapter 3, the choice of game environment is discussed. This is followed by an overview of previous research related to micromanagement in RTS games, and some CBR/RL approaches to learning in RTS games, which is discussed in chapter 4. The design, implementation and results of the implemented system are described in chapter 5, 6 and 7, before the discussion in chapter 8. Finally, the conclusion and further work sections are found in chapter 9.

Throughout this report several RTS game terms will be used, which may be unfamiliar to those who have not played an RTS game before. These terms are listed in appendix B table B.1, and the reader is recommended to read and understand these terms before reading further.

Chapter 2

Background

This chapter presents the background of the project. Section 2.1 describes a typical RTS game, and the state of AI in RTS games today. The machine learning techniques we used in the implementation of our project are explained in section 2.2.

2.1 Real-time strategy games

Games in the real-time strategy genre are very popular today. Many RTS games are released each year, and some catch on to become immensely popular. RTS games are war games, with two or more players controlling different types of buildings and units on a large map. Building new buildings allows production of different units and research of new technologies for improving these units, but buildings, units and technologies require resources. Resources are mined from specific map locations, hence controlling as many resource locations as possible is often vital. In order to win a game, a player needs to use his army to destroy all enemy buildings. Armies consist of different types of units which never get tired and never miss when attacking their targets in combat. Most players dislike having their buildings destroyed, and will use their own army to defend them from attacks. The key to winning an RTS game is finding a good balance between economy and warfare.

2.1.1 Real-time strategy vs turn-based strategy

Turn-based strategy games are classic games such as board games, like chess and backgammon. These games are somewhat predictable, and there exist a finite (though very large) number of possible game states which may be represented as a tree. In turn-based games, only a single player can affect the game at any given time, and turns are taken in rotation.

However, real-time strategy games are very different. In these games, all players simultaneously affect the state of the game, and the state of the game is measured in milliseconds, not in turns. Hence such strategy games are called real-time strategy games. Since multiple players can execute different actions at all times, it makes predicting game states of real-time strategy games almost impossible.

2.1.2 Micromanagement vs macromanagement

Macromanagement in RTS games refers to the economic strategy of the game. This includes deciding which buildings to construct, units to train, upgrades to research, and when to build expansions. In short, macromanagement can be defined as how to best use the available resources. In most strategy games, macromanagement is more important than micromanagement, due to low health points of units and few unit abilities.

On the other hand, micromanagement in RTS game terms are defined as small, detailed gameplay commands, most commonly commands such as moving units or using a unit's special abilities during combat. Micromanaging units in an RTS game are essentially the task of giving orders to units. The ultimate goal of micromanagement is to win by losing as few units as possible. In RTS games made before year 2000, most unit orders had to be manually addressed by the player. However, steps to reduce micromanagement have been taken from older real-time strategy games. Improved pathfinding for units and formation movement are now state-of-the-art in most RTS games.

In some RTS games, units can have many special abilities which could be quite a hassle to manually activate during combat. In Warcraft 3, almost every unit has an activated special ability, and several units have more than one. In order to reduce the micromanagement required by players, Warcraft 3 introduced autocasting; beneficial abilities which the player would want to use in every battle are used automatically by the computer [5]. The autocast system is implemented as series of rules, and an ability is used once all the conditions of the rule are true. Autocast reduces the micromanagement required by players, and gives units limited intelligence, in the sense that they can execute beneficial actions without being told to do so. We extended this concept in our system, by giving units more control over their own actions, without explicit orders from a player or an AI.

2.1.3 AI in RTS games today

AI in games has existed since the first single player games in the late 1970s, like Pong. However, video game AI is a broad term covering the behavior of computer players or agents, such as pathfinding or finite state machine behaviors. Topics covered by the academic definition of artificial intelligence are seldom used in games, but there is one game which has utilized academic AI with success. This game is discussed in chapter 4.1.

Even though the first RTS game was released 17 years ago [7], the AI of commercial RTS games today is poor. Buro [3] gives several reasons for this:

- RTS games feature imperfect information, as well as hundreds of objects, which may move and act at the same time, independent of each other. In games like chess, an AI has perfect information about the game world, and each move has global effect. Hence human players may be outsmarted by enumeration.
- Most RTS games are multiplayer games, hence developers prioritize entertaining human vs human games, not human vs machine games.
- RTS games are complex and closed source. Even though most of these come with a mapmaking tool, there is no AI tool available to the public.

The challenge with RTS game AI is the vast search space. RTS games feature hundreds of interacting objects, partially observable environments and fast-paced micro actions [3]. Even in the early phases of an RTS game round, the decision complexity can be estimated to $1.5 * 10^3$, compared to 30 for chess [2].

To overcome their weaknesses, RTS AIs often cheat. For example in the RTS game Warcraft 3, the computer can see the whole world map without fog of war. Sight is usually limited to how far units can see, hence being able to see the whole map is a huge advantage. This means the AI never has to scout ahead, and always know where the player base is. Warcraft 3 features three AI difficulty levels, and on the hardest one, the AI earns resources at double the normal speed.

2.2 Machine learning techniques

We used two machine learning techniques for implementing our system. These were case-based reasoning and reinforcement learning. Additionally, we used utility values to compute unit actions. All techniques are described below.

2.2.1 Case-Based Reasoning

Case-Based Reasoning (CBR) is a machine learning technique in which problems and their solutions are stored in a knowledge base as *cases*. These cases may be retrieved later should a similar problem arise, as the solution to this problem will already be in the knowledge base called the case base. Aamodt and Plaza [1] defines CBR as: ”To solve a new problem by remembering a previous similar situation and by reusing information and knowledge of that situation.”

CBR may be described as four processes: [1]

1. RETRIEVE the case or cases from the knowledge base that are most similar to the current problem.
2. REUSE the information from the retrieved cases to solve the current problem. If no exact match is found, the solution to the new problem must be adapted from one or more cases.
3. REVISE the proposed solution if it failed.
4. RETAIN the experience for solving this case in the knowledge base.

Even though CBR is a machine learning technique, it has three key properties which differentiate it from other techniques [11]. First, it is a lazy learning method, which means that generalization beyond the training data is delayed until the system receives a new query. This is opposed to eager learning, where the system tries to generalize the training data, before observing new queries. Second, new query instances are classified based on similar instances, while those which are very different are ignored. Third, instances are represented as symbolic descriptions, as opposed to real-valued points.

2.2.2 Reinforcement learning

Reinforcement learning (RL) is a machine learning technique where an agent learns how to select the optimal actions for achieving its goal. [11] The agent learns from delayed rewards when the agent reaches a good state (for example, winning a game of backgammon). Hence the agent must learn how to select actions from this indirect reward, which results in the greatest cumulative reward. The agent will select a sequence of actions, and over time learn to select the actions which give the highest cumulative reward.

In all RL algorithms, finding a good balance between *exploration* and *exploitation* is important. Exploration is the exploration of uncharted states, such as choosing actions where the reward for the action is unknown. By exploration,

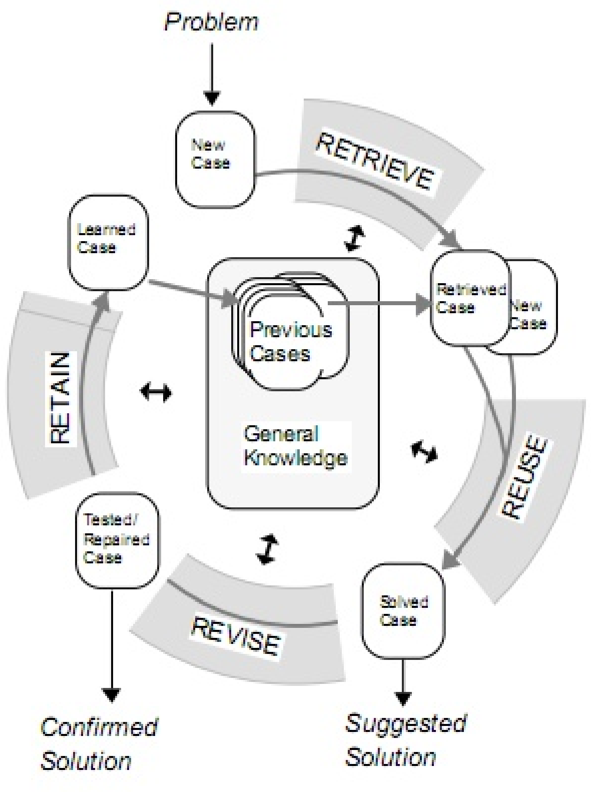


Figure 2.1: The CBR cycle [1]

the agent will learn new information, which may be used later. Exploitation is the exploitation of information the agent has already learned. It will select actions which will yield a high reward, to maximize cumulative reward. However, there might be better actions to take which are not know to the agent, because it has not explored enough. Finding a good balance between exploration and exploitation for a given RL problem is often difficult.

2.2.3 Utility value

Utility value is a method for measuring how good a state is [23]. The utility value is a numeric value, where higher value means better state. When faced with a choice, an agent calculates the utility values of each state that may be reached from the current state, and executes the action which will lead to the state with the highest utility value. This technique is not a machine learning technique, but it is used to select the best action from a state without going through several sets of rules. An advantage of using utility values is that the designer of an agent only needs to tell the agent what to do, not how to do it.

Chapter 3

Game environments

This section describes the process we used for choosing a game environment in order to implement our CBR/RL system. First, our requirements of an RTS game environment for implementing the system is listed, together with the game environments considered for the implementation. In section 3.3, the development possibilities of the game environment used in our project is discussed.

3.1 Requirements for game environment

In order to test a system which learns from experience in a micromanagement setting, the chosen game environment had to fulfill some requirements. These requirements are listed in table 3.1 below.

3.2 Choice of game environment

In order to implement and test the system described in section 1.2, a game environment was needed. Several open-source RTS games exist, and some of the most popular of these which had already been used in AI research [9], [18], [17], [14] were considered for our project. A game environment had to fulfill all the requirements of table 3.1 to be suitable for the implementation of our system. Below is a summary of these game environments, and which requirements they fulfill.

3.2.1 MadRTS

Though MadRTS had been used as a game environment for testing some of the CBR/RL hybrid systems described in section 4.3, we could not find any mention

Requirement ID	Description
1	The environment must support the creation of custom map scenarios, like a map which only has two players with a defined number of units, and no buildings.
2	The environment must be suitable for micromanagement. Units should have a fair amount of health points, and be able to take several hits before dying. It should also be easy to heal units, to encourage losing as few units as possible in battle.
3	The units in the environment must have the option to have activated, special abilities, such as targeted healing or unit summoning.
4	The environment must have the ability to read and write files when the AI is running, in order to read and update the knowledge base.
5	The AIs created for the environment should be written in a programming or scripting language which we already know, or may learn in a short time. This excludes C and C++.

Table 3.1: Requirements for the game environment

of it on the internet. Further research revealed that the creators of MadRTS, Mad Doc, was bought by Rockstar, and MadRTS was no longer available.

3.2.2 Wargus

Wargus is an open-source Warcraft 2 mod written in C++, which allows players to play Warcraft 2 with the Stratagus engine. Wargus support custom map creation, special abilities for units, file IO, and AI scripted in the scripting language LUA, which we should be able to learn for our project. However, there are some limitations with micromanagement in Wargus, most noticeably no support for micromanagement in the Wargus user interface, and low unit health points. Wargus fulfills all requirements except requirement 2.

3.2.3 ORTS

ORTS (Open Real-Time Strategy) is an open-source programming environment written in C, for studying real-time AI problems such as pathfinding, dealing with imperfect information, and planning in the domain of RTS games. [NEED SOURCE?] It is most commonly used to study low-level AI functions such as pathfinding, resource gathering and attack and defense formations. ORTS is a game engine, and new units may be defined though simple modifications of game files. AIs written in ORTS are written in C, and are generally complicated, spanning from 5 to 10 classes, each dealing with a specific behavior. ORTS support creation of custom maps, an environment suitable for micromanagement, and file IO. However, it does not support unit abilities. Hence ORTS fulfills requirements 1, 2 and 4.

3.2.4 Spring Total Annihilation

Spring TA is an open-source project written in C++, which is aiming to recreate the original Total Annihilation RTS game in a 3D environment. The Spring TA engine supports mods as well, hence it is possible to make a new game on top of the engine. It has a large and active development community, and quite a few AIs have been developed. AI bindings from C++ to java exist as well, but these are currently in an alpha version, hence being both buggy and unstable. Also, this environment supports micromanagement and file IO, but not special abilities for units. Spring TA fulfills all requirements, except requirement 2.

3.2.5 Bos Wars

Bos Wars is a futuristic RTS game written in C++. The implemented micromanagement AI is not very sophisticated, and the high level AI has only one strategy, namely rushing. However, Bos Wars support custom made AIs scripted in LUA, and file IO. It is not that suited for micromanagement, and units cannot have special abilities. Hence Bos Wars fulfills requirements 1, 4 and 5.

3.2.6 Warcraft 3

Warcraft 3 is a commercial game, hence it is not open source. It features a powerful editor, but the target audience for the editor is gamers, not programmers. Maps and AIs made through this editor cannot read data from external files. However, Warcraft 3 is an excellent game for micromanagement, and almost every unit has a special ability. Warcraft 3 fulfills all requirements except requirement 4.

3.2.7 Summary of game environments

A summary of the requirements fulfilled by the game environments may be found in table 3.2. None of the environments we considered proved to be suitable for our project. While most game environments fulfilled nearly every requirement, none were particularly suited for what we wanted to implement. As we could not use any of the open-source game environments, we decided to develop a new game environment.

Environment	Req 1	Req 2	Req 3	Req 4	Req 5
Wargus	X		X	X	X
ORTS	X	X		X	
Spring TA	X		X	X	X
Bos Wars	X			X	X
Warcraft 3	X	X	X		X

Table 3.2: Summary of RTS game environments

3.3 Development of game environment

The benefits of developing an environment from scratch was that it would fulfill all the requirements for game environments listed in table 3.1. Additionally, several features absent in other game environments related to micromanagement could be added, such as visible health bars for tracking health points of units. In order to quickly develop an environment, a game-development framework was needed.

3.3.1 Microsoft XNA

XNA (XNA's Not Acronymed) is a framework by Microsoft [21] for game development for Windows and Xbox 360. The framework is built on top of the .NET 2.0 framework, and provides game developers with a basic implementation of the game loop architecture [20] running at 60 frames per second at default. XNA supports both threaded game loops and loops with variable time steps. The framework includes methods to handle game logic updates and rendering, hence developers are given more time to focus on the gameplay, and not common game methods such as drawing graphics, which are solved over and over for all games. XNA also has a large community, and many tutorials and examples are available for free online.

3.3.2 ORTT

The environment we developed in the XNA framework was named ORTT, the name being inspired by ORTS. As the environment we developed did not feature buildings, research or unit production, the 'Strategy' term of RTS was replaced by 'Tactics', hence ORTT is an abbreviation of Open Real-Time Tactics. Despite similar names, ORTT and ORTS are not related other than that both are used to study learning in RTS games. ORTT was implemented as a single-threaded game loop shown in figure 3.1, running at the XNA default of 60 frames per second.

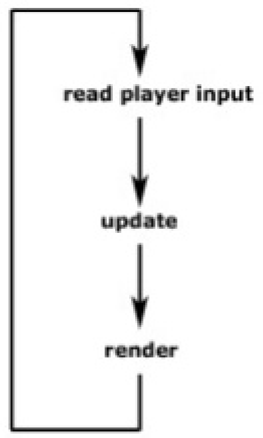


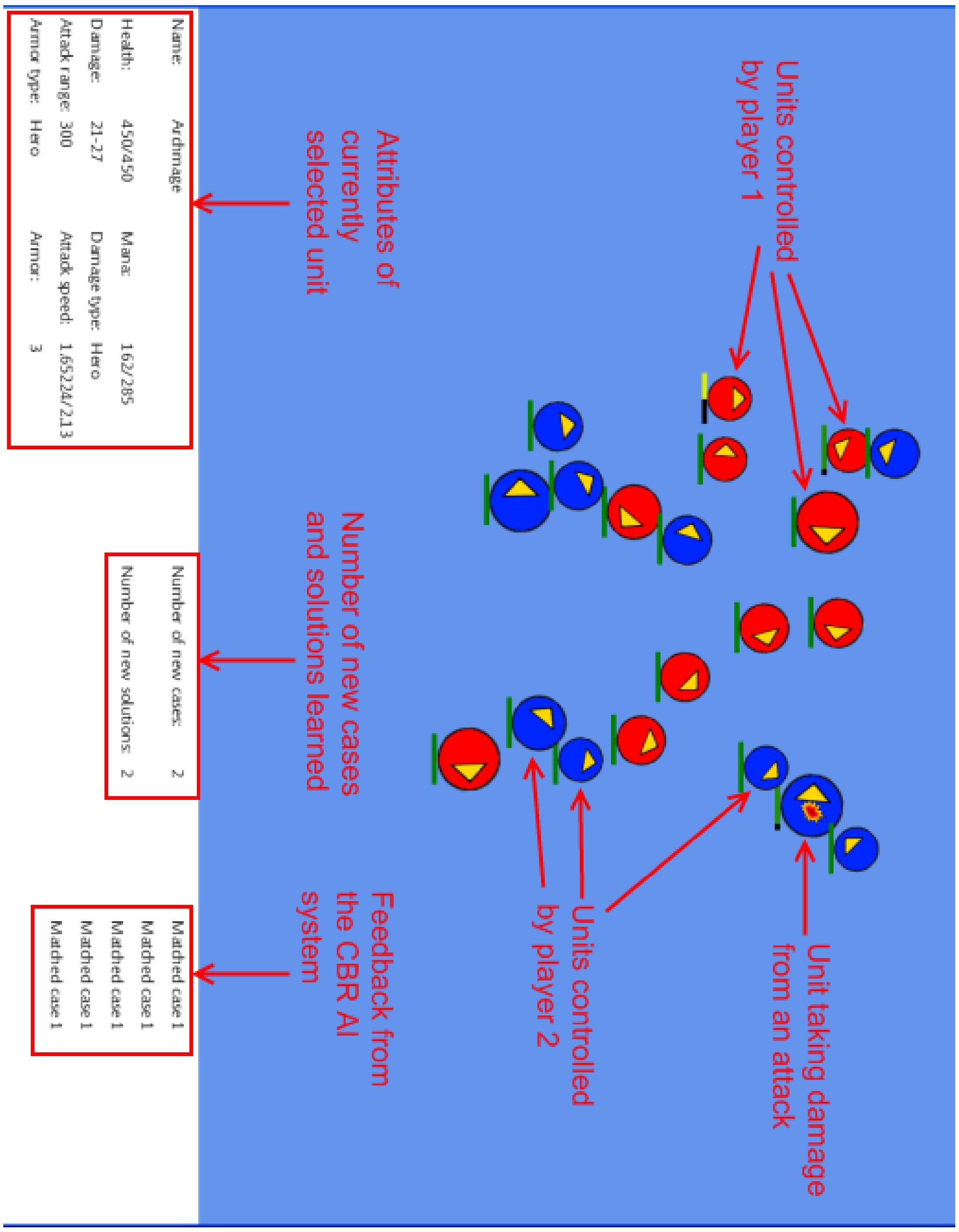
Figure 3.1: The game loop used in ORTT

The game world of ORTT contains two teams, with a number of different units, as shown in figure 3.2. Units have different attributes, inspired by the unit attributes of Warcraft 3. The attributes are such as size, move speed, damage, health points, attack speed and armor. Some units also have special abilities, such as a summon unit ability and a heal ability. The attack and armor types found in Warcraft 3 are found in ORTT as well. Each unit has one attack type and one armor type, and some weapons deal more damage to some armor types, and less to others. The full chart is found in table 3.3. This system is implemented to encourage units to attack specific enemy units for bonus damage. Units also have a numeric value named armor rating. Units take $0.06 * armorrating$ less damage from each attack.

	Light	Medium	Heavy	Fortified	Hero	Unarmored
Normal	1.00	1.50	1.00	.70	1.00	1.00
Pierce	2.00	.75	1.00	.35	.50	1.50
Siege	1.00	.50	1.00	1.50	.50	1.50
Magic	1.25	.75	2.00	.35	.50	1.00
Chaos	1.00	1.00	1.00	1.00	1.00	1.00
Spells	1.00	1.00	1.00	1.00	.70	1.00
Hero	1.00	1.00	1.00	.50	1.00	1.00

Table 3.3: The Warcraft III: The Frozen Throne: Attack Type vs. Armor Type Chart [4]

As in other RTS games, units in ORTT may not stand on top of each other. In order to avoid other units when moving from point to point in the game


Units controlled by player 1
Attributes of currently selected unit
Number of new cases and solutions learned
Feedback from the CBR AI system
Units controlled by player 2
Unit taking damage from an attack

Name: Archmage
Health: 450/450 Mana: 162/285
Damage: 21-27 Damage type: Hero
Attack range: 300 Attack speed: 1.65224/2.13
Armor type: Hero Armor: 3

Number of new cases: 2
Number of new solutions: 2

Matched case 1
Matched case 1
Matched case 1
Matched case 1
Matched case 1

Figure 3.2: A screenshot of ORTT, with descriptions

world, each unit has to calculate a path before moving. This is done by an implementation of the A* algorithm written specifically for XNA by Ziggyware community member Semei [16]. A unit calculates a new path to its target every 400 milliseconds, unless it was already at its target position.

The basic actions of all RTS units are orders. Without an order, units will stand idle until they spot an enemy. In ORTT, there are four different orders. These are attack, move, use ability, and use ability on target. Use ability is for abilities which does not require a target, like the summon ability, while use ability on target is the order for abilities like heal. When given an order, a unit will immediately do its best to fulfill the order. If the unit is given an attack order and is too far away from the target, it will calculate a path to the target, move, and attack when it is close enough to attack.

Orders in ORTT are given by the current AI controlling a player. Only one AI may control a player at any given time, but the current AI controlling a player may be changed every second, by specifying which AI to use in the update method of the game loop. This may enhance learning by playing against opponents which are changing their strategies mid-game, which would not happen in commercial RTS games. In ORTT, the current player AI will update unit orders every second. The AI procedure of ORTT is referred to as the AI loop in later chapters of this report.

Chapter 4

Learning in RTS games

This chapter presents the current state of learning from experience during play in RTS games. Section 4.1 describes the state of automated learning in commercial RTS games, while section 4.2 presents several projects related to automated learning in RTS games. Finally, section 4.3 mentions some examples of how learning in RTS games can be achieved using CBR/RL hybrid approaches.

4.1 Learning in commercial RTS games

Though we believed there to be more, we could only find one commercial RTS game using automated learning. For most RTS games, it is usually enough to implement competitive multiplayer to sell the game, hence AI in these games is often not a priority [3]. However, the game Black & White and the sequel Black & White 2 feature some impressive implementation of machine learning techniques.

4.1.1 Black & White

The real-time strategy game Black & White by Lionhead Studios was released in 2001, and was one of the first commercial games which utilized machine learning [22]. The game place the player in the role of a god, tasked with controlling the villagers of a certain tribe of Eden island, which is the fictional world the game takes place. The objective of the game is to make as many villagers as possible obey the player as their god. This can be achieved either by ruling through admiration, or ruling by fear.

There are two types of interesting agents in Black & White, which may be perceived as intelligent. The first of these are the villagers. Unlike most RTS games, these villagers cannot be directly controlled by the player, only moved to different locations. However, the villagers have limited intelligence, represented in desire tables and situation calculus. If left alone, they will do tasks they believe is best for their village. When the player wants a villager to do a specific task, all he has to do is move the villager to the location of the task. If the player wants a villager to chop wood, he can place the villager next to a tree, and the villager will deduct from its lookup table that the player wants him to chop wood. This behavior can be seen as intelligent, but the villagers are not learning during gameplay.

The second kind of agent in Black & White is the creature. The creature is a learning agent with a Beliefs-Desires-Intentions (BDI) architecture [6]. The player can only have one creature at any given time, and he has no direct control over it. Three different representations are used to represent the creature's beliefs and desires. Beliefs about individual objects are represented as symbolic attribute-value pairs, and an example of a creatures' representation is:

```
Strength of obstructions to walking:
    object.man.made.fence->1.0
    object.natural.body-of-water.shallow-river->0.5
    object.natural.rock->0.1
```

Beliefs about general types of objects are represented as decision trees. If the creature is hungry, it knows that it must eat in order to satisfy its hunger. However, the creature has not yet learned *what* will satisfy its hunger, so it will try to eat different objects. Example of objects may be rocks, fences or cows. The creature receives feedback based on how good the object tasted, and the creature updates its internal food decision tree based on this feedback. See table 4.1.

What he ate	Feedback
A big rock	-1.0
A small rock	-0.5
A tree	-0.2
A cow	+0.6

Table 4.1: Example of feedback for the creature in Black & White

Finally, neural networks are used to represent a creature's desire. After completing an action to satisfy a desire, the creature will evaluate how well the desire was satisfied, and update the weights of the neural network accordingly.

The creature learns in different ways. The first way is learning from reflecting on experience, as is described above. Second, the creature will learn from the orders it is issued by the player, believing that all orders issued by the player are beneficial. For example, the player may order the creature to attack a Celtic village. The creature will learn that villages of this kind should be attacked, and may do so on its own in the future. Third, it will observe the actions of the player, other creatures or villagers and try to mimic these. If the player picks up a rock and throws it at a Celtic village, the creature will get the impression that this is a good idea. Lastly, the player has the option to give direct feedback to the creature, by either stoking it, or slapping it after it has completed a task. Based on this feedback, the creature will learn which tasks are beneficial, and which are bad. Since the player can rule through admiration or fear, actions which are beneficial for a player ruling through fear may be devastating for a player ruling through admiration. The player ruling through fear may teach his creature to throw rocks at random houses of his own village, but this would not be feasible for the player ruling through admiration. The creature will maintain its own representation of the player, in order to track his playing personality.

4.2 Learning from experience in RTS games

Though automated learning is rarely used in commercial RTS games, it has been the focus of several research projects. This section presents four research projects related to learning in RTS games which we adopted elements from in order to use them in our system.

4.2.1 Adaptive reinforcement learning agents in RTS games

The goal of a recent project by Eric Kok [9] was to test whether a belief-desire-intention (BDI) agent could learn to outperform other scripted agents in the RTS game Bos Wars shown in figure 4.1. Kok used reinforcement learning to train his agent, and each agent had complete control over a player. The scripted agents either follow a single script, or switch between scripts in programmed patterns between learning episodes, where each game round is an episode. The agent has to learn to solve four problems in order to win an episode:

- What kind of units is needed to defeat the opponent.
- The requirements for training this army.
- How to counter attacks?
- How to exploit knowledge about the opponent for own benefit.

Ultimately, the BDI agent learned to adapt to different strategies, while trying to come up with new strategies at the same time. This way the agent is

Figure 4.1: A screenshot of Bos Wars

constantly improving, and by saving previous results, it is learning to avoid ineffective behavior. In the end, no single strategy proved dominant, as the game had a counter for everything. The AI of this project focused only on macromanagement learning, while micromanagement was handled by the default unit behavior already defined in Bos Wars. However, the idea of training the system by letting it play against opponents changing their strategies between learning episodes was used in our project as well.

4.2.2 Reinforcement learning in RTS Games

Nicolas Imrei [8] sought to investigate whether reinforcement learning could be used to develop a human-like computer player. In order to test his hypothesis, he implemented a scaled-down version of an RTS game, which was only a battlefield with units. Common RTS game elements such as economy management and base building was omitted, hence he was investigating micromanagement learning. However, the testing environment was severely limited; all units used in the simulations had the same attribute values, and they could only execute the actions 'move' and 'shoot' (no special abilities). The reinforcement learning units were tested in play against both human and computer players. Each unit was controlled locally, and not by a central AI. After running several simulations, the units developed the following behaviors:

- Shoot when seen unless health is low
- If health is low, move to a health spot
- Units form a health-spot queue
- Diversion of a centralized opponent's attention

Health spots are specific spots on the battlefield which will slowly regenerate the health of any unit standing on top of them. A screenshot of the testing environment is illustrated in figure 4.2. Here, the arrows of different colors are agents of different teams, and the pink spots are health spots. This project shows that advanced behaviors can emerge when units are given intelligence, and the idea was adopted for our project as well, as our units acted individually on orders received from our system which chose enemy units to target.

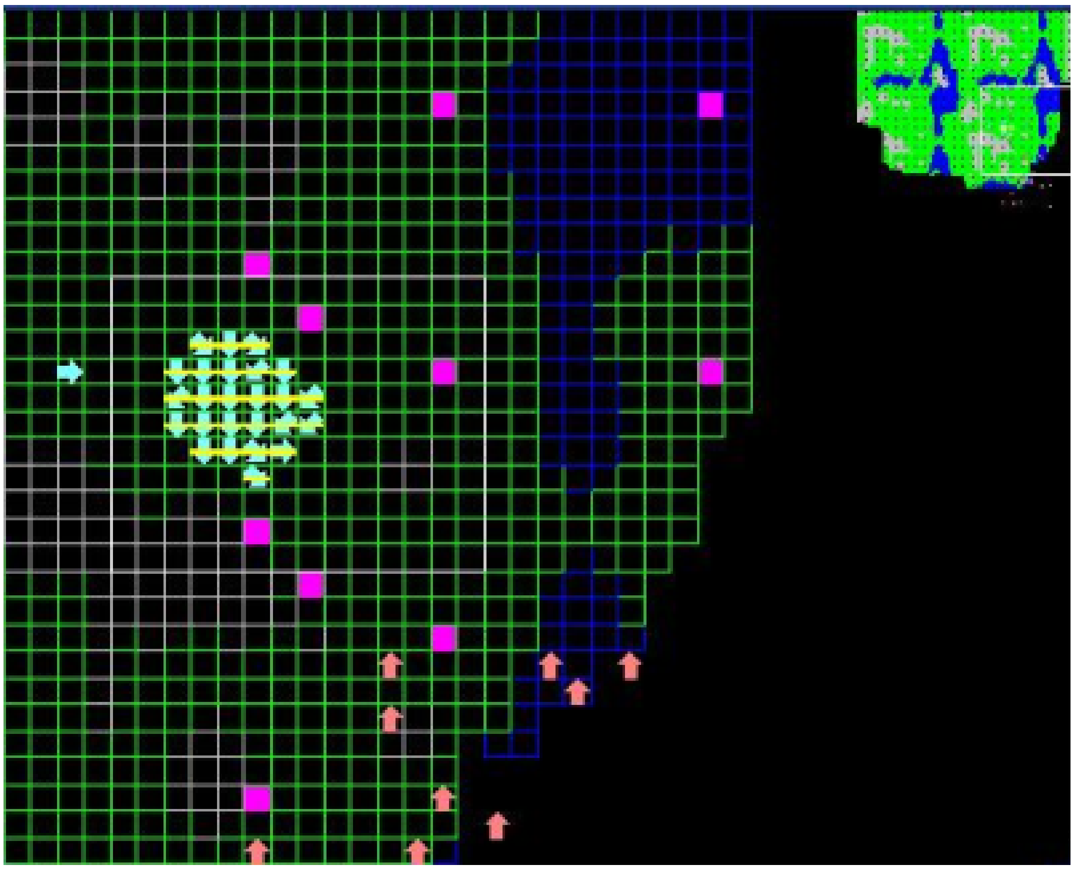
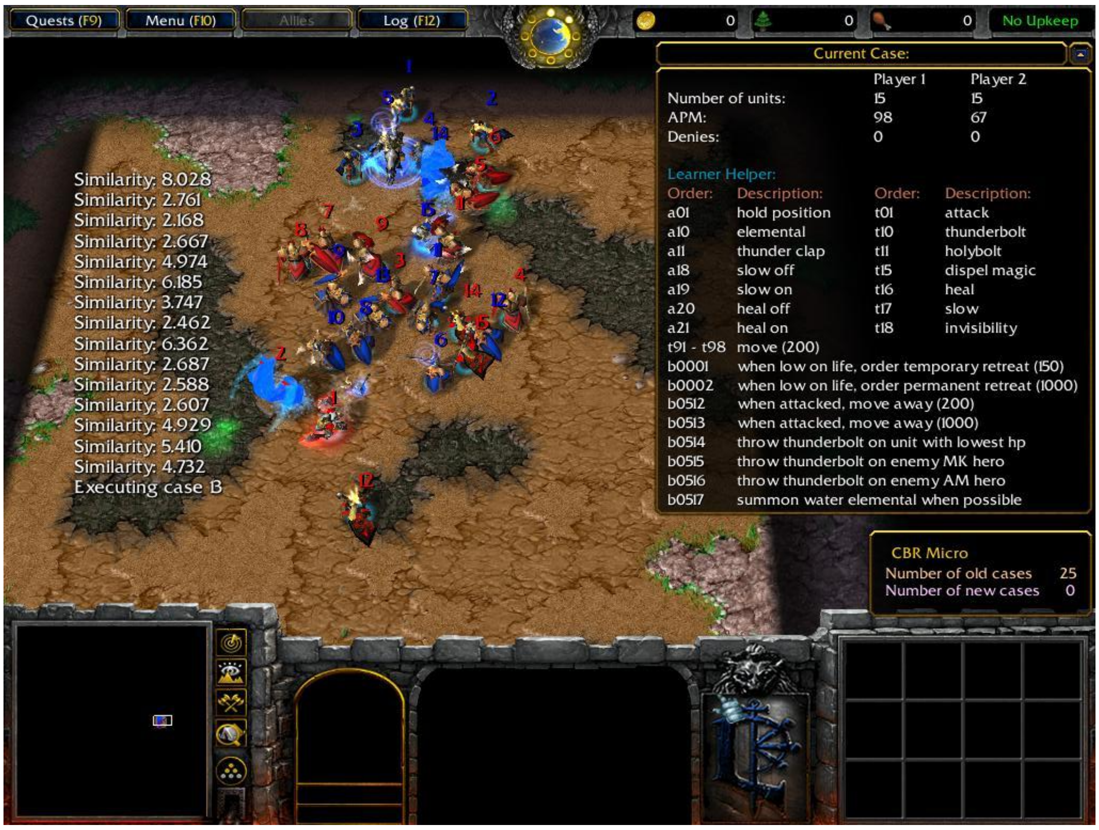


Figure 4.2: A screenshot of Imrei's project

4.2.3 CBR for improved micromanagement in RTS games

Tomasz Szczepanski [19] researched how CBR could be utilized to improve micromanagement in the RTS games. A CBR system for controlling a player was implemented in the game Warcraft 3, and tested against both human and computer opponents. Similar to the tests done by Imrei, RTS game elements such as economy management and base building were omitted. Situations observed during gameplay were matched to cases in the case base, and the solutions of matching cases were applied to the current situation of the game. Even though this project did not employ automatic learning, it was learning through supervision from an expert player during gameplay. While observing a simulation, the expert player would pause the simulation and add a new case if he noticed that the AI system was executing the wrong case. Figure 4.3 illustrates the graphical interface used by the expert player during training.



Similarity: 8.028
Similarity: 2.761
Similarity: 2.168
Similarity: 2.667
Similarity: 4.974
Similarity: 6.185
Similarity: 3.747
Similarity: 2.462
Similarity: 6.362
Similarity: 2.687
Similarity: 2.588
Similarity: 2.607
Similarity: 4.929
Similarity: 5.410
Similarity: 4.732
Executing case B

Current Case:

	Player 1	Player 2
Number of units:	15	15
APM:	98	67
Denies:	0	0

Learner Helper:

Order:	Description:	Order:	Description:
a01	hold position	t01	attack
a10	elemental	t10	thunderbolt
a11	thunder clap	t11	holybolt
a18	slow off	t15	dispel magic
a19	slow on	t16	heal
a20	heal off	t17	slow
a21	heal on	t18	invisibility
t91 - t98	move (200)		
b0001	when low on life, order temporary retreat (150)		
b0002	when low on life, order permanent retreat (1000)		
b0512	when attacked, move away (200)		
b0513	when attacked, move away (1000)		
b0514	throw thunderbolt on unit with lowest hp		
b0515	throw thunderbolt on enemy MK hero		
b0516	throw thunderbolt on enemy AM hero		
b0517	summon water elemental when possible		

CBR Micro

Number of old cases	25
Number of new cases	0

Figure 4.3: A screenshot of Warcraft 3

After training the CBR system, it was able to successfully outperform the hardest AI difficulty in Warcraft 3 in terms of micromanagement. Additionally, once the CBR system was trained, it could be used as support for novice Warcraft

3 players, by aiding them in battle against the computer AI. The implementation and results of this project were highly relevant both for motivation and for the implementation of our project. We decided to create the same units used in this project for our system, and use a similar unit setup for testing our system.

4.2.4 Intelligent combat behavior in RTS games

A system similar to the one tested by Imrei was developed by Sindre Berg Stene [18]. This project was implemented in the open source game Spring Total Annihilation, and the goal was to create an AI with behavior that gave the impression of human intelligence. The AI computer players implemented did not learn from experience through automated learning, as their behavior was pre-programmed. Even though the project dealt with all aspects of RTS games, only the micromanagement part was relevant for our project. This part was implemented as a 'military decision module', which controlled all unit actions.

The military decision module is responsible for moving friendly units, deciding which enemy units to attack, and how to move friendly units around during combat to minimize losses. Deciding which enemy units to attack is done by calculating which unit does the most damage per second, and then attacking this unit. The other micromanagement done in the project is moving units out of range from attacking units in order to minimize incoming fire. Individual units will analyze their position based on how many units that can hit them, and move as far away as possible while still being able to hit the desired target. The military decision module also uses the RTS game technique focus fire attacking described in table B.1. An example of how the AI micromanage a unit is illustrated in figure 4.4.

The behavior implemented in the military decision module was relevant for our project as well. We wanted to be able to order units in our system to focus fire attack enemy units for maximum damage, as this seemed to be effective in Spring TA. Targeting the enemy unit which did most damage per second was also a strategy we used for one of the opponent AIs we tested our system against.

4.3 Use of CBR/RL hybrid systems in RTS games

This section describes some previous research on automated learning in RTS games with CBR/RL hybrid approaches.

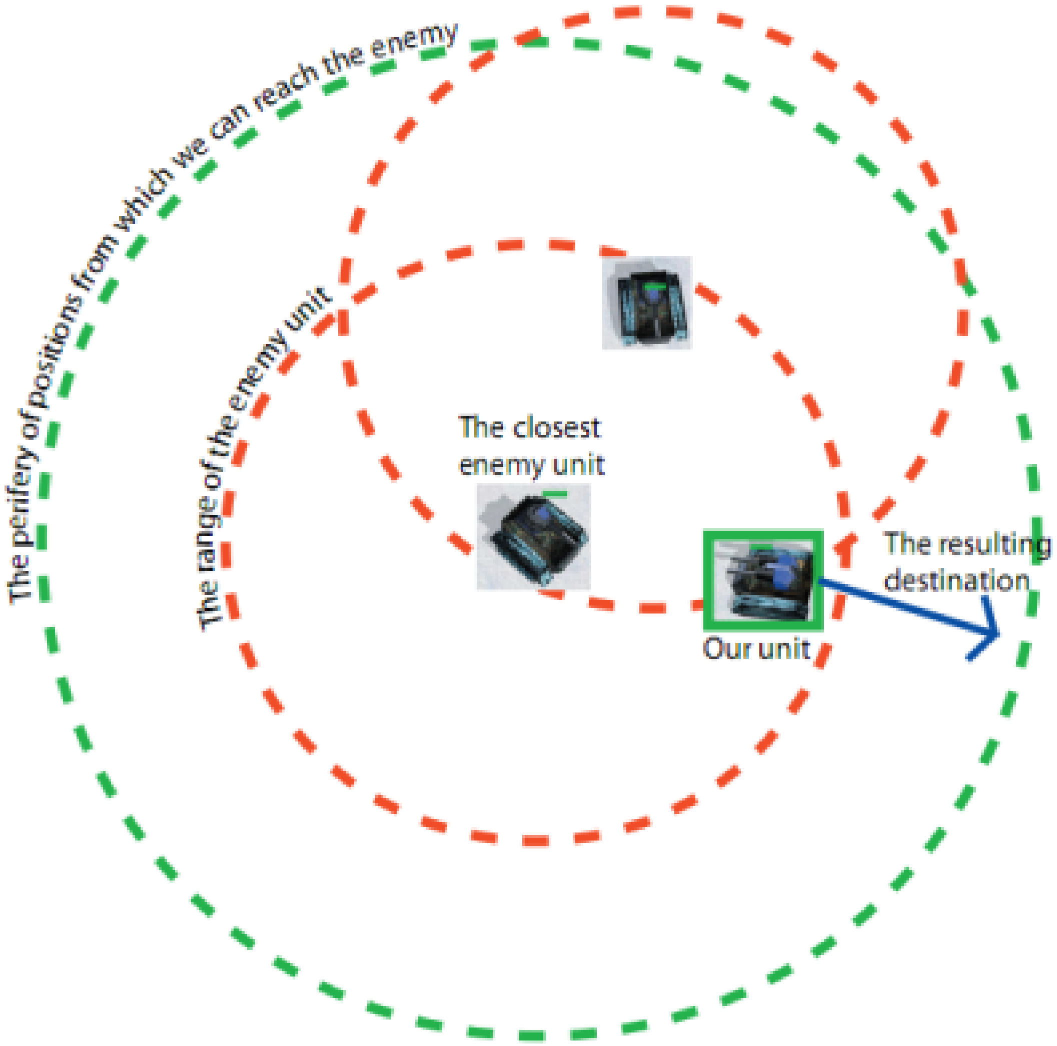

The perifery of positions from which we can reach the enemy
The range of the enemy unit
The closest enemy unit
The resulting destination
Our unit

Figure 4.4: An example of micromanagement in Spring Total Annihilation

4.3.1 Case-based Tactician

Case-based Tactician (CaT) [2] is a plan retrieval algorithm that combines three sources of domain knowledge to learn to defeat dynamic opponents in RTS games. The first of these is the state lattice, which defines game states by which buildings a player owns. Buildings built opens new possibilities for research and units, which lead to more available tactics. Hence each state of the state lattice has an associated number of tactics which may be executed in the state. These tactics are the second domain knowledge used in CaT. The third source is cases that map game states to tactics and their performance. Thus the goal of CaT is to learn which tactics to execute from different states, in order to win the game. Testing of CaT was done by letting it play against eight custom scripted opponents one at a time for 100 games. CaT outperformed these, and achieved an average win percentage of 82.4%.

CaT is similar to our system, except that it operates on the macromanagement level, while our system operates on the micromanagement level. First, CaT learns to win by selecting correct strategies for different cases, which is exactly what we wish to achieve. Where CaT uses buildings to differentiate states, we use units. Each state in CaT has specific buildings which enable different strategies. Our cases have different priorities, which enable friendly units to attack different enemy units. CaT keeps track of the performance in order to execute winning tactics later, and we do the same for our states. Where we use elements of RL to give rewards to cases, CaT updates the score of each case after each game, and selects the tactics with highest value during case retrieval. CaT even has an exploration parameter, which is also an element from RL we use in our system. We implemented numerous elements from CaT in our system.

4.3.2 Continuous Action State Space Learner

The Continuous Action State Space Learner (CASSL) [12] is an integrated CBR/RL algorithm developed for selecting actions in an RTS game environment. CASSL models the space of possible actions in an RTS game directly as a countinuous model, and not as CaT which use discrete states. This allows the algorithm to more quickly improve the accuracy of actions taken. Learning is achieved by using two case bases; the transition case base for modeling effects of applying actions, and the value case base for modeling the value of a state. Each case base supports the CBR cycle described in section 2.2.1. Cases are retained and revised as a series of gameplay episodes.

Selection of action states is done in a 5-step process. First, similar actions are retrieved from the case base by a k-nearest neighbor algorithm. Next, the algorithm predicts the next states from the current situation, and calculates the values of the predicted states. Finally, a value model of states is computed, and the action with the highest value is selected and executed. An illustration of the action selection process may be found in figure 4.5.

The task to be learned was to minimize the number of orders given to units in order to win a predefined scenario. After training, CASSL performed significantly better than other algorithms which selected actions from discrete states, hence improving performance of the algorithm. Where algorithms such as CaT needed to issue on average 80 orders, CASSL won with 12. Like our project, CASSL is a CBR/RL hybrid system. The goal of CASSL was to reduce the number of orders

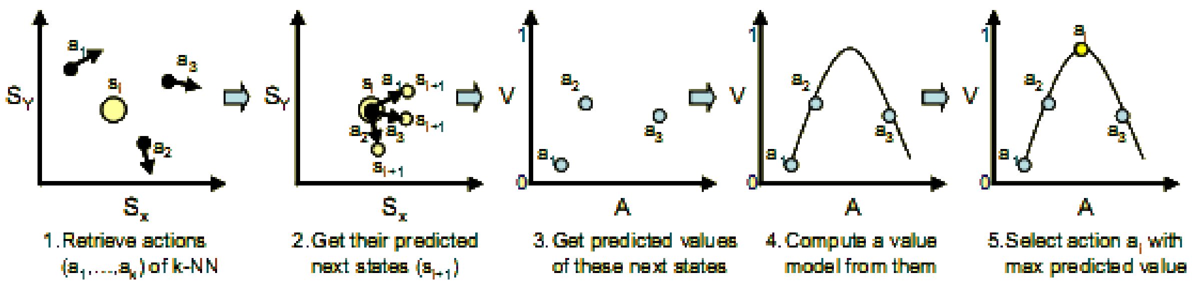

1. Retrieve actions (a_1,...,a_k) of k-NN
2. Get their predicted next states (s_{i+1})
3. Get predicted values of these next states
4. Compute a value model from them
5. Select action a_i with max predicted value
S_Y
S_x
V
A

Figure 4.5: The action selection process of CASSL

given, while in our project, the goal is to learn to improve micromanagement, no matter how many orders are given. We believed that our system would benefit more from an architecture like CaT, hence we did not use any elements from CASSL.

4.3.3 Case-Based Reinforcement Learner

The CAse-Based Reinforcement Learner (CARL) [17] is a CBR/RL hybrid system, which learns by the concept of transfer learning. Transfer learning is defined as [17]:

> Transfer learning is somewhat like generalization across tasks; that is, after seeing example instances from a function task, the transfer learning agent is able to show performance gains when learning from instances of a different task or function. For example, one might imagine that learning how to play checkers should allow one to play chess, a related but unseen game.

The implementation of CARL use transfer learning levels 0 - 4. A description of these levels may be found in table 4.2.

Upper levels of CARL reason about strategy, while the lower levels handle tactics. More important decisions are taken in the higher layers, and these are passed down to the lower levels as goals. The top level strategy of CARL is hard-coded. The layer below consists of a CBR/RL hybrid system, which makes tactical decisions such as attack, explore, retreat, or conquest new territory. Layers below are responsible for performing the above tasks in a predefined manner by using a planner. Like previous examples of CBR, the CBR system of CARL is implemented as the CBR cycle of section 2.2.1. Retrieval is done with a k-nearest neighbor algorithm, and revision use the RL algorithm Q-learning.

Using CARL reduces the time needed for learning new tasks after the system is trained. In some cases, the final performance of the system increases in complex scenarios. CARL has several similarities to our project. We use a two-layer

Transfer level	Description
0. Memorization	New problem instances are identical to those previously encountered during training. The new problems are solved more rapidly because learning has occurred.
1. Parameterization	New problem instances have identical constraints as Memorization, but have different parameter values chosen to ensure that the quantitative differences do not require qualitatively different solutions.
2. Extrapolating	New problem instances have identical constraints as Memorization, but have different parameter values that may cause qualitatively different solutions to arise.
3. Restructuring	New problem instances involve the same sets of components but in different configurations from those previously encountered during training.
4. Extending	New problem instances involve a greater number of components than those encountered during training, but are chosen from the same sets of components.

Table 4.2: Transfer learning levels 0 - 4 [17]

architecture, where the higher level handles the strategy, and the lower layer gives orders based on input from the high layer. Additionally, RL is used for case revision, like in our project. Like CARL we are also trying to achieve learning from experience. Where CARL does this with CBR and RL on the macromanagement level, we are also using CBR and RL, but on the micromanagement level.

Chapter 5

System design

This chapter describes the design and architecture of the system we implemented in our project. Section 5.1 describes decisions taken while designing the system, while section 5.2 describes the design of cases used in the CBR system. Finally, an illustration of the complete system and a description of how the system operates during gameplay may be found in section 5.3.

5.1 Design decisions

In RTS games, telling units *where* to attack is very different from telling them *who* to attack. In order to win battles, it is important to know which enemy units to attack first. Unit targeting is the task of selecting which enemy units to attack during a battle, and which friendly units to attack with. As described in section 3.3.2, Warcraft 3 uses an attack-defense table system of damage multipliers to encourage different mixes of units. A naive strategy would be to have every unit attack a target which would result in achieving the highest possible damage multiplier. This would involve a lot of unit movement, and possibly not targeting the most dangerous units first. For example, one of the units in Warcraft 3 has the ability to summon a special stationary unit, which will restore 12 percent of total health points to all friendly units within a large range every second. The summoned unit has 5 health points, which means it will die if it is attacked once. Because it gives the player who summons it a huge advantage, it should be destroyed as quickly as possible. Which enemy units to attack are based on several conditions, hence choosing enemy units to attack are situation-specific.

There exist many strategies for selecting which enemy units to kill first when in a battle. Some strategies are to just attack the closest unit, attack the weakest unit, or the unit which does the most damage, which was the measurement Stene [18] used in his project. In his project, the enemy unit which did most damage per second was attacked first. However, the optimal strategy may be a mixture of these, or something completely different. Choosing which enemy units to attack in a battle is a very important task which is used many times during an RTS game round. These target choices often determines the outcome of battles. While the strategy of focus fire attacking the enemy unit which does the most damage works in Spring TA, it will not work in an environment like Warcraft 3, refer to the example of the summoned healing unit in the previous paragraph. The summoned healing unit should be the highest priority, as it is the most important unit to kill in order to win the battle.

As seen from the previous paragraph, the task of choosing which units to attack is situation dependent in many RTS games, and it is difficult to program a rule-based AI to handle all situations. In our project, we wanted to solve this problem by implementing a priority system which could learn which units to attack given different situations. We believed that we could solve this problem with a CBR/RL hybrid system, where all battle situations encountered by our system in the game environment could be matched to a case. Once a battle was over, the system would learn through an RL system how good the attack choices used in the battle were from a delayed reward given by the environment. Our system would assign priorities to enemy units, where higher priority meant more important to victory. The units with highest priority were the units which should be killed first. Units controlled by our system were given orders to attack the enemy unit which was most important to victory, according to the solution of the best-matching case. This is similar to how humans learn to play RTS games. The first time an experienced RTS player tries a new game, he is likely to build some units and engage in battle. He will observe the battle, and learn how different units perform in different battle scenarios. Over time, he will know how to best use each unit, and which enemy units to attack first.

As the cases of our system were learned with automatic learning, no case revision from expert players could be done during play or after a case had been learned. Hence our system had to learn without supervision. In unsupervised learning, it is difficult to determine if a decision is *right*; when is a solution good, and how can a failed solution be improved? Since the solutions learned could not be revised by a human, we decided that each case stored in our system needed several solutions in order to keep track of past failed solutions. Revising one solution for each case might have ended up revising in a circle without the system knowing, hence each revised solution was stored as a separate solution. Each solution also had an attribute which kept track of how good the solution was. This attribute was updated after each battle by the RL system, based on the delayed reward from the environment after a battle was finished.

The evaluation and revision of the solutions was done like evaluation of tactics in CaT, by implementing a simple RL system. These elements were only basic evaluations of solutions. Like CaT, we also implemented an exploration parameter. Exploration rate of new solutions could be set by this parameter, defining when to exploit a solution, or explore in order to find a new one. The value of the exploration parameter was set by us prior to each test.

Our system was directly responsible for the actions of a computer-controlled player, as unit orders were given based on priorities. In order to simplify the implementation of the system, it was implemented as a two-layer system. The priority system with the CBR and RL systems was the high layer, and was responsible for determining the attack priorities of enemy units. The lower layer consisted of unit actions. Friendly units were controlled by this layer, as opposed to being directly controlled by the priority system. This distribution of control was done because we knew from work by Imrei [8] that units could act on their own, or with guidance from a higher entity (in this case, the priority system), once an overall strategy had been set, from work by Stene [18]. A second argument for this division was the results of tests run by Szczepanski [19]. Here, units were controlled by a CBR system, but each unit had a list of behaviors implemented as rules. This was done to reduce the number of cases required, and allowed the units to react to sudden changes on the battlefield. The lower layer of our system acted like the behaviors. This layer was responsible for giving orders to individual units after the overall strategy had been chosen by the CBR system. These unit orders were found by utility values. For each game state our system encountered, each unit calculated the actions it could take based on input from the CBR system, and executed the action with the highest value. The actions of a unit were to attack an enemy, run away, or use a special ability.

5.2 Case structure

The case structure for the CBR system consisted of identifiers and solutions. The identifiers were simply data from when the case was recorded, namely attribute values of the units currently alive. As described in section 5.1, cases were implemented in such a way that they could have more than one solution each. A section on choices for case identifiers and solution structure may be found in section 6.2. The case structure is illustrated in figure 5.1. The case base containing all cases was stored as an XML file. XML was chosen because it was easy to implement an XML reader in ORTT, and because XML is also easy to read for humans.

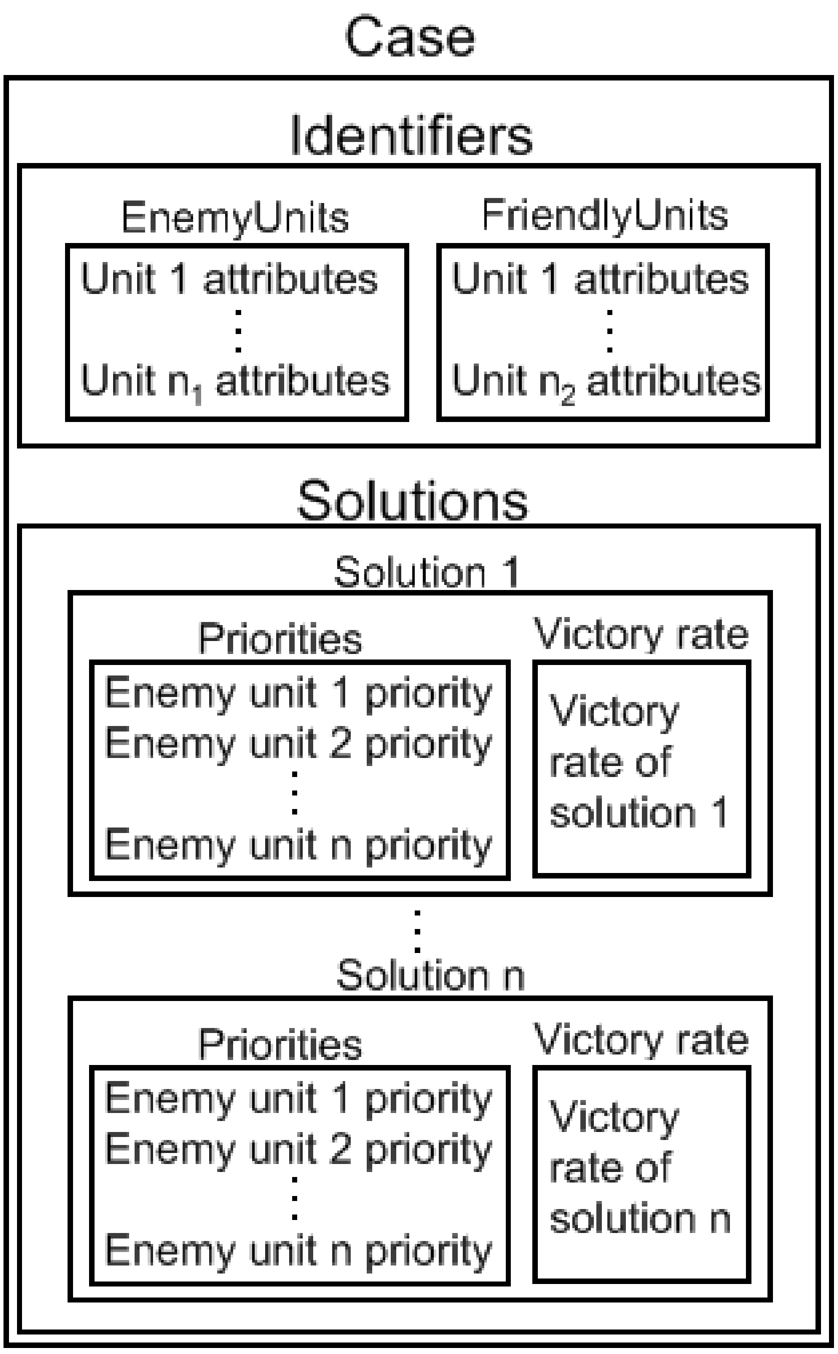


Figure 5.1: The case structure

5.3 System overview

We chose to name our entire system the Unit Priority Artificial Intelligence (UPAI). UPAI consists of the CBR system, the RL system and the utility value system. An overview of UPAI is illustrated in figure 5.2. A description of the flow of data is described below the figure. Diamonds are choices UPAI have to make, and rounded squares are processes.

The current game state is retrieved from the system every second, and matched against all cases in the case base. If the current game state cannot be exactly matched to any of the cases in the case database, a new case is created. Since each case also needs at least one solution, a new solution is adapted for the new case by combining solutions from the three cases with the highest similarity value to the new case. The priority values found in the solution are then sent to the utility value system.

If an exact match to the current game state is found, the best solution of the case is evaluated by the solution attribute keeping track of how good the solution is. If it is not good enough, a new solution will be created for the current case, and this new solution will be used immediately. If the solution attribute of a solution is larger than the exploration parameter, the solution is considered to be good, and the priority values of this solution will be used. Either way, the priority values of the best solution found are sent to the utility value system.

In the utility value system of UPAI, every action a friendly unit can take is calculated by a utility value. The action with the highest value is the best action, and a new order is created to execute the best action every second. The unit is given the order, and executes it in the environment. This procedure is done for all friendly units, hence they will all be assigned a new order every second, yet the new order may be identical to the old order.

After a battle is over, either the player controlled by UPAI or the opponent won by killing all enemy units on the battlefield. The RL system of UPAI updates the victory rate of all solutions used in the battle, and the solutions are stored in the case base. This step happens only once per battle, and the environment exits automatically after this step, in order to run a new test run. The creation of new solutions could have been moved to the RL system, but we chose to not create new solutions until the last moment when they were needed right away.

A full walkthough of a battle where a player is controlled by UPAI is found in appendix D. However, there are some spesific terms described in chapter 6 used in this walkthough. The reader is recommended to read chapter 6 before reading appendix D.

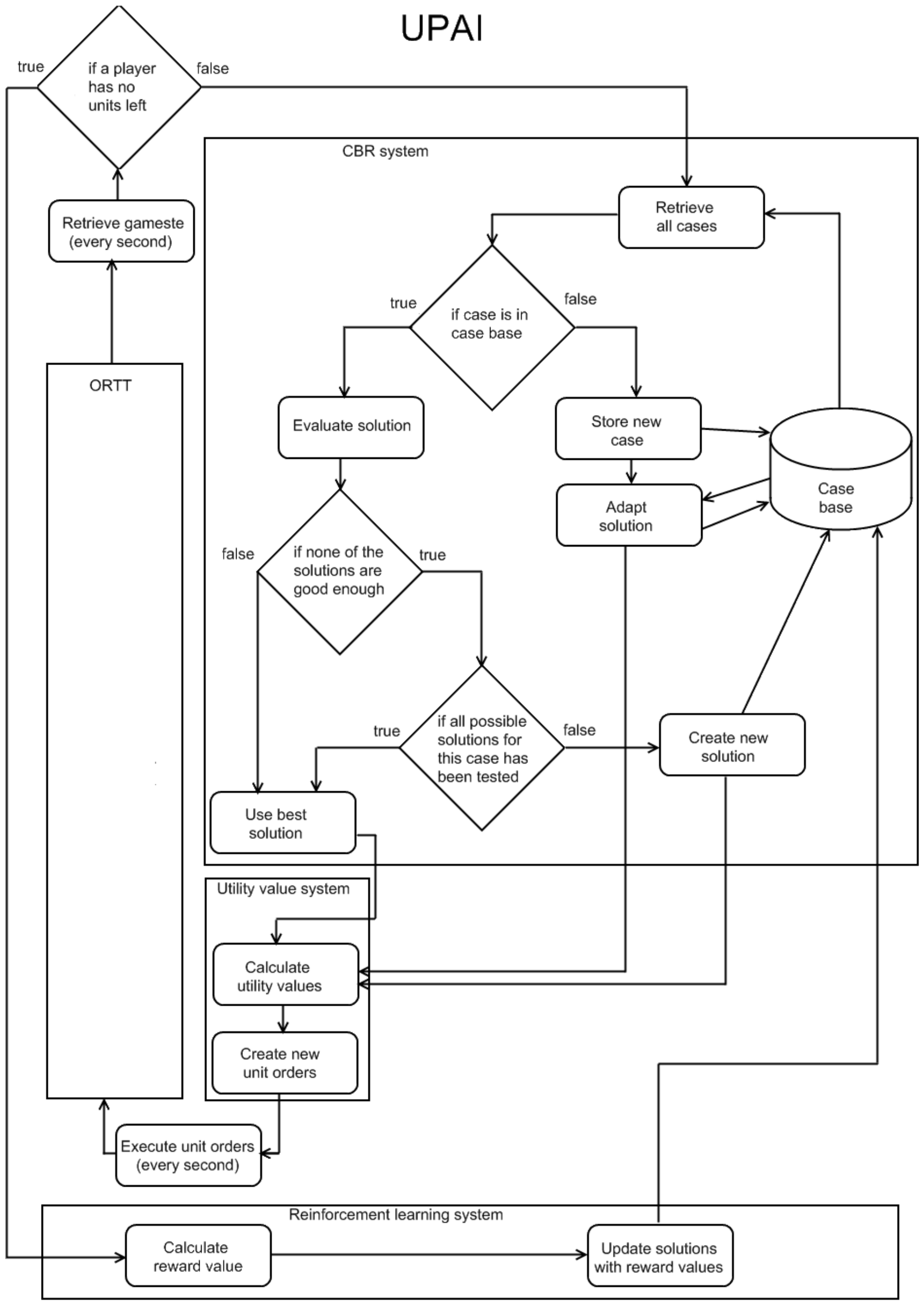


Figure 5.2: Overview of UPAI

Chapter 6

Implementation

This chapter describes how the CBR system was implemented. It is structured as follows: first, a description of the game state in section 6.1. This is followed by a description of the implementation of the case structure in section 6.2, with sections on case identifiers and case solutions. Further, section 6.3 explains the implementation of the CBR cycle. Lastly, the utility value system is explained in section 6.4, and the RL system in section 6.5. Section 6.6 closes the chapter with a short description of how other AIs were implemented in ORTT.

6.1 Game state

The game state of an RTS usually consists of a large amount of information. Each player has a number of resources, some sections of the map explored, and several units and buildings. ORTT simplifies the game state by removing irrelevant information such as the number of resources, buildings, and percentage of map explored. In ORTT, the entire map is visible to all players at all times.

However, the game state retrieved from ORTT at any given time still contains a lot of information. Each unit has 15 attributes which are visible to all players. This is information about the enemy an AI may use to describe the game state. The AI is not able to see other attributes of enemy units. If a case is represented by all 15 attributes for all units, in a battle between two players with 10 units each, the total number of stored attributes will be 300. Comparing two cases with 300 attributes would be computationally infeasible in a real time strategy game, as it would simply take too long. A case solution must be ready within milliseconds. A list of all visible unit attributes are found in appendix C.

6.2 Implementation of case architecture

As described in section 5.2, the case architecture consists of both identifiers and solutions. In this section, we will present our choices of identifiers and solutions, and present the final case structure used in UPAI.

6.2.1 The case matching problem

During the first implementation of the CBR system, we encountered a problem concerning case matching of numeric attribute values. Cases were first sorted by unit type, then sorted by attribute values for each unit. Matching units in cases were done unit by unit, trying to match same unit types first. The formula used to match units was a modified 1-nearest neighbor algorithm, similar to the one used by Szczepanski [19]:

$$Case\,difference(unit1, unit2) = \sqrt{\sum_{i=0}^{n}(\frac{p_i - q_i}{P_i})^2 \text{ if same unit type}}$$

$$Case\,difference(unit1, unit2) = \sqrt{\sum_{i=0}^{n} 1 \text{ otherwise}}$$

Here p_i is attribute value i of unit p, q_i is attribute value i of unit q, and P_i is the maximum distance between the attribute value (the value of the maximum attribute value of the unit with highest maximum of the attribute). All values are positive, as the distance cannot be less than zero. If a case consists of more units than another, the remaining units will be matched up to no unit, which is the same as a different unit.

The matching problem was identified when trying to write a good matching algorithm. Basically, matching units must be done in a smart way to find the actual match between cases. An illustration of this problem may be found in figure 6.1.

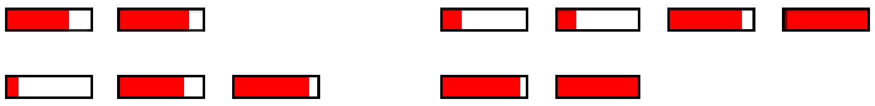


Figure 6.1: The matching problem, represented by unit health bars

On the left side of figure 6.1 is a typical matching problem, where similar units are to be matched by their health points. The length of color in the health bars determines the amount of health points left. The two cases to the left are discussed first. Here, the top case contains two of units, while the bottom one contains three. Units are sorted by health points to make the matching easier. However, matching by first occurrence would not be the optimal solution for these cases. The first unit of the bottom-left case should be matched to no unit for the best match, while the other units should be matched against each other. This problem may be fixed by searching for the best match, but this would take time. Consider the cases on the right side of figure 6.1. Here, the first two units of the top case should be matched to no units. However, the only way to discover this is to search through the other case once for each unit in the top case. In a case with 20 units on each team, this would take a long time. This problem is not just a problem with health points, but other numeric attribute values of units as well. We believe that this problem must have been encountered by other researchers of AI for RTS games, but we have not found any papers concerning this problem.

6.2.2 Case identifier implementation

Having the limitations described in section 6.1 in mind, the case identification attributes of the cases had to be simple. The most relevant unit attributes were unit type, health points, mana points and position. Sczcepanski [19] used these attributes as identifiers in his CBR system, yet encountered problems with the position attribute. He concluded that unit positions should be compared as patterns, not directly. To see why this is a problem, consider the two cases shown in image 6.2. The case is exactly the same, except that unit positions are turned 180 degrees. Instead of solving this problem by patterns, we moved the evaluation of unit positions to the utility value system. All units controlled by UPAI act independent of each other in the utility value system, and we took advantage of this. Instead of comparing whole patterns of both friendly and enemy units, each friendly unit would use the distance from itself to all enemies in the calculation of utility values. Distances from unit to unit could be easily calculated, hence we avoided the unit position problem by using distances in the calculation of utility values, and not as case identifiers.

The health point and mana point attributes are also important identifiers for game states. However, these attributes could also be moved to the lower layer, to simplify the case identification and avoid the case matching problem described in section 6.2.1. We wanted to implement a system which could learn to attack enemy units in the correct order, and only unit type was relevant for this task, not enemy position, health points and mana points. These three attributes could be used to differentiate units of the same type in the utility value system.

By moving the three attributes to the lower layer, the only case identifier attribute remaining was the unit type. Units of the same type have the same priority, as they are all alike when not matched by health points, mana points

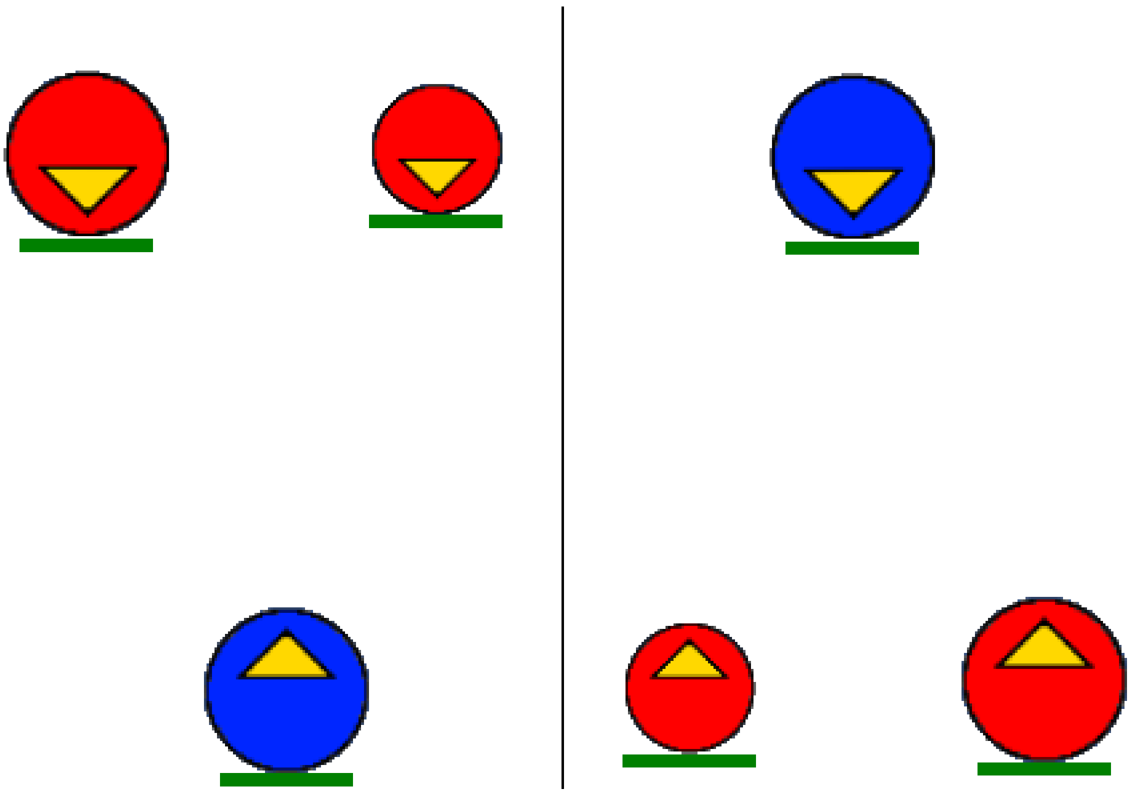


Figure 6.2: The unit position pattern problem

or position. Both enemy and friendly units are used for case identification, as attack priorities are dependent on both friendly and enemy units. For example, if a player only has flying units, he would not want to attack ground melee units, when there are other dangerous ranged units nearby.

6.2.3 Case solution implementation

The solution to a case consists of a list of unit names, and a list of priority values, and both lists are the same length. The indexes of both lists are the same as well, meaning that the unit with index 4 in the unit list has the priority value of index 4 in the priority value list. As described in section 5.1, each case needs several solutions due to case revision. The only input from the environment is whether UPAI wins or loses, and this information is what the CBR system has to learn solutions from. Two approaches were designed in order to find solutions which would work with automatic learning from experience during gameplay.

Solutions in the first approach consist of a single list with priority values between 0 and 100 for each enemy unit, where units of the same type have the same priority. How many units there are of the same type has no influence on the priorities. Additionally, there are two additional integer attributes which keep track of how many times the execution of the solution lead to victory or defeat. If UPAI win a battle, the victory counter of all cases used from the beginning of the battle to the victory is increased by 1. Should the system lose, the defeat counter of all cases is increased by 1. However, using this solution approach also caused some problems with the creation of new cases, which are described in section 6.3.3. The structure of a solution of this approach is illustrated in figure 6.3.

```
<Solution Number="0">
  <WinCount>3</WinCount>
  <LoseCount>1</LoseCount>
  <PriorityUnits>
    <Priority>100</Priority>
    <Priority>40</Priority>
    <Priority>20</Priority>
    <Priority>20</Priority>
    <Priority>20</Priority>
    <Priority>20</Priority>
    <Priority>45</Priority>
    <Priority>45</Priority>
    <Priority>45</Priority>
  </PriorityUnits>
</Solution>
```

Figure 6.3: Structure of first solution approach

In the second approach, we constructed a simpler priority value list. This list still hold priority values for all units, but the priority values are limited to 0 and 1. Units of the same type have the same priority, and only one type of unit can have priority 1 at any time. A value of 1 means that the unit should be attacked, while a value of 0 mean ignore. The idea behind this decision is that focus firing on an enemy unit is a smarter choice than attacking several targets. This strategy is true for almost every RTS game. When facing a group of the same units, where all units have a priority value of 1, the utility value system will favor units with low health points. If all priority units of the same type have the same amount of health points, the first unit which is attacked will loose health points, causing every other friendly unit to attack this damaged enemy unit as well.

The other solution attributes were also changed slightly in the second approach. The victory and defeat counters of the first approach were replaced by a reward value attribute and a play count (number of times the case has been used) attribute. The reward attribute held the total reward the solution had received from all tests, and the times used attribute held the number of times the case had been used. These attributes are used to measure how good a solution is, in the case retrieval process of the CBR cycle. A complete case with two solutions of the second approach is illustrated in figure 6.4.

```
<Case Number="0">
   <EnemyUnits>
     <EnemyUnitName>Archmage</EnemyUnitName>
     <EnemyUnitName>MountainKing</EnemyUnitName>
     <EnemyUnitName>Spellbreaker</EnemyUnitName>
     <EnemyUnitName>Spellbreaker</EnemyUnitName>
     <EnemyUnitName>Priest</EnemyUnitName>
     <EnemyUnitName>Priest</EnemyUnitName>
   </EnemyUnits>
   <FriendlyUnits>
     <FriendlyUnitName>Archmage</FriendlyUnitName>
     <FriendlyUnitName>MountainKing</FriendlyUnitName>
     <FriendlyUnitName>Spellbreaker</FriendlyUnitName>
     <FriendlyUnitName>Spellbreaker</FriendlyUnitName>
     <FriendlyUnitName>Priest</FriendlyUnitName>
     <FriendlyUnitName>WaterElemental</FriendlyUnitName>
   </FriendlyUnits>
   <Solution Number="0">
     <TotalReward>26</TotalReward>
     <PlayCount>2</PlayCount>
     <PriorityUnits>
       <Priority>1</Priority>
       <Priority>0</Priority>
       <Priority>0</Priority>
       <Priority>0</Priority>
       <Priority>0</Priority>
       <Priority>0</Priority>
     </PriorityUnits>
   </Solution>
   <Solution Number="1">
     <TotalReward>-12</TotalReward>
     <PlayCount>1</PlayCount>
     <PriorityUnits>
```

```
      <Priority>0</Priority>
      <Priority>0</Priority>
      <Priority>1</Priority>
      <Priority>1</Priority>
      <Priority>0</Priority>
      <Priority>0</Priority>
    </PriorityUnits>
  </Solution>
</Case>
```

Figure 6.4: Structure of a complete case

6.3 The case-based reasoning system

As described in section 3.3.2, the AIs of ORTT are only allowed to update unit orders every second. UPAI is no exception, and the current state of the battlefield is evaluated by UPAI every second. The full implementation details of the CBR cycle 2.2.1 is described in this section.

6.3.1 Case retrieval

During case retrieval, all cases from the case database are matched to the current case by the case identifiers, and the similarity value calculated by the following formula:

$$Casedifference(case1, case2) = \sum_{i=0}^{n} UnitDifference(case1Units, case2Units)$$

Hence the total case difference is the sum of the difference in both friendly and enemy units. A total case difference value of 0 is an identical match. The difference between units is weighted as follows:

$UnitDifference(unit1, unit2) = 0$ if same unit type

$UnitDifference(unit1, unit2) = 1$ otherwise

The similarity values of all cases are calculated because the similarity calculation was fast, and there are a limited number of cases in the case base. However, in a real game the number of cases in the case database could be huge. A better solution to case matching is to sort cases based on the number of units in each. A breadth-first search could be done to first calculate similarity values of cases with an equal number of units as the current case, and expand the search by one unit more and less if no satisfactory match is found, hence potentially reducing the amount of similarity values calculated. If an exact match is found, the current case would use the best solution from this case. If no exact match is found, some adaption of cases is necessary.

6.3.2 Case reuse

All cases are stored in the case base once they have been observed, hence no further adaption is needed on cases already stored in the case base. As a result, case adaption is done several times during the first test, and then more seldom as cases are added to the case base. Two case adaption algorithms were implemented; one for each of the solution approaches.

The adaption algorithm for the first solution approach creates a solution for the current case by combining the three cases closest to the current case, retrieved by the case matching algorithm described in section 6.3.1. The best solutions from each of these best matches are found by searching for the solutions with the highest win percentage. For each unit type in the current case, the priority value is set to the average value of the unit type from the best solutions. If a unit type is not present in any of the cases, the priority value was set to 10. The distance from the current case does not matter, as long as three cases are retrieved. If the three best matches has similarity values of 1, 9 and 11, they all contribute equally much to the new case solution, even though similarity values of 9 and 11 are far away from the current case.

The adaption algorithm for the second solution approach use a similar combination method to create a solution for the current case. The three cases with the highest similarity value (lowest total case difference value) compared to the current case are retrieved, and the solutions with the highest solution reward is found. Each solution contains exactly one unit type which has a priority value of 1. If two or more of the cases contain the same unit with priority value 1, then this unit type will also be the priority unit type of the solution for the new case. If all three cases have different priority unit types, the priority unit of the best matching case is chosen as the priority unit for the current case. Like in the first approach, the distance from the current case to the three best matches is not important.

6.3.3 Case revision

In order to achieve automatic learning during gameplay, the solutions of the cases had to be revised without interference from a human. To revise solutions, each solution needed some measurement of how 'good' it was. Additionally, a method for creating new solutions while utilizing the information of previous failed solutions was needed. Revision of cases happen only once per battle, once a player has won. Therefore, we need to revise all cases used in a battle based on the delayed reward received at the end of each battle. Case revision is done by the RL system, which is described in section 6.5. The first solution approach was discarded during the implementation of case revision, due to a problem with the creation of new case solutions. This problem is also described in section **??**.

6.3.4 Case retainment

Each new state encountered during a battle is stored in the CBR system as a new case. A solution for this new case is created by adapting the three nearest cases, and the case can be used for adaption of new cases immediately after it is learned. After a battle, the attributes of the solutions used are updated with new victory or defeat counters for the first solution approach, and with new reward values for the second approach.

No sorting or indexing is done for cases after they are learned. All cases are retrieved and matched to the current case during case retrieval. As described in section 6.3.1, sorting the cases by the number of units in each could possibly have decreased the time needed for case retrieval.

6.4 The utility value system

The utility value system was only implemented for the second solution approach, as the first solution approach had been discarded at this point. It is implemented as several lists, where each unit controlled by UPAI has its own list. A list contains the utility values of possible actions a unit can take at a given state in the game. These actions are attacking, using a special ability, or running away. The lists are of different length, depending on whether the unit has any special abilities. Utility value lists are calculated for all units controlled by the utility value system every time the AI loop is executed. After calculating all values, the action which results in the highest utility value is chosen. This action is used to create a new order for the unit, which is executed immediately.

6.4.1 Calculation of utility values

The basic action available to all units controlled by UPAI is to attack units with a priority value of 1. We constructed a formula to calculate the utility values of attacking enemy units:

$$Attack\ pri(unit\ u) = 10 * pri_u * (1 + \frac{H_u - h_u}{H_u} + (1 + \frac{M_u - m_u}{M_u} * 0.05)) - d(u, f) * 0.001$$

Where

$$d(u, f) = \sqrt{pos_u^2 + pos_f^2} - range_f$$

Where pri_u is the priority of unit type u received from the current solution, H_u is the maximum health points of unit type u, h_u is the current health points of the unit, M_u is the maximum mana points of unit type u, m_u is the current mana points of the unit, and $d(u, f)$ is the distance between enemy unit u and the friendly unit f for which the attack priority value is calculated, minus the range of the friendly unit f. If $d(u, f)$ is negative, it is set to 0. This is a measure of the movement effort the friendly has to do in order to attack unit u. A $d(u, f)$ value of 0 means that no extra movement effort is involved, and the enemy unit can be attacked without moving.

We constructed this formula because we wanted to differentiate units of the same type. Since only one type of unit can have a priority value of 1 at any time, a simple solution is to order units controlled by UPAI to attack the first enemy unit of this type in the enemy unit list. However, this unit can be far away, and have a lot of health points left. In our formula, we calculate the percentage of health points left, the percentage of mana points left, and the distance from attacker to target. Units with a lower amount health points left will result in a higher priority, which was what we wanted. This way, the utility value system favors units of the same type with low health points over units with high health points. We implemented the same system for mana points, but multiplied this by 0.05, as mana points are not nearly as important as health points. However, we implemented this formula backwards; we actually wanted the utility value system to favor units with higher mana points, but implemented a favorisation of low mana points. This was a bug we did not discover before after we had run tests. However, as the mana point percentage was multiplied by a factor of 0.05, compared to a factor of 1 for the health point percentage, this did not influence the priorities that much.

However, some units use a slightly different formula. The spellbreaker unit type has a special skill which does increased damage if the target has mana left, hence the mana calculation for spellbreakers is multiplied by 0.25 instead of 0.05. Note that if a priority is 0, the attack priority will be 0 or lower (depending on the distance between the friendly unit and enemy unit), and the enemy unit will be ignored.

Calculating the utility value of using a special ability follow other formulas, which are very different. For example, the very useful summoning ability will summon a new friendly unit to the battlefield, and this ability should be used as often as possible. If the conditions of using the summon ability are met, the utility value of using the summon ability is set to 1000. However, if the conditions cannot be met, the utility value of using this ability is set to -1000. Note that only specific units have access to the summon ability. If a unit has an ability, using the ability would have a higher value than attacking.

An example of a solution from the CBR system, current enemy unit attributes and a calculated priority list for an archmage unit who cannot use the summon ability and does not need to run away may be found in table 6.1. Here, the action with the highest utility value is attacking enemy spellbreaker 3. Note that the archmage listed in the table is the enemy archmage.

Case solution		Unit attributes					Utility
Name	Pri	H_u	h_u	M_u	m_u	$d(u,f)$	Utility
Archmage	0	450	450	285	167	23	-0.023
MountainKing	0	700	700	225	76	0	0
Spellbreaker	1	600	600	250	250	12	19.988
Spellbreaker	1	600	590	250	250	23	19.937
Spellbreaker	1	600	420	250	250	0	23.000
Priest	0	290	290	200	86	122	-0.122
WaterElemental	0	425	425	0	0	0	0
Run away	NA	NA	NA	NA	NA	NA	0
Summon ability	NA	NA	NA	NA	NA	NA	-1000

Table 6.1: Example of calculated utility values

6.4.2 Unit orders

After calculating utility values for all friendly units, the values are used to generate orders. If the action with highest utility value of a friendly unit is attacking an enemy unit, a new attack order telling the friendly unit to attack the enemy unit is generated. For example, the order given to the friendly archmage unit in table 6.1 will be a new attack order targeting spellbreaker 3. If the action with the highest utility value is using a special ability, a new ability order is generated for the unit.

6.5 The reinforcement learning system

The RL system handles the revision of case solutions, based on a delayed reward it receives after each battle. This reward is used to update all solutions used in a battle. When designing the RL system, we had not yet abandoned our first solution approach, hence we implemented both solutions in our RL system.

6.5.1 Reward values

In the first solution approach, we used following formula was used to determine how good a solution was:

$$Win\,percentage = \frac{victories}{victories + losses}$$

The delayed reward in this approach is simply whether our system has won or lost. If the total victory percentage falls below 20 percent, the solution is discarded, to be used again only when all other possible solutions have been tested, and none of these found satisfactory. If this happen, a new solution which is believed to be winnable is created based on all the previously discarded solutions. This proved to be harder than anticipated, as we could not find an algorithm for teaching the CBR system how to modify the priority values to create a better solution without human interference. With priority values between 0 and 100 for every unit type, the number of possible solutions is too high for trial-and-error. Since we could not implement an algorithm for creating new solutions without a large number of trial-and-error solutions, we discarded the first solution approach.

The measurement of how good a solution is in the first approach only hold information about the victory percentage, which mean how many times the solution has been used to win a battle. However, it does not give an information about how good the solution is in terms of surviving units. The primary goal of micromanagement in battles is of course to win a battle, but the second goal is to win by loosing as few units as possible. As the first solution approach could not measure this, we constructed a new formula for the second approach. This formula could also be used the first solution approach instead of the victory percentage, but we had already discarded it when implementing this formula. In this formula, the delayed reward is based on how many of the units controlled by UPAI survive, or a penalty based on how many of the enemy units survive, if

UPAI lose the battle. To measure how good a solution is in the second approach, the following formula is used:

$$Reward\,value = \frac{total\,reward}{times\,used}$$

In order to calculate the total reward value, the RL system has to measure the reward of a solution after a battle. We implemented this similar to how the performance score of tactics are updated in CaT. After a battle, the RL system will notice which units are still alive when the battle is over, and use this as a measure for how well UPAI did. Each unit has a certain 'worth' value specified by us, and the RL system simply looks up the values of surviving units, and calculates the sum of these values. If UPAI win a battle, the reward value is positive. If UPAI lose, the reward will be the same value, only negative. We know from experience with RTS games that the decisions taken early in a battle are more important than decision taken later. This also mean that solutions used early in the battle have the greatest effect on the outcome, and hence these receive a larger reward or penalty than the later solutions, which might not affect the outcome that much. The reward is distributed to the solutions by the RL system with the following formula:

$$Distributed\,reward = \frac{Calculated\,new\,reward}{solution\,number\,used + 1}$$

Solution number used is a number specifying which place a solution has in an ordered list of all used solutions. The first solution used has a solution number used equal to 0, the second solution used has solution number used equal to 1, and so on. This distributes the reward as described above, according to the graph in figure 6.5.

This distributed reward is added to the total reward of the solution after battles, hence the new reward value for each case is updated with the following formula:

$$New\,reward\,value = Old\,reward\,value + Distributed\,reward$$

If all solution values of a case are lower than the specified exploration parameter value described in section 5.1, the CBR system will try to create a new solution. This is done by simply looking at the previous solutions, and prioritize a unit type which has not yet been prioritized. If the RL system is unable to create a new solution, it will simply execute the best solution found. This approach is also a trial-and-error approach, but it is limited to the number of enemy unit types present on the battlefield. This number is far less than the values between 0 and 100 for every unit on the battlefield used in the first solution approach.

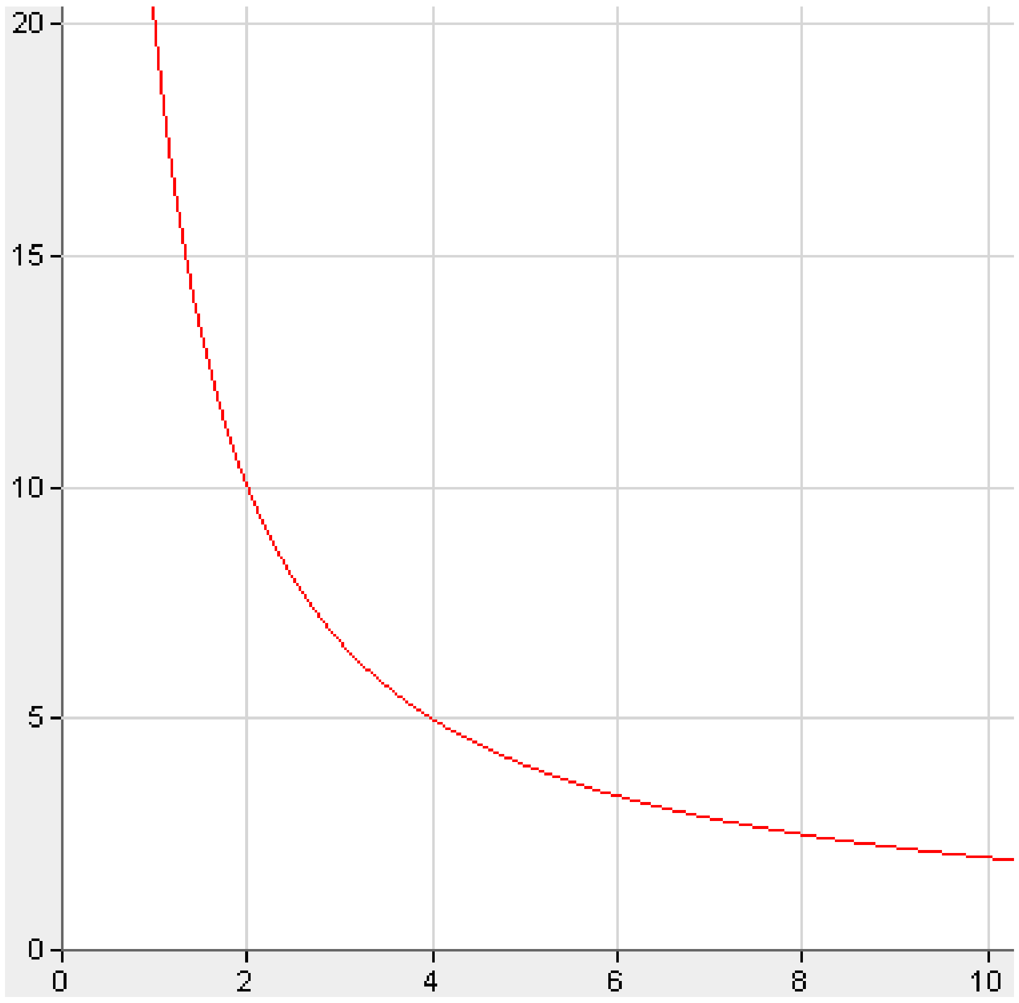


Figure 6.5: Reward distribution graph for a solution where distributed reward = 20

Using this approach for solution representation removes the problems from the first approach, by limiting the priorities so only one type of unit can be targeted at any time. By using a reward value, it is easier for our system to learn how *good* a case solution is. This can be read directly from the reward value, while in the first approach, a solution has to be used several times in order to have enough data to calculate an average win percentage. The win percentage for the first approach can change from 1 to 0.5 very fast if the solution has not been run many times, while we believed that the reward value in the second approach should remain more stable.

6.5.2 The exploration parameter

In order to be able to adjust the balance between exploration and exploitation, we implemented the exploration parameter e. This parameter is the measurement of how high a reward value of a solution had to be in order to be exploited. When the CBR system find an exact case match, it will check if the reward values of any of the case solutions are higher than e. If this is the case, the solution with the highest value is used. If none of the solutions has higher reward value than e, the CBR system will try to create a new solution.

Setting e to 0 means that any solution which will lead to victory will be used. However, this might not be the best solution available. If e is set higher than 0, the CBR system will be forced to test new solutions. How high to set e to explore is determined by the reward value, which is again determined by the unit rewards ('worth' of units). The unit rewards of our system are specified in table 7.3.

6.5.3 Graph data

In order to track the number of cases learned and the reward value of each test run, we write these values to a text file after each test run. After several test runs, the text file contains the data of the entire test. This data is interesting for us, since it allows us to see how the growth of the number of cases for each test run, and the reward values of each test run. By creating graphs of these, we can easily see if UPAI is improving at all, by looking at the reward value graph. If this graph is growing over test runs, UPAI is learning. We use the free data graph plotter LiveGraph [10] to plot our graphs, as LiveGraph can read text files and automatically create graphs from these. The graphs created by LiveGraph may be found in chapter 7.

6.6 Other AIs in ORTT

In order to test UPAI, it needed an opponent. Since support for letting a human player control a player in ORTT was not implemented, UPAI had to learn by battling a player controlled by another AI. These AIs are implemented as scripts, without the ability to learn. The AIs has the same functionality as the utility value system; giving orders. Every second, the units controlled by an AI other than UPAI give new orders to the units it controls. Hence ORTT cannot distinguish orders given by the utility value system and another AI. The AIs used for testing are described in section 7.1.2.

Chapter 7

Testing and results

This chapter describes the testing of UPAI. Section 7.1 explain our choice of testing environment, such as units and AIs used for testing. The results of all test runs are discussed and illustrated in section 7.2

7.1 Testing environment

Testing was done using two players, with an equal number of units in each army. The first player (red color) was controlled by UPAI, while the other player (blue color) was controlled by another scripted AI. A test was over when all units belonging to a player had been killed. Test runs were done in rapid succession; once a test run was over and the solutions used were updated in the case base, a new test began. This cycle continued for as long as we wanted to.

7.1.1 Units used in testing

The units used to test UPAI had the same attribute values as the ones used by Szczepanski [19]. These units were chosen based on studies of professional players done by Szczepanski, where he discovered that this particular unit setup was often used in human vs human games. In order to run quick tests of UPAI, the number of units used for testing was less than the number of units used by Szczepanski. An overview of the unit setup used during testing may be found in table 7.1.

This unit setup was also quite interesting due to the abilities of the units involved. The archmage has the ability to summon a new water elemental unit every 20 seconds, to aid him in battle. While the water elemental disappears when killed or after 60 seconds, summoning it does help killing the enemy units faster. The archmage has enough mana to use the summon ability twice, but

Unit name	Number used
Archmage	1
MountainKing	1
Spellbreaker	4
Priest	3

Table 7.1: The units used in testing

it can be used additional times if the archmage survive long enough to regain sufficient mana points. The mountain king has a storm bolt ability where he can throw a hammer at an enemy units every 9 seconds, instantly doing 100 damage. This ability can be used three times per battle. The priest units can heal a hurt friendly unit for 25 health points every second. This ability is usable 42 times per battle for a total healing of 1050 health points, and more times if the priests survive to regain mana points.

The really interesting part of this unit setup was the spellbreakers. These units have a special 'feedback' skill which allows them to drain the mana points of enemy units with each hit. The archmage and mountain king units lose 4 mana points when hit, while the priests lose 20. Enemy spellbreakers are immune to the effect. The feedback skill opened up some interesting strategy choices; should the spellbreakers attack the archmage first and drain him of mana points, making him unable to use the summon ability, or attack the priests to prevent them from healing other units?

The best way to prioritize in this scenario was to kill all the enemy priests first, before they could use all their mana to heal other units. This limited the maximum amount of health points the enemy could regain, which gave the friendly army the upper hand, since they had more total health points available. The second priority should be the enemy archmage hero. If his mana was not drained, he was able to summon a third water elemental to the battlefield. The water elemental has the second highest damage per second ratio of all units involved in the battle, which made it a powerful enemy. Further priorities were not as important as the first two.

7.1.2 AIs used in testing

We implemented a total of six different AIs which were used during testing to train UPAI. It was important to have several different training AIs, in order to avoid that UPAI learned to defeat a specific AI, but failed against other AIs. The AIs used, as well as their strategies are listed in table 7.2. We implemented these AIs because we wanted to test how UPAI did against several known RTS strategies. The strategies we wanted to test were: attacking the closest unit, attacking the unit which does the most damage per second, attacking units to maximize the damage multiplier, dancing, and a strategy which we believed was the best one. All AIs implemented can use special abilities, or else UPAI would have an unfair advantage. On the other hand, how the special abilities are used is different for each AI. In all AIs, the priests will heal the most damaged unit, and the archmage will summon a water elemental as soon as the conditions for doing so are met.

7.1.3 Rewards for the CBR system

Rewards are given to the solutions used, according to the distributed reward formula in section 6.3.3. To measure the reward, each unit used during testing is given a reward value where the values of surviving units are summed after a test is complete. The rewards for the units used in testing may be found in table 7.3. The water elemental gave a reward of 0, since it would disappear after 60 seconds. The maximum reward value UPAI could achieve in a test was 56, by killing all enemy units without losing any friendly units. Similarly, the maximum penalty value was -56, if UPAI lost all its units without killing any enemy units. 56 is the total sum of all unit values used in testing.

7.1.4 Exploration parameter

The exploration parameter e implemented in our system defines when a solution is good enough, or if the system should try to find a new solution. If e is set to 0, the CBR system will not try to create a new solution if there exists one for the current case with average reward value above 0. Similarly, if e is set to -56, the CBR system will never try new solutions, hence it will not learn. We tested our system with values of e equal to 0 and 56. In the latter case, no solution is considered good enough, and the CBR system will try a new solution every time it has the opportunity to do so.

7.2 Test results

The testing of UPAI was done systematically; first, the unit setup used for testing was tested in Warcraft 3, to see how the AI of Warcraft 3 would play. Then, UPAI was tested against all six implemented AIs with values of e equal to 0 and 56. The case base was set to the empty set before each test run. The performance of UPAI vs the other AIs was tested several times, until no more cases or solutions were learned. The result of each test is illustrated as two graphs; one for reward value and one for the total number of cases.

Name of AI	Strategy
SmartAI	This AI will focus fire attack and use storm bolt ability on the enemy priests first. It will then focus fire attack the remaining units in the following order, moving further down the list once all units of one type are dead: archmage, mountain king, spellbreaker.
DumbAI	This AI will move forward, and the units will attack the first enemy they spot. The mountain king will use the storm bolt ability on the first enemy he spots. Focus fire is not used by this AI. As this AI does not have any overall strategy, it is the most random of the implemented AIs.
FocusFireAI	This AI will focus fire attack the enemy unit which does most damage per second. Once this enemy is dead, the AI will focus fire attack the enemy with the second highest damage per second, and so on. The mountain king will use storm bolt on the enemy with highest damage per second. The priority list from highest to lowest damage per second is: mountain king, water elemental, archmage, spellbreaker and priest.
DamageOptimalAI	This AI will look up the attack and defense chart 3.3 and order each friendly unit to attack the enemy unit which will result in the highest damage multiplier for the friendly unit. The mountain king will use the storm bolt ability on the archmage. Focus fire is not used by this AI.
UtilityAI	This AI will use the utility value system of UPAI to calculate best actions. The AI is given the following priorities of units, which it will try to kill in order: priest, archmage, spellbreaker, mountain king, water elemental.
UtilityAICoward	This AI is the same as the UtilityAI, except for one difference; it has an override for giving orders to retreat whenever a friendly unit is hit. Once hit, the unit will dance around the battlefield for 6 seconds. If it is hit while dancing, it will continue dancing for 6 new seconds.

Table 7.2: Description of the AIs used in testing

7.2.1 Test in Warcraft 3

Since the testing environment was implemented as a system based on Warcraft 3, we decided to test our chosen unit setup in Warcraft 3, where both sides would be controlled by the insane difficulty AI. We ran the test five times, in order to get an idea of how the AI would micromanage in this setup and what to expect from UPAI.

Unit	Reward value
Archmage	10
MountainKing	10
Spellbreaker	6
Priest	4
WaterElemental	0

Table 7.3: Unit rewards used in testing

The results were somewhat surprising; the AI focus fire attacked the enemy mountain king, while using the storm bolt ability on the enemy archmage. While testing, we also disabled some micromanagement abilities of one player, so it could not use the summon water elemental or storm bolt abilities in battle. This player was crushed, while the micromanaging player lost no units at all.

7.2.2 UPAI vs DumbAI

The DumbAI was the easiest AI to learn to defeat, despite the fact that it was the only implemented AI without a predefined strategy. For tests with e equal to 0, UPAI explored three different solutions before learning that attacking the priests first would lead to victory. This knowledge was exploited in subsequent test runs, leading the UPAI to victory in nearly every test run without losing a singe unit. However, the DumbAI sometimes managed to kill the archmage, and even win. If UPAI could not kill all enemy priests before the archmages summoned their second water elementals, UPAI would ignore the priests, and attack the enemy spellbreakers instead. This rare occurrence threw UPAI completely off balance, and resulted in defeat. We believe that UPAI will eventually learn to target the priests in this scenario as well, if it is played several times. Since UPAI quickly learned to kill the priests first, it did not explore much, which meant that it only learned 103 cases.

The test result where e was equal to 56 was a little more varied. Here, UPAI used slightly longer time to find the best solutions. As seen from the graph in figure 7.1, the exploration caused the reward value to drop from 56 in some cases, yet once this fact was learned, UPAI would not try the same again. As with test runs where e was equal to 0, UPAI sometimes lost when it could not follow its original priority sequence. Additionally, UPAI sometimes lost the archmage, like in test runs where e was equal to 0. The value of e forced UPAI to explore, hence it learned more cases than the test with e equal to 0.

The reward values of all UPAI vs DumbAI test runs for both values of e is illustrated as a graph in figure 7.1. The total number of cases in the case base for each test run of UPAI vs DumbAI for both values of e is illustrated in figure

7.2.

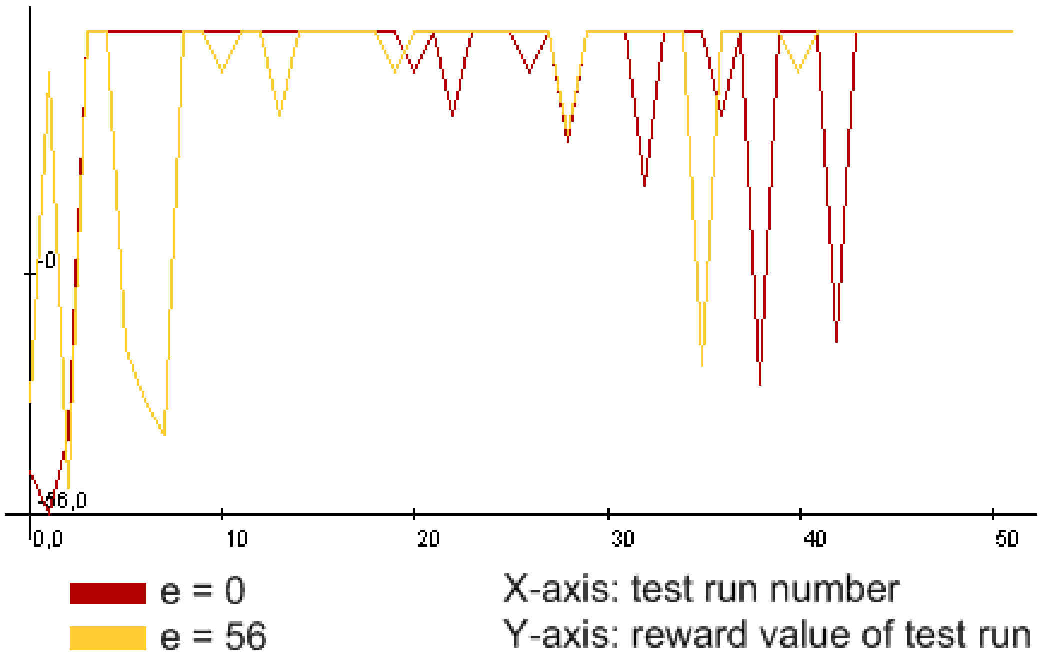


Figure 7.1: Reward values of UPAI vs DumbAI test

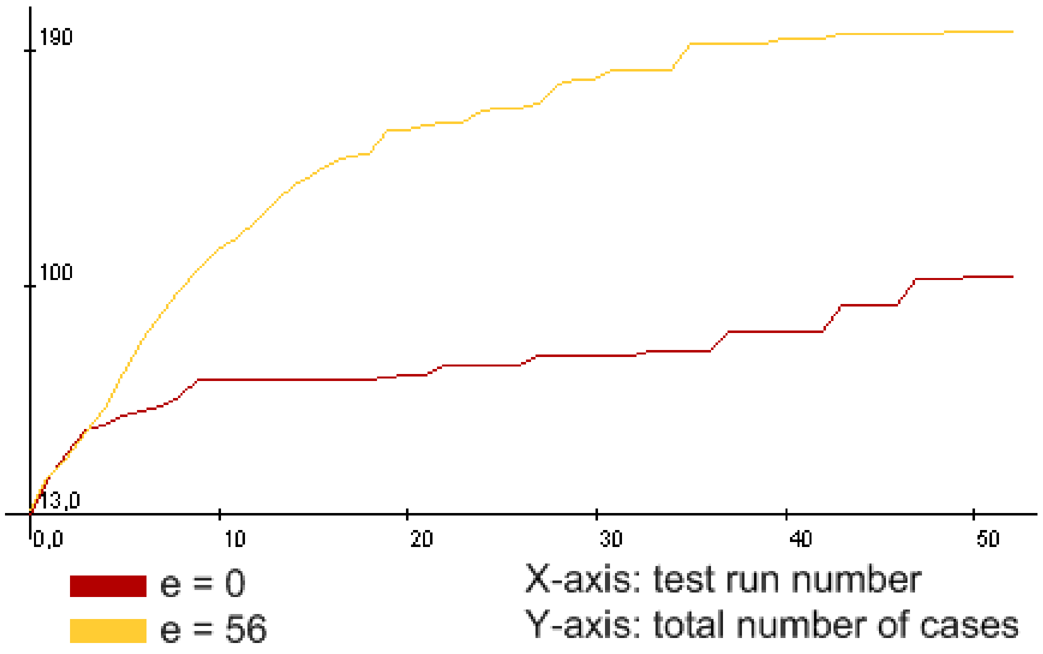


Figure 7.2: Total number of cases of UPAI vs DumbAI test

7.2.3 UPAI vs DamageOptimalAI

The DamageOptimalAI was somewhat tricky for UPAI to defeat without knowing the trick. In the first few test runs with e equal to 0, UPAI would attack some other target than the priests, while the DamageOptimalAI would try to maximize the damage done. This caused UPAI's priests to run out of mana faster than the DamageOptimalAI's, as UPAI's units received more damage per second and as a result needed more healing. Eventually, this lead to crushing defeats for UPAI. However, once UPAI leaned to target the priests first, most subsequent games were won without losses, as seen in the graph of figure 7.3. Here, UPAI receives reward values of 56 in each test run for e equal to 0. Due to the fact that UPAI quickly learned to win, the number of cases for e equal to 0 was only 71.

As for the test with e equal to 56, UPAI struggled to learn how to win without loosing a singe unit. We believed that the UPAI would be able to perform as well in this test as with e equal to 0. However, it seemed that this was not the case. We believe that this was caused because of the problem described in section 8.2.2, where UPAI explore a good solutions together with several bad ones, which results in the good solution receiving a large penalty value.

The reward values of all UPAI vs DamageOptimalAI test runs for both values of e is illustrated as a graph in figure 7.3. The total number of cases in the case base for each test run of UPAI vs DamageOptimalAI for both values of e is illustrated in figure 7.4.

7.2.4 UPAI vs FocusFireAI

Test results from test runs with the FocusFireAI were among the most interesting of the results. With e equal to 0, UPAI discovered that prioritizing the enemy mountain king first was a smart decision. This often led to victory, yet never a total victory where all of UPAI's units survived. This was because UPAI never tried attacking the priests first, as it found a solution which was better than 0, which it exploited every time. In spite of this, UPAI learned more cases with e equal to 0 than with e equal to 56. This was because the exploitation encountered more cases, due to unit a lot of unit movements of the FocusFireAI.

However, setting e equal to 56 yielded a different result. The UPAI explored several case solutions, and once it had tried attacking the priests first, it won all subsequent test runs without losing a single unit. The amount of cases learned was lower than with e equal to 0, because UPAI did not encounter as many cases with e equal to 56.

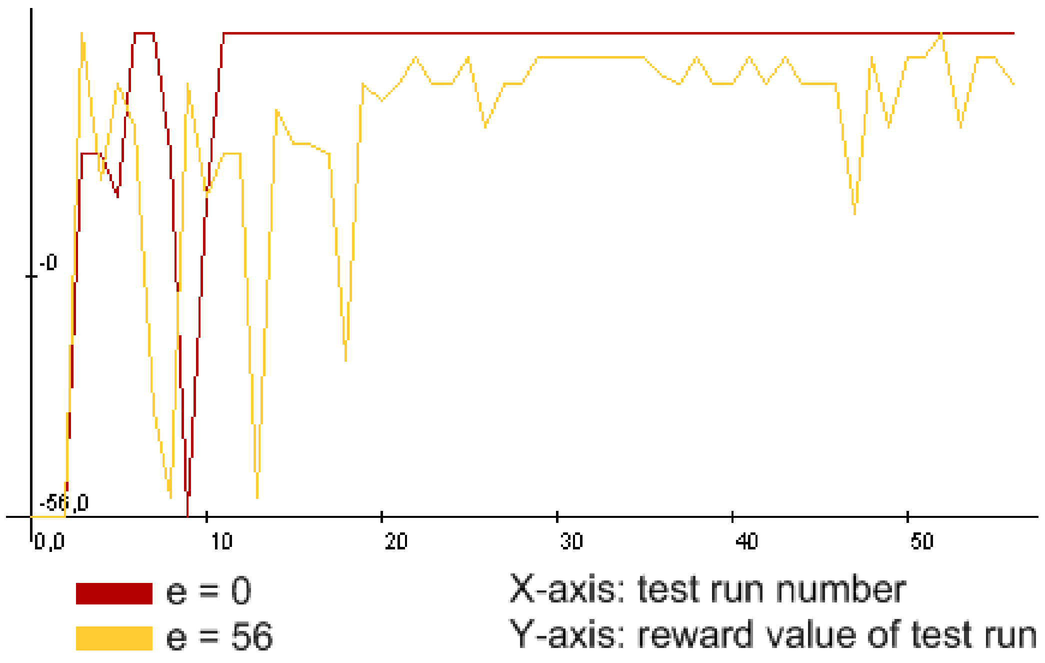


Figure 7.3: Reward values of UPAI vs DamageOptimalAI test

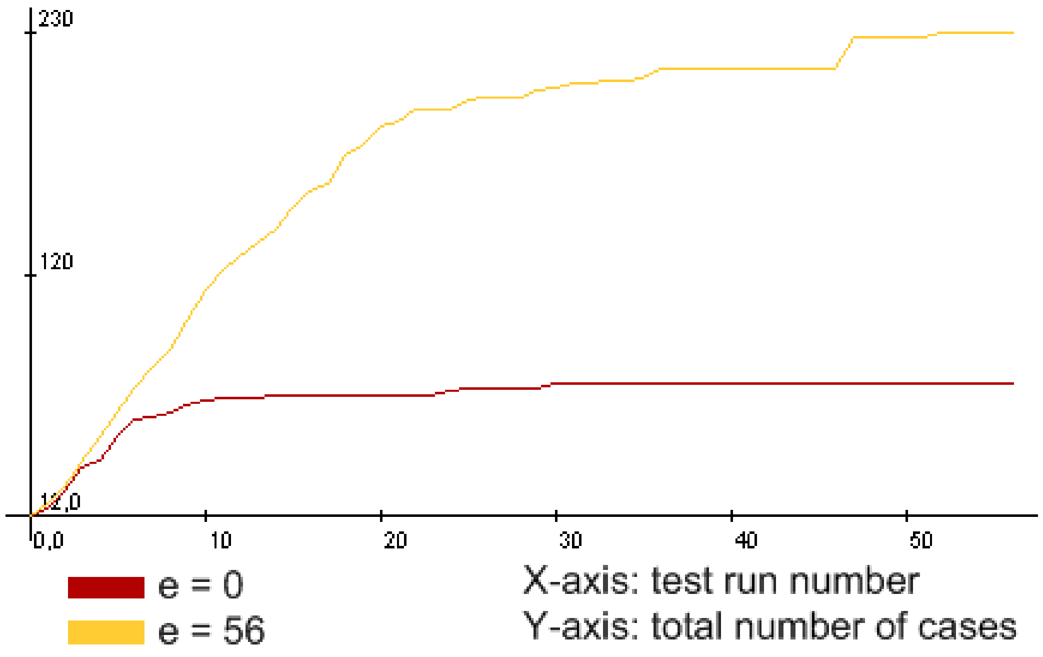


Figure 7.4: Total number of cases of UPAI vs DamageOptimalAI test

The reward values of all UPAI vs FocusFireAI test runs for both values of e is illustrated as a graph in figure 7.5. The total number of cases in the case base for each test run of UPAI vs FocusFireAI for both values of e is illustrated as a graph in figure 7.6.

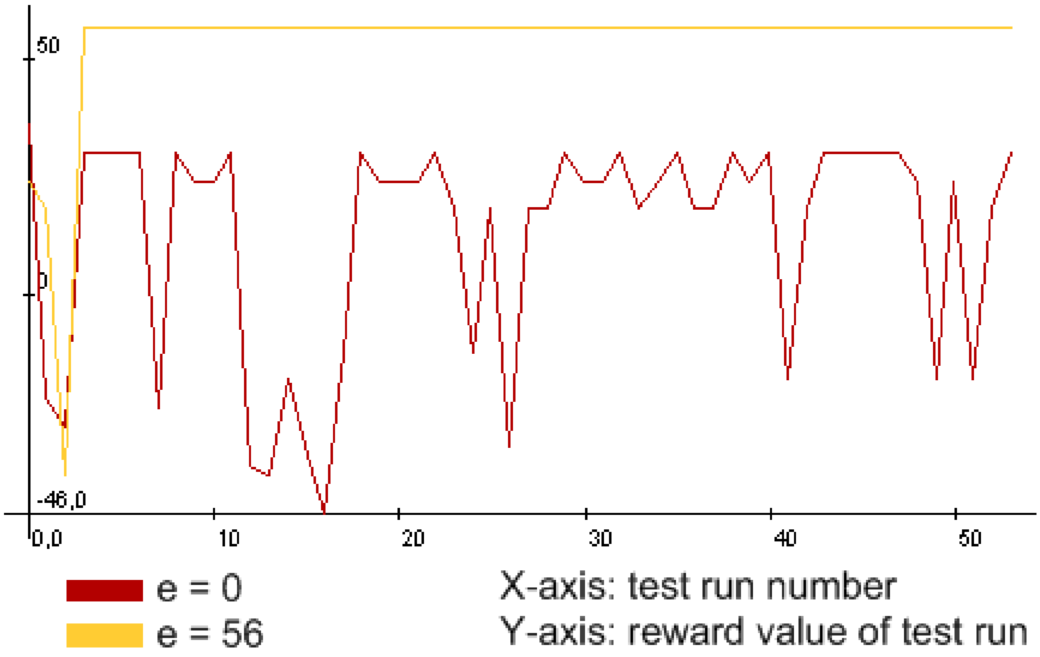


Figure 7.5: Reward values of UPAI vs FocusFireAI test

7.2.5 UPAI vs UtilityAI

The UtilityAI used exactly the same utility value calculations as UPAI, yet the priorities of the UtilityAI were hard-coded. However, UPAI learned to defeat the UtilityAI based on one difference; when the archmage and all priests were dead, the most dangerous unit was the mountain king, not the spellbreaker. As UPAI learned to prioritize the mountain king, it was able to learn to win. Tests run where e was equal to 0 showed a slow learning curve. Many solutions were tested before UPAI began winning several games in a row. However, UPAI and UtilityAI were about equally good, and UPAI achieved a victory percentage of about 50 after training.

The result from the test where e was equal to 56 was not very different from when e was equal to 0. This was because a loss meant a reward value below 0, and UPAI lost many test runs with e equal to 0, hence it was forced to explore. However, once winning, test runs with e equal to 56 produced better results, as UPAI explored more. As seen from the graph in figure 7.7, UPAI won more and better victories with e equal to 56 than with e equal to 0. The number of cases learned were less than with e equal to 0, and we believe that this was by chance.

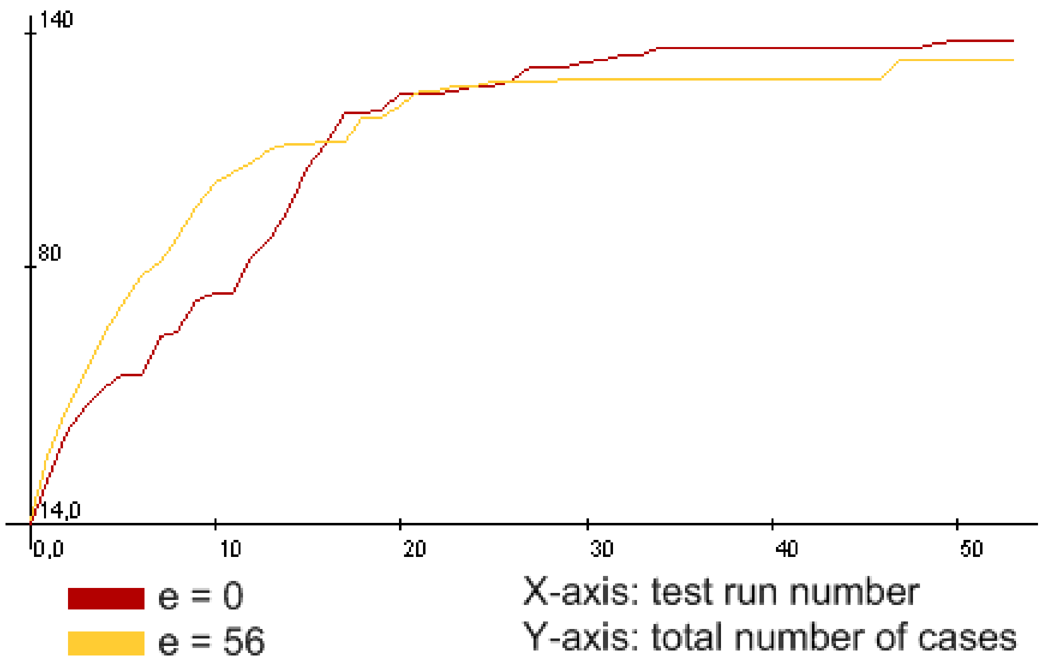


Figure 7.6: Total number of cases of UPAI vs FocusFireAI test

We had believed that more exploration would result in more cases learned, but test runs with e equal to 56 simply encountered fewer cases than with e equal to 0.

The reward values of all UPAI vs UtilityAI test runs for both values of e is illustrated as a graph in figure 7.7. The total number of cases in the case base for each test run of UPAI vs UtilityAI for both values of e is illustrated as a graph in figure 7.8.

7.2.6 UPAI vs UtilityAICoward

The UtilityAICoward was implemented in order to abuse a flaw in UPAI: dancing. However, the UtilityAICoward danced too much. When UPAI learned to attack the priests first, they ran away, which resulted in UPAI having three priests for healing, while the UtilityAICoward had two. This resulted in health points in the form of healing being lost, hence one of the reasons UPAI won was because it had more health points available. A second reason was because almost all units used during testing were ranged units. Dancing is more effective when the goal is to bait melee units into pursuit. With e equal to 0, UPAI won most of the test runs. However, in most cases only a few units survived. While observing the test runs, we discovered that the UtilityAICoward could have won more test runs if it had stopped dancing near the end, as it usually had one more unit than UPAI

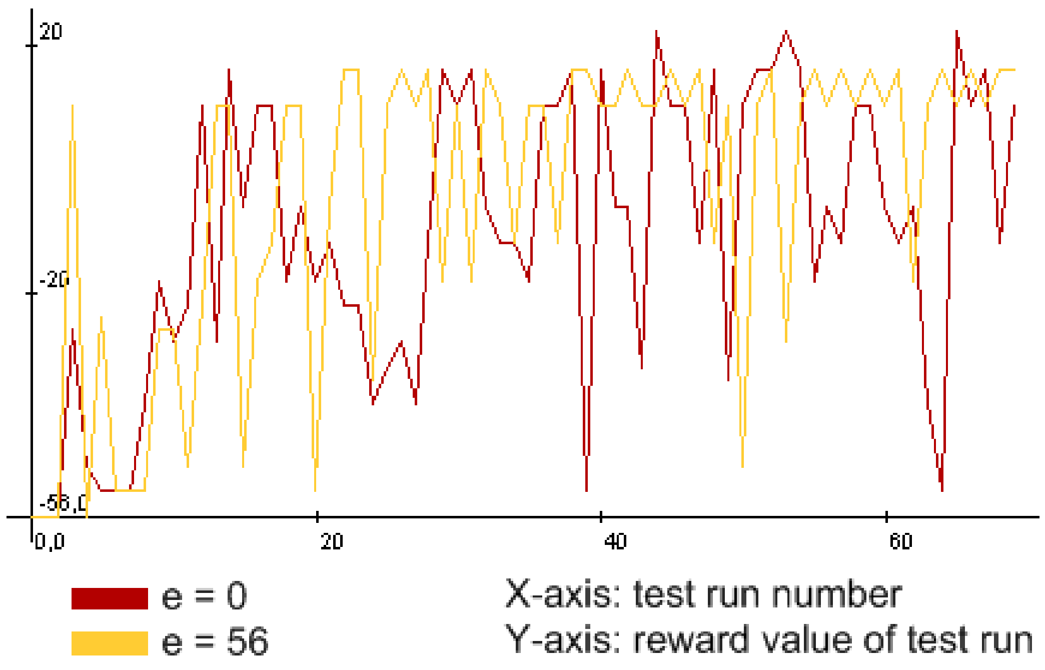


Figure 7.7: Reward values of UPAI vs UtilityAI test

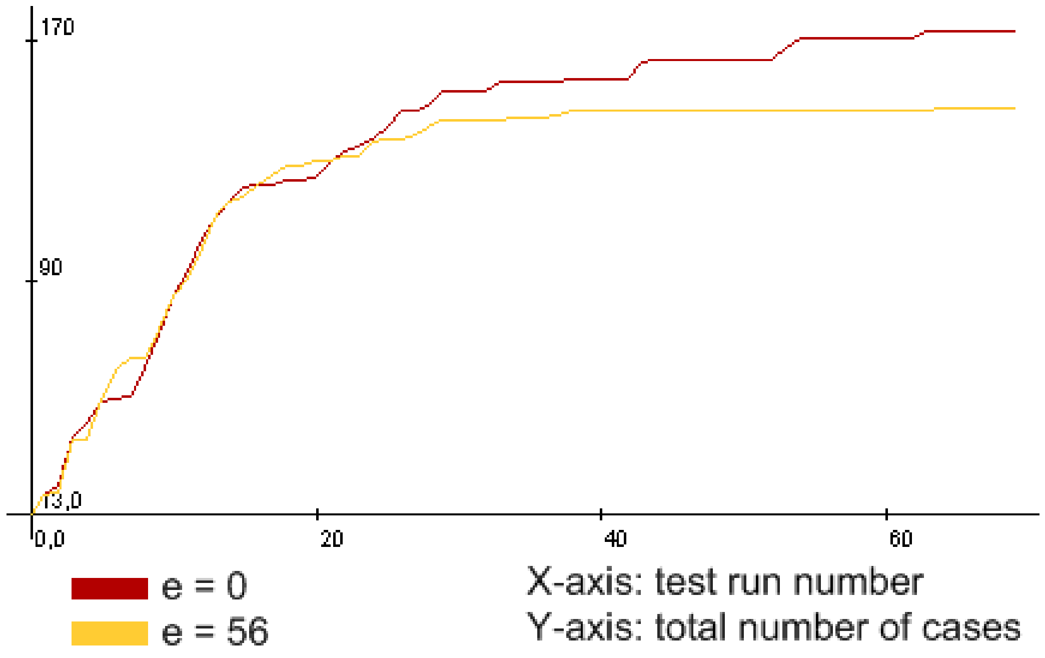


Figure 7.8: Total number of cases of UPAI vs UtilityAI test

left. UPAI learned 120 cases with e equal to 0.

The result of the test where e was equal to 56 was almost the same. UPAI learned to attack the priests first, and this leads it to victory. However, UPAI also explored more with e equal to 56, which resulted in over double the amount of cases learned compared to the test where e was equal to 0. Ultimately, UPAI won better victories with e equal to 56, yet lost a few test runs more than with e equal to 0.

The reward values of all UPAI vs UtilityAICoward test runs for both values of e is illustrated as a graph in figure 7.9. The total number of cases in the case base for each test run of UPAI vs UtilityAICoward for both values of e is illustrated as a graph in figure 7.10.

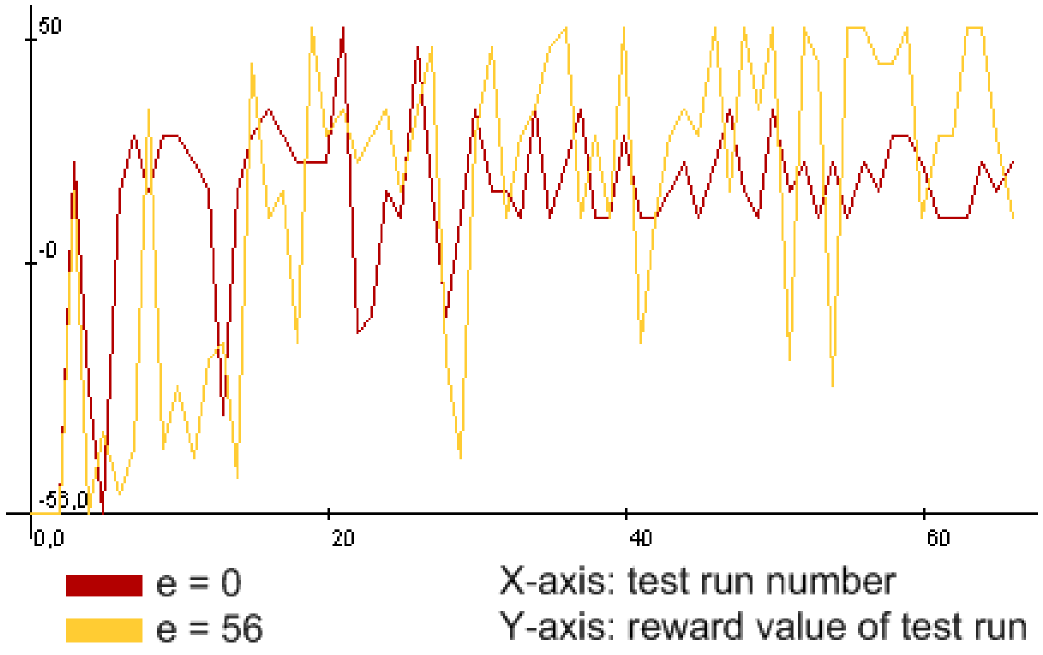


Figure 7.9: Reward values of UPAI vs UtilityAI test

7.2.7 UPAI vs SmartAI

The SmartAI was the hardest AI for UPAI to beat. With e equal to 0, UPAI won only 4 of 100 test runs. One of the advantages of SmartAI was that the units it controlled never ran away, as they did with UtilityAI and UtilityAICoward. This enabled the units to attack once or twice more before they died. During observations, we observed that UPAI did not always make the best choices (i.e. attacking another unit when there were still enemy priests alive). This was because UPAI had tried the solution of attacking the priest before, and lost without killing any

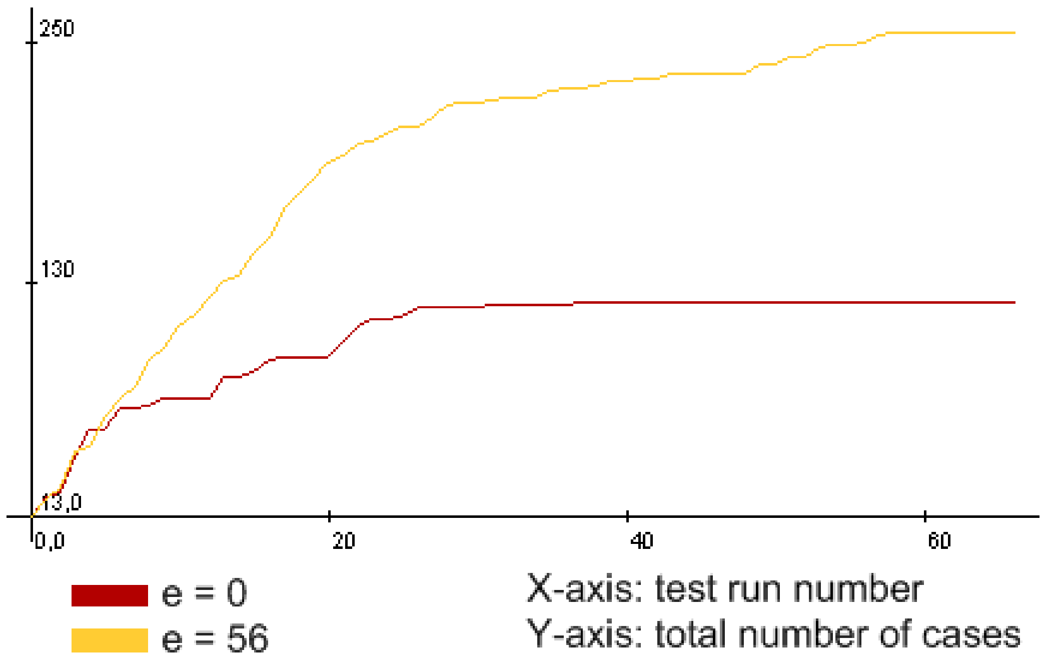


Figure 7.10: Total number of cases of UPAI vs UtilityAI test

enemy units. Hence the solution received a large penalty, because other solutions also used in the same test run were poor.

Tests with e equal to 56 showed little difference. Test runs with e equal to 0 and 56 both lost almost every test run, hence both explored equally much. In the test with e equal to 0, UPAI learned more cases than with e equal to 56. We believed that both tests would learn the same amount of cases, as the tests were both forced to explore. The UPAI probably learnt more cases with e equal to 0 by chance, similar to the UPAI vs UtilityAI test with e equal to 0, described in section 7.2.5.

The reward values of all UPAI vs UtilityAICoward test runs for both values of e is illustrated as a graph in figure 7.11. The total number of cases in the case base for each test run of UPAI vs UtilityAICoward for both values of e is illustrated as a graph in figure 7.12.

7.2.8 UPAI vs all AIs

While testing UPAI against all other implemented AIs showed interesting results, we could not be certain whether UPAI learned *general* good solutions, or good solutions for defeating a certain AI. Opponents in an RTS game are unlikely to use the exact same strategies 20 games in a row, hence we had to add diversity to

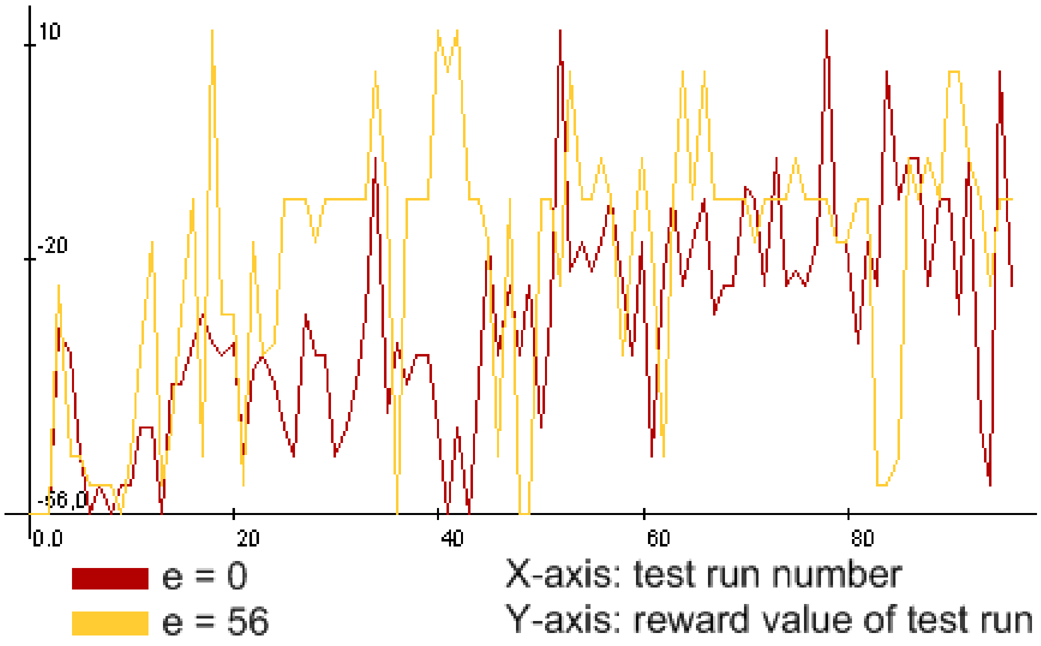


Figure 7.11: Reward values of UPAI vs SmartAI test

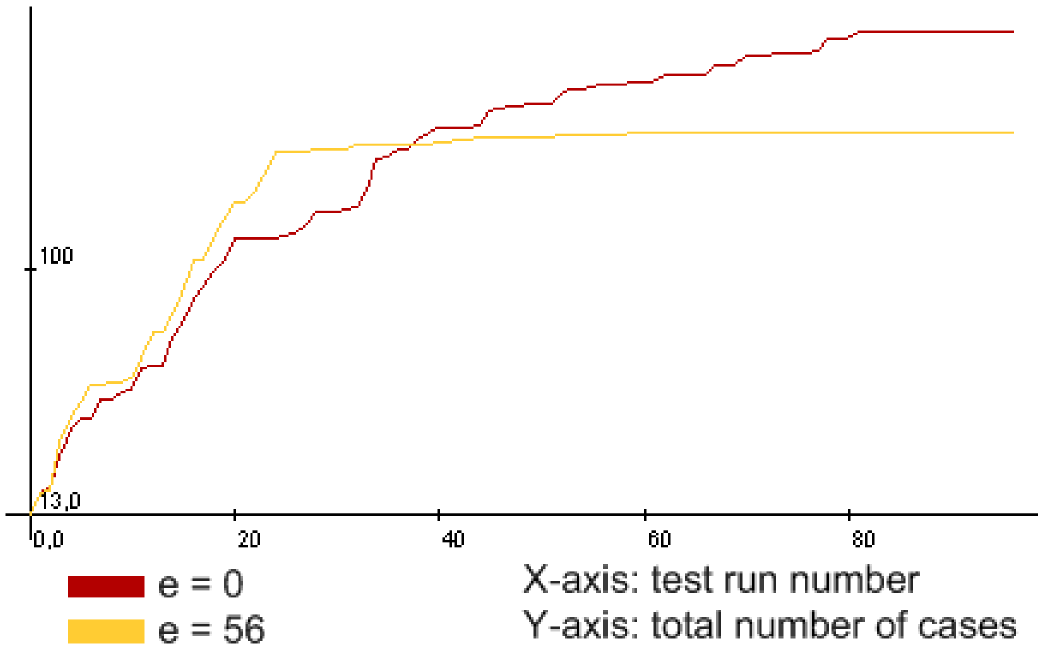


Figure 7.12: Total number of cases of UPAI vs SmartAI test

testing. We did this in two ways, both based on how Kok [9] tested his RL agent, described in section 4.2.1. The first was to run six test runs, and let UPAI play against one of the other implemented AIs each test run. The opponent AI was selected at random, and once used, it could not be selected again. This ensured that UPAI would play against all AIs exactly once. After six test runs, the cycle was reset and run again. The opponent AIs were changed randomly because we wanted UPAI to learn to find general good strategies, and we believed that we could achieve this by randomizing the order of play. Based on the results from previous tests, these test runs were only run with e equal to 56.

UPAI required considerably more test runs to learn to beat all the other AIs. In previous tests, the number of total cases learned stopped increasing at around 200, while in this test, the number was 457 by test run 150. By this test run, UPAI had learned to win. When playing against opponent AIs which did not use focus fire attack, UPAI won without losing a single unit every time by test run 100. However, the other AIs using focus fire were more difficult to defeat. Against these, UPAI usually won with one or two surviving units, thus receiving a low reward value. The result of test runs of UPAI vs all AIs are illustrated in figure 7.13. This graph spikes a lot, as the reward values were dependent on the opponent. Notice that as test runs progress, the reward values also increases. By run 120, the reward value was never less than -28, and often equal to 56. The total number of cases learned in the UPAI vs all AIs test is illustrated in figure 7.14.

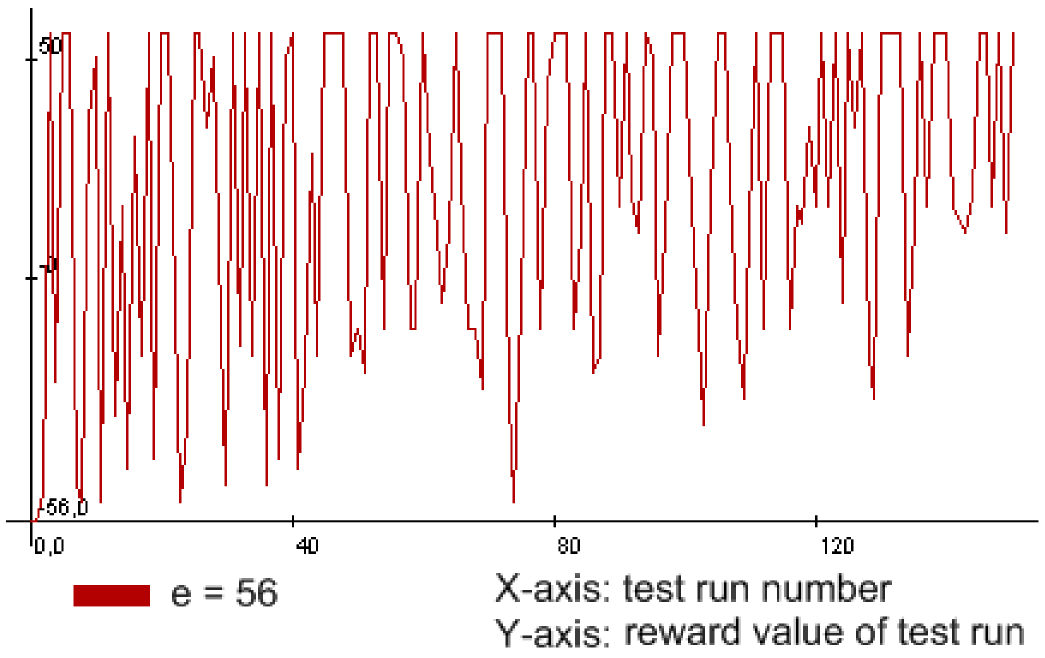


Figure 7.13: Reward values of UPAI vs all AIs, changing AI after test runs

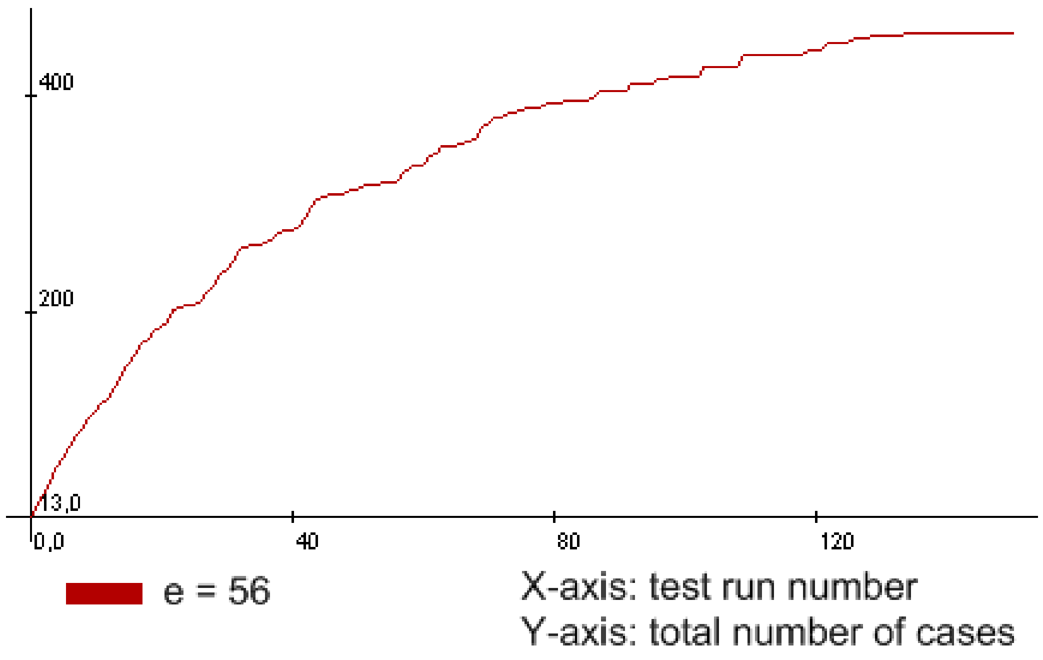


Figure 7.14: Total number of cases of UPAI vs all AIs, changing AI after test runs

Although changing the opponent AI from test run to test run allowed UPAI to learn a general strategy, it was still playing against static opponents. We also wanted to test UPAI against dynamic opponents, which would play different every test run. To achieve this, the AI of the opponent player was randomly changed to one of the implemented AIs every 20 seconds. Since all AIs were part of the random selection the same AI could be selected several times in a row.

While the number of learned cases stopped at 457 for the test of UPAI against all AIs played in different order, the number of cases grew to a total of 660 in the test against dynamic opponents. When the opponent changed AI as often as every 20 seconds, UPAI learned more cases, and used longer time to learn to win. The graph illustrating the total number of cases in the case base at each test run is illustrated in figure 7.15. We ran a total of 285 test runs before the UPAI stopped learning cases. The results of the test runs were just what we had hoped for. As we can see from the graph in figure 7.16, the reward values of later test runs are generally higher than reward values of earlier test runs. Additionally, few of the later test runs received reward values less than 0, which meant that UPAI only lost a few of the later test runs. From this graph, we can conclude that the UPAI did learn to win based on previous experiences against dynamic opponents.

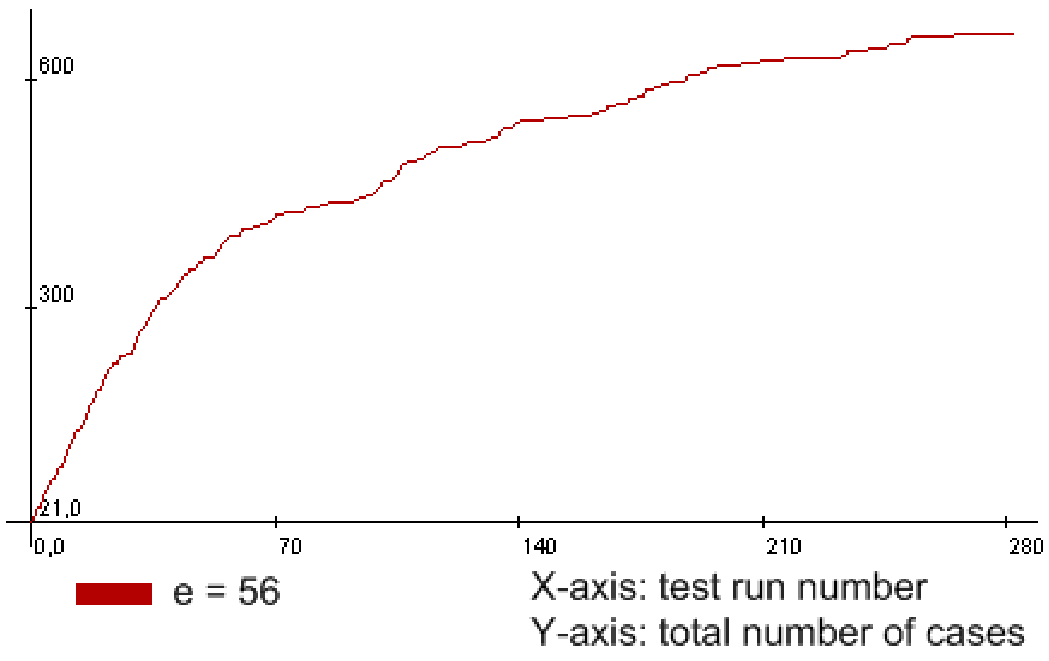


Figure 7.15: Total number of cases of UPAI vs all AIs, changing AI during test runs

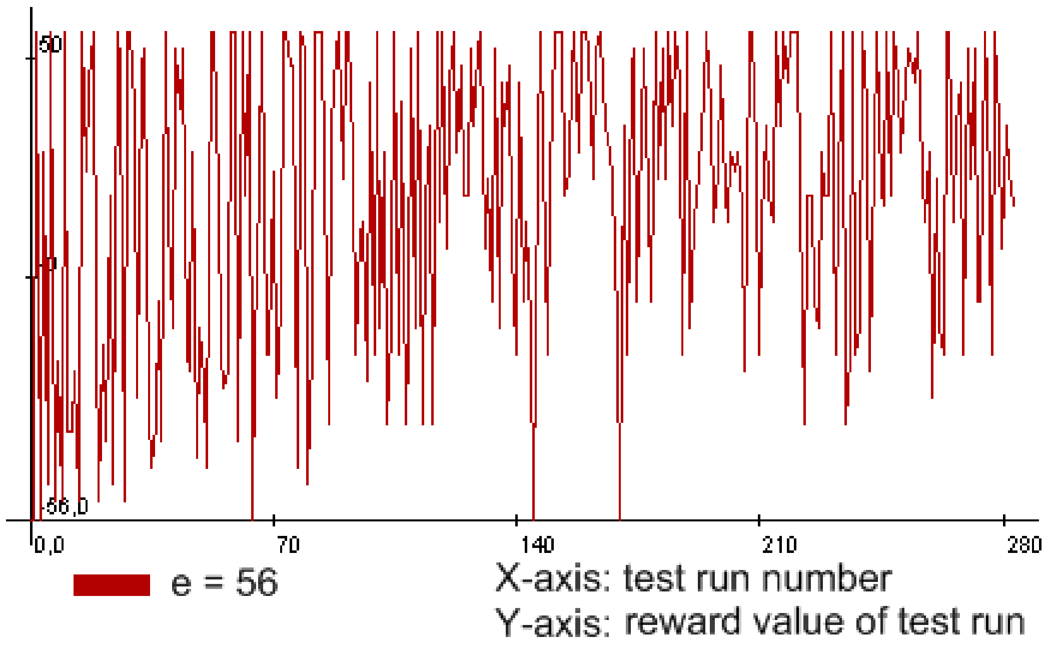


Figure 7.16: Reward values of UPAI vs all AIs, changing AI during test runs

Chapter 8

Discussion

In this chapter, we evaluate our project in terms of choice of game environment in section 8.1, the implementation of UPAI in section 8.2, and test results in section 8.3.

8.1 ORTT as game environment

When searching for a game environment to use for implementation and testing of our system, we severely underestimated the time it needed to implement an RTS game environment. Even though we used a free A* pathfinder implementation for unit pathfinding, it took two weeks of development time to fine-tune the pathfinder for use with ORTT. Even then, the system had a few problems which we were not able to solve.

The first problem with pathfinding in ORTT was that sometimes units would get stuck, when moving from one point to another. In order to continue movement, the unit had to receive a new order, which happened only when either a friendly or enemy unit was killed on the battlefield. When a unit was stuck, it would not attack either, reducing the total damage per second a player could do. This resulted in an unfair advantage to the opponent, yet this problem happened to both players, and added some randomness to the environment. However, this was a disadvantage for UPAI. If a unit became stuck when UPAI was testing a new solution, the new solution could receive a penalty if UPAI lost the test run. This solution might have been the best solution, but it would not be used again in a long time, as UPAI believed it to be a bad solution. Another problem seen in ORTT, was that units sometimes stood on top of each other, and other times moved right through other units. This rarely happened, but it is clearly not a desirable effect of an RTS game. The problems with unit movement could be fixed by implementing separation-like flocking behavior for units.

While test runs only involved a maximum of 22 units, executing pathfinding for all units at the same time sometimes slowed down the visuals of the simulation. If the time taken to update the game and draw the updates take longer than one game tick (16 milliseconds for ORTT), XNA will prioritize the update. This meant that XNA would skip the drawing of some frames, to update the game logic instead. The result of this was somewhat slow visual updates at some points.

Despite the flaws of ORTT, it was easy to implement a CBR/RL system in the XNA framework. Since the XNA framework is built on top of C#, it was possible to pause ORTT while it was running, and debug anywhere in the code. This proved to be very helpful for finding bugs during the implementation and testing of our system. Additionally, setting environment parameters such as exploration rate and opponent AI prior to each run was easily done with XML files.

8.2 UPAI

The system we implemented had the same architecture as CaT, yet the systems were focused on different learning tasks. While UPAI did learn to win, some limitations emerged during testing, which were not found in CaT. One of the most important differences where timing. Where actions in CaT are overall strategy decisions on the macromanagement level, actions in UPAI are short and fast micromanagement decisions. The speed in which the game state changes caused some problems with cases and reward values, which are discussed in the following sections.

8.2.1 Case representation

Moving important unit attributes such as position and health points from the case representation to the utility value system seemed like a good idea. While this division of attributes worked to some extent, it did cause a few problems such as unintelligent unit behavior.

As an example of unintelligent behavior, consider the case illustrated in figure 8.1; the best solution chosen by UPAI was to prioritize attacking the healthy spellbreaker. However, the enemy archmage with only 42 health points left is ignored. The obviously best choice had been to kill the archmage first, since it was almost dead. This problem was a result of the fast transition between game states. UPAI would order its units to attack the archmage, and the units would comply. However, UPAI lost a unit just before the archmage could be killed, and this resulted in a change of game state. In the new state, the best solution of the case matching the current game state was to attack a spellbreaker, hence the archmage was ignored. This rapid change in game states also resulted in some unnecessary unit movement.

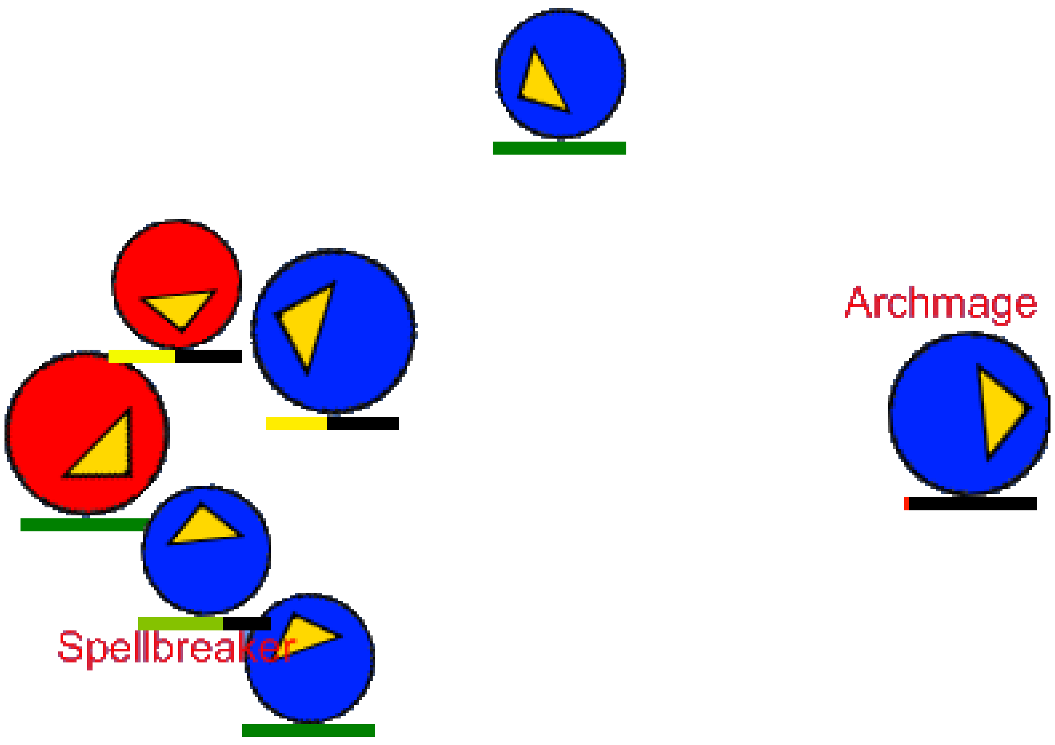


Figure 8.1: Example of unintelligent unit behavior

Another example of a weakness with the case representation can be found in figure 8.2. The two cases illustrated here are equal in terms of case similarity value, yet very different in terms of health points. In both cases, UPAI chose to attack the archmage, while the best choice had been to attack the most damaged unit. This problem could be solved either by extending the case representation to include unit health points as well, or add behavior to the utility value system, overriding priority orders when units with low health were nearby.

8.2.2 Reward value

During the first test runs, UPAI would try different solutions to cases, in order to find good solutions which could lead to victory. Since all solutions used in a test run were rewarded by a distribution of a single reward value, a test run

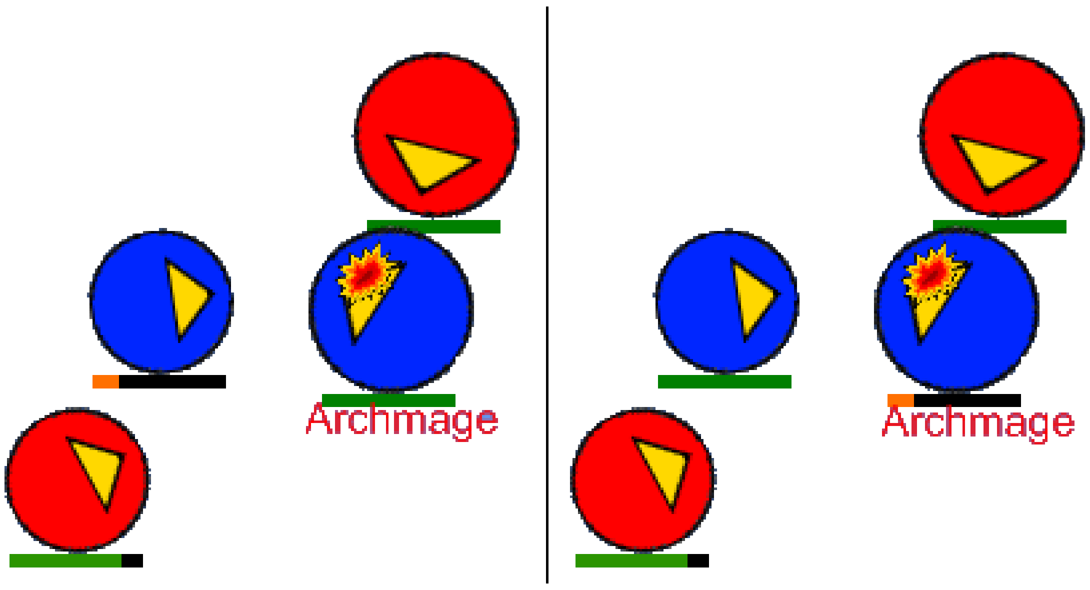


Figure 8.2: Case representation weakness with health points

which received a large penalty could contain some good solutions, and vice versa. These good solutions would not be used again in a long time, since they were part of a bad test run. This problem was observed several times during testing. For example, in later test runs of UPAI vs DamageOptimalAI with e equal to 56, UPAI would always attack the enemy mountain king before killing the last priest, due to the fact that this solution was the solution with the highest reward divided by times used. This resulted in victory, but UPAI always lost one or two spellbreakers. This problem could be solved by forcing UPAI to explore; for example retrying solutions which had only been used once, to rule out the possibility of ignoring good solutions which had been used during a bad test run.

Another cause of the problem described above was that some solutions were rewarded even though they had not been tested, since transition between the different game states could happen very fast. For example, if two units were killed within two seconds, this would result in two different states. For each state, UPAI would find the case matching the state, and execute the best solution, or create one if the other solutions were unsatisfactory. The solution to the case which matched the state in the second between the unit deaths would only be active for one second, hence it was difficult to evaluate the effectiveness of the solution. Nevertheless, the solution was rewarded in the same way as the other cases.

8.2.3 The utility value system

As the CBR part of UPAI was only concerned with which units were on the battlefield, the utility value system dealt with information such as health points, mana points and position. However, these attributes only provided information on the current game state situation, and gave no information of how they had changed since the previous game state. Hence it was impossible for UPAI to measure change in mana points, health points or position. This lead to problems in situations where enemy units were running away from UPAI's units, as all units controlled by UPAI could potentially chase the enemy unit endlessly. Is also made UPAI very vulnerable to dancing units. The UtilityAICoward used dancing, but lost most games because priests were dancing instead of healing. In battles where UPAI control only melee units, it would lose to an opponent which orders his units to dance.

8.2.4 Unit retreat order

Units controlled by UPAI were ordered by the utility value system to retreat whenever their health points dropped below 10%. However, after successfully retreating, units would stand idle on the edge of the battlefield. Having successfully retreated, units should be moved away from the battlefield, to simulate that the unit managed to run further away. This could reduce the penalty for losing the unit in combat, by giving for example half penalty for units which managed to run away. Currently, ordering a unit to retreat does not give any benefit over losing it in combat. Hence UPAI was at a disadvantage against the SmartAI, which never ordered units to retreat.

8.2.5 Exploration vs exploitation

As described in section 2.2.2, finding a good balance between exploration and exploitation is often a difficult task with reinforcement learning. In our project, we tested two approaches for all tests, which were controlled by the exploration parameter e. In the first approach, e was set to 0, which meant that when a winnable solution was found, it should be exploited every subsequent test runs. This resulted in victories for UPAI, but it lost some units against almost all AIs in every test run. In the second approach, e was set equal to 56, which meant that no solution found could ever be good enough. This forced UPAI to explore new solutions whenever it had the opportunity to do so. While the second approach used somewhat longer time to achieve frequent victories against most of the test AIs, the victories were better. UPAI lost less units in tests of the second approach, and this is one of the goals of good micromanagement in addition to winning.

8.3 Test results

The results from test runs of UPAI showed that our system did learn to win by prioritizing enemy units correctly in most situations. We observed some situations where UPAI prioritized a water elemental instead of a priest, but these situations rarely occurred. Additionally, UPAI also won test runs where these situations occurred, because it usually had an advantage at that point in a test run.

Six AIs were implemented in order to provide opponents for UPAI during testing. Test runs against these showed that UPAI learned to defeat most of these after several test runs. However, the SmartAI proved to be better than UPAI, as UPAI won only 4 of 100 test runs with e equal to 0, and 12 of 100 test runs with e equal to 56.

UPAI could be set to focus on exploration or exploitation by the exploration parameter e. Through tests of UPAI vs the six AIs, we discovered that forcing UPAI to explore resulted in more victories and fewer units lost in battles. However, the number of test runs needed to win when focusing on exploration was higher than when focusing on exploitation. Since we were concerned with finding the best solutions and not really cared how many test runs this required, we decided to focus on exploration by setting e equal to 56 for both the UPAI vs all AIs tests.

Learning to win in the UPAI vs all AIs tests required more test runs than we had anticipated, both for the static opponents changing AIs between test runs and the dynamic opponents changing AIs during test runs. In the first test with static opponents, we collected data from 150 test runs, and observed that the average reward value increased during the test runs. While UPAI lost most of the early test runs, it won nearly all the latter test runs, hence we could conclude that UPAI had learned to defeat the static opponents we implemented. Learning to win against dynamic opponents was much more difficult for UPAI, because the opponents were dynamic. With 600 cases, UPAI did not achieve frequent victories before test run 200. Like with static opponents, we observed that the average reward value increased with each test run. Hence we concluded that UPAI had learned to win, or at least make a good attempt at winning.

Chapter 9

Conclusions and Further Work

This chapter will close our project, by first evaluating how we reached the goal of the project in section 9.1. In section 9.2, we will present ways to improve the system we implemented.

9.1 Conclusion

In this project we have designed and implemented a CBR/RL hybrid system for learning which enemy units to target in given situations during a battle in an RTS game. We investigated the effects of learning with different focus on exploration and exploitation, and trained the unit priority system (UPAI) so that it learned to defeat both static and dynamic AI opponents.

The goal of this project was:

> To study how learning from experience during game playing can improve the micromanagement of a computer controlled AI player.

We feel that we partly managed to reach our goal, as the system we implemented improved the micromanagement of a computer player, by learning to win by prioritizing enemy units. However, learning to prioritize correctly required many test runs just for the units used in our test setup. As the number of cases for the unit setup used in testing exceeded 600, the number of cases and test runs required in a full game with 40 different units would be too large to learn. Our system also had some other problems, such as unintelligent unit behavior when attacking and lack of option to track unit attributes over time.

While our system did improve the micromanagement of a computer player, it would not be feasible to use the system in an RTS game without some modifications to our system. We do not recommend using our implemented system any further, unless some modifications described in section 9.2 are done.

9.2 Further work

This section lists some possible improvements of UPAI, which we would want to focus on should we decide to continue work on the project.

9.2.1 Index and sort cases

When implementing the CBR system of UPAI we decided to not index cases, since we believed that the number of cases would be rather small. However, as the test in section 7.2.8 showed, the number of cases exceeded 600 just for the test setup. With more units and test setups, the number of cases will increase exponentially, and searching through all cases every second would be computationally infeasible.

9.2.2 Prioritize hurt units

UPAI should not ignore severely hurt units if they are not a priority. Attacking a unit if it can be finished of with a few attacks is always a good choice of action, even if the target in question is not a priority. Since the utility value system already calculates the utility value for attacking units, it should calculate additional low-health utility values. If a unit with sufficiently low health points is found, it should be attacked.

9.2.3 Observe changes in game state

As described in section 8.2.3, our system does not measure attribute values over time. In order to be able to react to certain situations, it should be able to observe changes in attribute values over time. For example, if an enemy unit is running away, there will be a large value change for the position attribute between game states. Also, if the health points of a friendly unit are decreasing at a quick rate, it probably means that the unit is being focus fire attacked. UPAI should detect this, and order the unit to try to shake off the attackers by running away.

9.2.4 Add other goals

UPAI has currently only one goal: win the battle by losing as few units as possible. However, during a battle players often wish to achieve other less important goals in addition to victory, such as not losing important units, and not using mana points if it is not necessary. For example, in Warcraft 3 it is considered bad micromanagement to lose a hero unit. Our implemented system does not take this into account.

9.2.5 Add support for planning

Some advanced micromanagement maneuvers require players to plan ahead. An example of this is the surround maneuver, where several units move to surround a specific enemy unit, preventing it from moving. Adding support for planning in the UPS would enable units to execute such advanced maneuvers.

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. In Artificial Intelligence Communications, 7:39-59, 1994. [cited at p. v, 8, 9]

[2] David W. Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In ICCBR 2005, number 3620 in LNS, pages 5-20. Springer-Werlag, 2005. [cited at p. 2, 3, 7, 26]

[3] Michael Buro. Call for ai research in rts games. Proceedings of the AAAI-04 Workshop on Challenges in Game AI (pp. 139-142). AAAI Press, 2004. [cited at p. 1, 2, 7, 19]

[4] Blizzard Entertainment. Warcraft iii - basics -> armor and weapon types. classic.battle.net/war3/basics/armorandweapontypes.shtml (accessed 28.05.09). [cited at p. vii, 15]

[5] Blizzard Entertainment. Warcraft iii - basics -> spell basics. classic.battle.net/war3/basics/spellbasics.shtml (accessed 26.05.09). [cited at p. 6]

[6] Richard Evans. Ai in games: From black and white to infinity and beyond. www.gameai.com/blackandwhite.html (accessed 12.02.09). [cited at p. 20]

[7] Bruce Geryk. A history of real-time strategy games, 2001. www.gamespot.com/gamespot/features/all/real_time/ (accessed 28.05.09). [cited at p. 7]

[8] Nicolas Imrei. Reinforcement learning in real-time strategy games. School of computer science and software engineering, Monash university, 2004. [cited at p. 22, 33]

[9] Eric Kok. Adaptive reinforcement learning agents in rts games. Master thesis number INF/SCR-07-73, Intelligent systems group, Utrecht University, 2008. [cited at p. 11, 21, 66]

[10] LiveGraph. Livegraph: The real-time data graph plotter. www.live-graph.com (accessed 08.06.09). [cited at p. 51]

[11] Tom Mitchell. *Machine Learning.* McGraw-Hill Science/Engineering/Math, McGraw-Hill Science/Engineering/Math, 1997. [cited at p. 8]

[12] Matthew Molineaux, David W. Aha, and Philip Moore. Learning continuous action models in a real-time strategy environment. In FLAIRS 2008, pp. 257-262, Springer, 2008. [cited at p. 27]

[13] Nils J. Nilsson. *Artificial Intelligence: a new synthesis.* Morgan Kaufmann Publishers, Inc., Morgan Kaufmann Publishers, Inc., 1998. [cited at p. 1]

[14] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Case-based planning and execution for real-time strategy games. In ICCBR 2007: Lecture Notes in Computer Science, Volume 4626/2007, pp 164-178, Springer Berlin/Heidelberg, 2007. [cited at p. 2, 11]

[15] Jonathan Schaeffer, Vadim Bulitko, and Michael Buro. Bots get smart. IEEE Spectrum Online, 2008. www.spectrum.ieee.org/dec08/7011 (accessed 03.03.09). [cited at p. 1]

[16] Semei. Xna tutorials and xna tools - pathfinding sample for xna. www.ziggyware.com/readarticle.php?article_id=162 (accessed 28.05.09). [cited at p. 16]

[17] Manu Sharma, Michael Holmes, Juan Santamaria, Arya Irani, Charles Isbell, and Ashwin Ram. Transfer learning in real-time strategy games using hybrid cbr/rl. In IJCAI-07, pp. 1041-104, Morgan Kaufmann, 2007. [cited at p. vii, 11, 28, 29]

[18] Sindre Berg Stene. Artificial intelligence techniques in real-time strategy games - architecture and combat behavior. Master thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2006. [cited at p. 1, 11, 25, 32, 33]

[19] Tomasz Szczepański. Case-based reasoning for improved micromanagement in real-time strategy games. Specialization project, Department of Computer and Information Science, Norwegian University of Science and Technology, 2008. [cited at p. 2, 24, 33, 38, 39, 53]

[20] Luis Valente, Aura Conci, and Bruno Fiejo. Real time game loop models for single-player computer games. In SBGames 2005 - IV Brazilian Symposium on Computer Games and Digital Entertainment, Citeseer, 2005. [cited at p. 14]

[21] Mitch Walker. Xna team blog : What is the xna framework. blogs.msdn.com/xna/archive/2006/08/25/724607.aspx (accessed 28.05.09). [cited at p. 14]

[22] James Wexler. Artificial intelligence in games: A look at the smarts behind lionhead studio's 'black and white' and where it can and will go in the future. University of Rochester, 2002. [cited at p. 19]

[23] Michael Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons, John Wiley & Sons, 2002. [cited at p. 9]

Appendices

Appendix A

Running the project software

The system implemented in this project is attached to this report. It was implemented in Microsoft's XNA framework, which is built on top of the .NET framework. Hence running the implemented system requires both XNA Game Studio and .NET framework 3.5. Running ORTT require a desktop resolution larger than 1200 x 1000.

After installing the required software, the system may be run by running the "ORTT.exe" file. This will start a battle between two players, where each one is controlled by AIs specified in the AI.xml file found in the same folder as "ORTT.exe". The AI controlling a player may be changed by typing in new values between the AI tags of "AI.xml". The first tag is the AI for red player, while the second tag is the AI for the blue player. A list of possible AI values may be found in the text file "Possible AI values.txt". Note that only the red player may use the CBRAI.

Also included with the project software are the files "caseBase.xml" and "Copy of caseBase.xml" The "caseBase.xml" file is the current case base, which will be updated after each simulation. "Copy of caseBase.xml" is the case database which was used as a foundation during testing. It is not recommended to modify either of these files, as the system may not work correctly without it. To exit ORTT, either wait for the simulation to finish, click the right mouse button, or the close button in the menu bar.

Appendix B

RTS terms used

The special RTS game terms we used in our report are listed in table B.1.

Term	Meaning
Unit	A unit is a solider in an RTS game. Each player controls several units, which are used to battle and defeat the other players. Units have different attributes, and commercial RTS games today consists of about 20 - 50 different units.
Game	A round from start to finish in an RTS game is called a game. Similarly, a game of chess starts by setting up the pieces, and ends in checkmate for one of the players.
Battle	A battle is a small skirmish during a game. A game may contain many battles.
Health points	The measurement of a unit's life. When the health points of a unit reach 0, the unit dies and is removed from play.
Mana points	The measurement of a unit's magic power. Each ability of the unit costs a predefined number of mana points to execute. If the unit's total mana points are too low, it cannot use the ability.
Focus Fire	A player use all his units to attack a single enemy unit. Focus fire is usually a better strategy than to attack several enemy units at once.
"Dancing"	Dancing refers to the act of using units as bait. If a player orders a unit to pull back once it is hurt, and order it to run around the battlefield while it is being pursued by the enemy, this unit is "dancing". While enemy units are pursuing the dancing unit, the player may attack these units while they are running around.
Skill	A skill is a passive attribute of a unit. It is always active, and cannot be deactivated. Use of skills cost no mana points.
Ability	An ability is an activated special ability of a unit. Each ability costs mana points to execute. Use of abilities is activated by the player when he chooses to do so. To successfully use an ability, all preconditions must be fulfilled. There can be several preconditions, but is usually whether the unit has enough mana, and a target to use the ability on.
Melee unit	A melee unit, as opposed to a ranged unit, have an attack range of 0. This means that melee units must be right next to the target they want to attack.

Table B.1: Special RTS game terms used in report

Appendix C

Unit attributes

C.1 Unit attributes in ORTT

All units have the following attributes, which are visible to all players. Hence AI's may use this information in case representations. Some units have abilities as well, but these are not visible to other players.

Static attributes, not changeable during gameplay:

- Name
- Radius (in pixels)
- Range (in pixels)
- Move speed (pixels per timeframe)
- Minimum damage
- Maximum damage
- Maximum mana points
- Maximum health points
- Armor value
- Armor type
- Damage type

Dynamic attributes, likely to change during gameplay:

- Position
- Heading
- Current health points
- Current mana points

C.2 Attributes of units used in testing

This table lists the attributes of all units used in testing. For a description of the attributes, see table C.2

Name	Size	Rng	Spd	Dmg	HP	MP	Ar	ArTyp	DmgTyp
Archmage	31.8	300	3.2	21-27	450	285	3	Hero	Hero
MountainKing	31.8	6	2.7	24-36	700	225	2	Hero	Hero
Spellbreaker	25.2	125	3.0	13-15	600	250	3	Medium	Normal
Priest	22.5	300	2.7	7-8	290	200	0	Unarmored	Magic
WaterElemental	27.8	150	2.2	16-24	425	0	0	Pierce	Heavy

Table C.1: Attributes of units used in testing

Short Name	Meaning
Size	Size, in pixel radius. This is the measurement of how much space the unit takes on the battlefield.
Rng	Attack range, in pixel radius. This is how far a unit can shoot. Melee range is defined as 6.
Spd	Movement speed, the value is the number of pixel a unit can walk each frame.
Dmg	Damage, the minimum and maximum damage the unit does.
HP	Health points, the measurement of unit life. A unit dies if health points reach 0.
MP	Mana points, the measurement of unit magic energy. Each ability costs a defined amount of mana points.
Ar	Armor rating, the numeric armor value of a unit. Each point of armor rating reduces the damage a unit takes by 6&.
ArTyp	Armor type, one of the six armor types of table 3.3.
DmgTyp	Damage type, one of the seven damage types of table 3.3.

Table C.2: Description of unit attributes

Name	Mana cost	Cooldown	Available to	Description
Summon water elemental	125	60 seconds	Archmage	Summons a water elemental
Storm bolt	75	9 seconds	Mountain king	Throw a storm bolt
Heal	5	1 second	Priest	Heals a friendly unit.

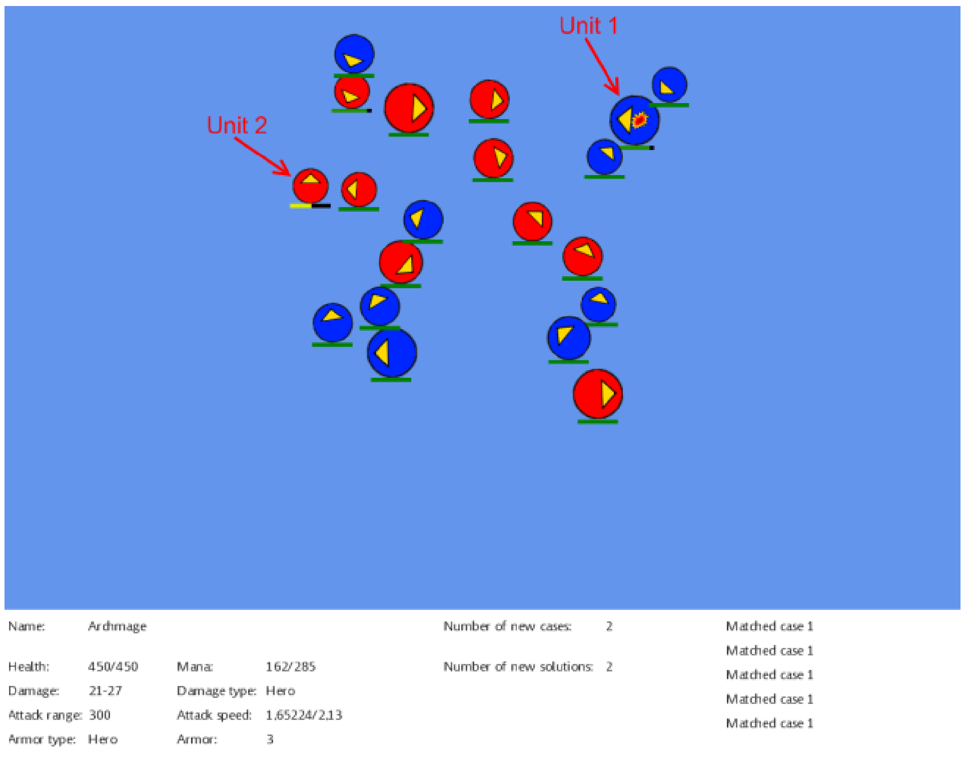
Table C.3: Attributes of abilities

Appendix D

System execution

This chapter contains a walkthrough of a battle, and what UPAI does at each step when it is controlling an AI in ORTT.

1. Player 1 is set to be controlled by UPAI.

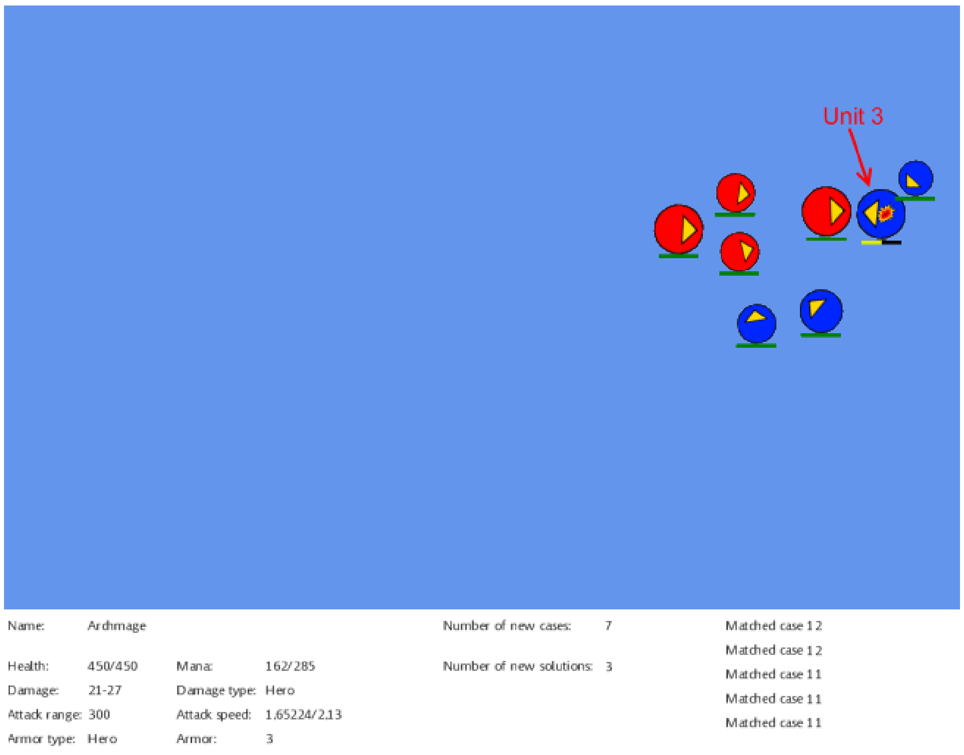


```
Name:         Archmage                                         Number of new cases:      2      Matched case 1
                                                                                                Matched case 1
Health:       450/450     Mana:          162/285               Number of new solutions:  2      Matched case 1
Damage:       21-27       Damage type:   Hero                                                   Matched case 1
Attack range: 300         Attack speed:  1,65224/2,13                                           Matched case 1
Armor type:   Hero        Armor:         3
```

Figure D.1: Execution of UPAI, step 2

2. ORTT is run, and the battle begins. UPAI will read all cases from the case base, and try to find and exact match to current game state in the case base. If no case is found, UPAI will store the current game state as a new case in the case base, and create a new solution for this case. If an exact match is found in the case base, UPAI will use the best solution of this case, or try to create a new one if the best solution has a lower value than the exploration parameter. The case seen in figure D.1 is matched to case 1, and solution 0 is used, and added to a list of used solutions. It is not possible to see which solution number is executed in the GUI. The enemy archmage (unit 1) get the highest priority value.

3. The priority values from the solution of step 2 is passed to the utility value system. This system calculates utility values of all units controlled by UPAI. In the case seen in figure D.1, the priests controlled by UPAI are ordered by the utility value system to use the heal ability on the damaged priest (unit 2). All other units controlled by UPAI are ordered by the utility value system to attack the enemy archmage (unit 1).

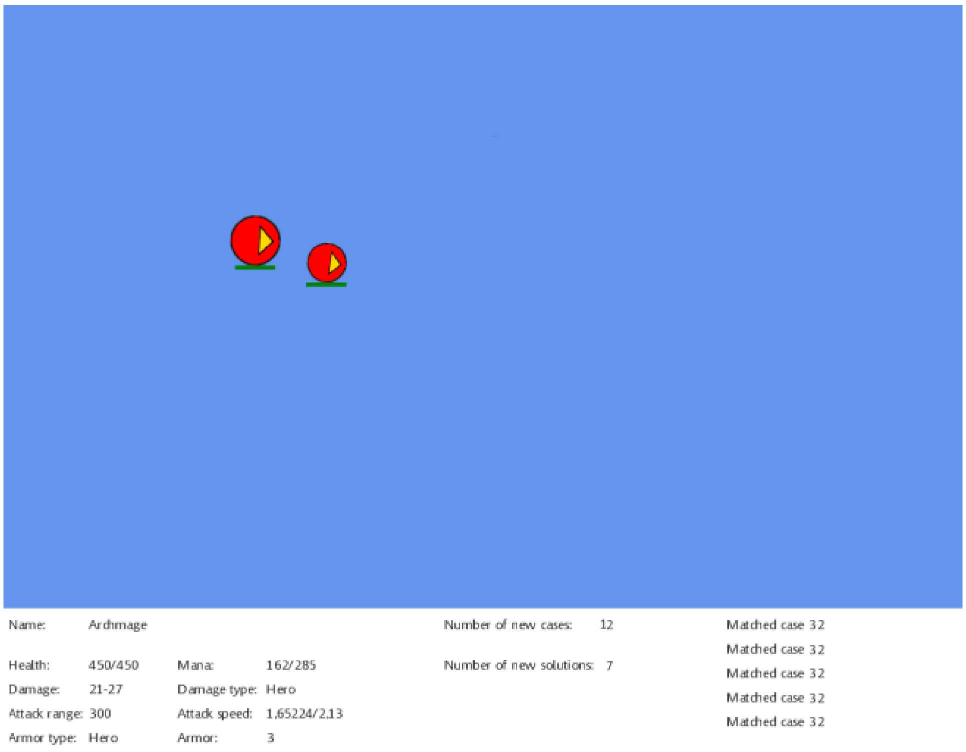


```
Name:         Archmage                                        Number of new cases:       7     Matched case 12
                                                                                               Matched case 12
Health:       450/450      Mana:          162/285             Number of new solutions:   3     Matched case 11
Damage:       21-27        Damage type:   Hero                                                 Matched case 11
Attack range: 300          Attack speed:  1.65224/2.13                                         Matched case 11
Armor type:   Hero         Armor:         3
```

Figure D.2: Execution of UPAI, later step

4. Step 1 and 2 is repeated every second until a player has lost all units, and the units controlled by UPAI receive new orders from the utility value system

every second. In the case seen in figure D.2, the battle has progressed, and the utility value system of UPAI has ordered the units to attack the enemy mountain king (unit 3). The current game state match case 12 of the case base, and the priorities of the best solution (solution 1) is used.



```
Name:         Archmage                                   Number of new cases:      12     Matched case 32
                                                                                          Matched case 32
Health:       450/450    Mana:           162/285         Number of new solutions:  7      Matched case 32
Damage:       21-27      Damage type:    Hero                                             Matched case 32
Attack range: 300        Attack speed:   1.65224/2.13                                     Matched case 32
Armor type:   Hero       Armor:          3
```

Figure D.3: Execution of UPAI, last step

5. As seen in figure D.3 player 2 lost all units, hence UPAI won the battle. The total reward value of the remaining units is calculated, and distributed to the solutions used according to figure 6.5. In this case, the surviving units are an archmage and a spellbreaker, which give a total reward value of 16. The battle is now finished, and ORTT automatically exits.