Learning to play Starcraft with Case-based Reasoning

Investigating issues in large-scale
case-based planning

Jan Eriksson
Dag Øyvind Tornes

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Helge Langseth, IDI
Co-supervisor: Anders Kofod-Petersen, IDI
Pauline Haddow, IDI

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

Jan Eriksson
Dag Øyvind Tornes

Learning to play Starcraft with Case-based Reasoning

Investigating issues in large-scale case-based planning

Master thesis, spring 2012

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science, Norwegian University of Science and Technology

Abstract

In this master thesis we describe our work in creating a planner for the real-time strategy game Starcraft using case-based reasoning. Our work has been focused on the challenges in creating a usable casebase, and the resulting issues arising from scaling up the casebase.

First, we present an agent designed to play Starcraft using plans from our CBR planner, and its architecture. We then move on to describe how this planner works, and how it overcomes the challenges in scaling up.

We then present several experiments designed to measure how well our approach works given the limitations we have set. Finally, we discuss our results, and provide some interesting unsolved challenges which may benefit from further investigation.

Sammendrag

I denne masteroppgaven beskriver vi arbeidet vårt med å lage en planleggingsalgoritme, basert på case-basert resonering, for strategispillet Starcraft. Arbeidet har fokusert på utfordringene i å konstruere en brukbar database av cases, og de utfordringene som følger av å skalere opp størrelsen på databasen.

Vi presenterer en agent som spiller Starcraft med planene fra algoritmen, og arkitekturen bak agenten. Deretter beskriver vi hvordan planleggingsalgoritmen virker, og hvordan den håndterer utfordringene oppskalering medfører.

Vi fortsetter med å beskrive flere eksperimenter som måler hvor godt løsningen vår virker, gitt de begrensningene vi har satt. Til slutt diskuterer vi resultatene, og staker ut mulige utvidelser til agenten.

Preface

This project was done by two master students at the Norwegian University of Science and Technology. During the course of our studies, we have selected Artificial Intelligence (AI) as our specialization, and have an interest in game development and use. Due to this, we were naturally drawn to this project because of its nature, the creation of AI agents for the game “Starcraft”. The project allowed us to choose fairly freely among different AI techniques for making agents, and this in turn motivated us further. Starcraft presents a challenge for AI agents, due to its complex environment.

Acknowledgments

First off, we would like to thank our thesis supervisors, Helge Langseth and Anders Kofod-Petersen, for their invaluable support in the work we have accomplished. We would also like to thank Agnar Aamodt and Tomasz Szczepański for sharing their expertise in case-based reasoning.

Jan would like to thank his wonderful fiancée, Silje Skage, for always being there for him.

Dag Øyvind would also like to thank ASS, BGB and FSK for keeping him motivated when the work seemed too overwhelming.

Jan Eriksson
Dag Øyvind Tornes

Trondheim, June 11, 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Overview

In this chapter we present an introduction of our thesis. First we present the relevant background for the thesis, followed by the goals we have for it. After that we present the possible contributions we can make, and the research method. Finally we give an overview of the remaining thesis structure.

1.1 The assignment

This is the assignment text as was agreed with our supervisor at the beginning of this project.

We wish to explore the possibility of using Case-based Reasoning in order to create plans that can be used by an agent playing Starcraft.

1.2 Background and Motivation

In this section we will go into the background of our project. First we will give a brief project description, followed by a short intro to Starcraft as an environment. Last we will show some existing AI work in Starcraft.

1.2.1 Project description

Prior to this master thesis we had done extensive research in architectures for an Real-time strategy (RTS) agent[22]. At the end of this project we had decided that our focus area should be Case-based Reasoning (CBR), and that we should implement an agent that uses CBR to perform planning. One thing we noted from our research was that the existing implementations did not elaborate on scaling issues, nor the creation of complete agents. We thus wanted to create a complete agent, and test how well it performed with regards to scaling. In order to achieve this, we set ourselves these tasks:

- Perform research on CBR tools and existing implementations
- Create a framework agent with minimal intelligence and features
- Create a CBR module, including modules to read and analyze replays
- Refine our agents other layers to play better
- Run tests to measure the performance of our CBR module

1.2.2 Real-time strategy games

Real-time strategy games is one of the most popular genres of computer games today. In a typical RTS two or more players build bases, train armies and research new technology in order to defeat their opponent on the battlefield. The success of the genre is in large part due to the possibility of playing online versus other humans. Most games also feature an artificial intelligence (AI) opponent for those who can not play against others. Unfortunately, these AI opponents are rarely robust enough to provide an entertaining challenge[3].

RTS games also provide a challenging testbed for AI researchers. Most RTS games feature complex environments and provide several challenges for AI methods to overcome. Russell and Norvig provide a classification of agent environments[18], and RTS games provide one of the most challenging combinations of environmental properties:

Partially observable environment
In RTS games most of the battlefield is covered by a “fog of war”, an area which cannot be observed. The agent will need to utilize its limited resources to scout the opponent or make decisions under uncertainty.

Stochastic
Most RTS games include an element of chance. The agent can not be certain that initiating an action will lead to a certain result.

Sequential decisions
In any RTS, decisions made now can have far-reaching consequences. Often,

a game is decided by a single early mistake made by one player.

Dynamic
: The real-time component of RTS games provides another challenge. They are fast-paced, and while the agent deliberates, the conditions it bases its decisions on may have changed significantly.

Continuous environment
: RTS games provide a continuous environment. An action does not immediately change the environment, rather, actions start a gradual change to the environment.

Cooperative and competitive multiagent
: Several agents can play simultaneously in a single game. Often their objective is to defeat all other agents, however, teams cooperating are not uncommon.

Together, these properties lead to a vast state-space, uncertainty and performance requirements, making RTS games an ideal test application for new AI approaches.

1.2.3 Starcraft

Starcraft is an RTS game released by Blizzard Entertainment in 1998. To date, it is one of the most popular computer games, having sold more than 11 million copies by 2009. Starcraft has its own pro-gaming league in South-Korea, USA and international tournaments. Starcraft, with its Brood War expansion, constitute the environment we use for testing. For readers unfamiliar with the concepts of Starcraft, please read appendix A.

Much of Starcraft's success is due to a break from the traditional way of creating RTS games. Prior to Starcraft, an RTS would feature two opposing factions, whose gameplay was nearly symmetrical. Starcraft changed this by introducing three factions (or races in Starcraft) with widely different play-styles. The three races are the insectoid swarms of the Zerg, the mechanized armies of the Terrans and the high-tech warriors of the Protoss. Each of these races have unique advantages and disadvantages, and all require different strategies to play effectively and win.

In Starcraft, two players face off with the ultimate goal of destroying the opposing player. In order to do so, the players must build bases, set up a resource harvesting operation and train armies to crush the enemy.

Figure 1.1 shows a Terran army defending against Protoss attackers.

Figure 1.1: **Starcraft**

The three races

As mentioned in the previous section, Starcraft features three races which the player may choose from. Each race plays in different ways, and have their own unique strategies.

The Terrans are human colonists struggling to survive in the vast space of the galaxy. Their armies combine infantry with armor, and are fairly powerful. The race provides strong defensive units, and many of their strategies rely on holding key locations on the battlefield.

The Protoss are an ancient people, with a strong hierarchical government. Their warriors rely on technology and honor on the battlefield. The Protoss strategies are based on fierce attacks and utilizing their technological advantage to defeat their foes.

Finally, the Zerg are a hivemind race and menace to all galactic civilizations. Their units are not individually strong, but numerous beyond imagining. The Zerg rely on their great numbers and fast evolution when entering battle.

Resource management

In Starcraft, the player must manage three distinct resources, namely minerals, gas and supply.

Minerals are the most common resource. Every unit, structure and upgrade in Starcraft requires at least a small amount of this resource. Minerals are harvested from mineral patches spread around the map, usually in clusters near good base locations.

Gas is the secondary resource in Starcraft. Much rarer than minerals, gas is a requirement of more powerful units and often used to research new technology. Gas can only be harvested from vespene geysers after the player has built a structure there to refine the volatile resource.

The last resource is supply. Each race has a unique way of providing this resource. Terrans have Supply Depots, Protoss have Pylons and the Zerg have Overlords. Before training a unit, the player must provide a certain amount of supply. Each unit has its own requirement, for instance Zealots require 2 supply, while Battlecruisers require 6. If a unit dies, the its required supply is available for another unit to use. If the supply sources are destroyed, units will not die, but the player is unable to train any new units before resupplying.

Bases

Building your bases is a large part of Starcraft. Different structures unlock new and more powerful units, allow you to research new technology, provide resources and defenses. Choosing the right structures to build can be crucial in winning games.

The act of creating additional bases is called “expanding”, and has a huge effect on the game. A secondary base will provide a much more powerful resource infrastructure and serves as a backup if your main base is attacked. Choosing the right time to expand can win or lose a game on its own.

Training and upgrading units

Creating a powerful army is the only way to win in Starcraft. In the most common mode of play, the “Melee”, a player wins when all of the opponents structures are razed. To achieve this, the player has several units at its disposal.

Some units are purely utilities, such as the Protoss Observer, which cannot attack, but is floating invisibly at the edge of space, and the drop-ships each race has available. Other units pack a punch. Generally, units are divided into three tiers, with the higher tiers being more powerful, but also more expensive. Some units fly, and can only be attacked by units with anti-air weapons. They are also not restricted by features of the map and can reach any positions. This grants them a special advantage, and some strategies in Starcraft revolve around this mobility.

In addition to training armies, the player can upgrade them. From special structures in the base, upgrades may be purchased. These are very expensive, but provide an advantage to all the players units. Some upgrades are called technology, and grant additional abilities to certain types of units.

Choosing the right mix of units for a given situation, and upgrading wisely is probably the single most important decision a player can make during a game.

Army management

Army management, more commonly referred to as “micro” in Starcraft, is the skill of controlling your units in an effective manner. Some examples may be retreating with wounded units, selecting targets which are susceptible to your army’s weapons and sneaking past your opponents defenses.

Professional players have shown time and again that a superior army can be defeated by an inferior force, if the smaller force is better managed. These pro-gamers often use the measure “actions per minute” (APM) as an indicator of their skill in micro. The more orders you can issue in a short amount of time, the better you are able to manage your troops. The most experienced players can reach APMs in excess of 400, nearly 7 orders issued every second.

Planning

Starcraft’s depth and complexity allows for a plethora of strategies. A successful player must be able to identify his opponents strategy, and devise a plan to counter this.

As the enemy is doing the exact same thing, continuously analysing the situation and revising the plan is an essential skill. Players facing a situation they have not planned for are easily defeated by the more prepared player.

Brood War API

Brood War API (BWAPI) is a free open-source C++ framework which gives us access to the same information a player has from the game itself. It also facilitates ease of implementing agents in Starcraft. We give a more detailed description of BWAPI in section 4.1.1.

Starcraft as an environment

Using Starcraft as the environment provides us with several advantages. First and foremost, Starcraft presents our agent with a highly dynamic environment, with hidden information and a multitude of available actions to perform at any point in the game. Starcraft has been played for more than 10 years, and in that time there has not been a single prevalent strategy that works every time. Due to this rock-paper-scissor effect of any strategy, the agent needs to continually ensure that it is using a strategy that beats the opponent in some way. In addition, the agent needs to be able to utilize its forces in a proper manner, since a plan is only as good as its execution. This provides us with an environment that can be just as hard as the real world, and as such it provides an excellent place to test new AI methods.

Second, due to the fact that the game is fairly old, it has very low computational requirements and can be run on almost any computer that exists today. This ensures that we have most of the computational resources available for the agents internal functions, and do not need to worry about Starcraft being a bottleneck in our execution.

Finally, we have access to the BWAPI framework, which speeds up the process of implementing our agent. By eliminating the need to create our own framework to interface with the game itself, we can start on the AI related work right away.

Using a single race

We have chosen to create an agent which only plays as Protoss. Since the three races are different, it requires a lot of additional work to create a generic agent which is capable of playing all the races. Creating a separate agent for each race is also out of the question, as this is a time-consuming process and would divert our focus from the research we wish to conduct. The choice of the race Protoss was made due to personal preferences from the group members. As we explained earlier, each race is balanced, so we do not get any special advantage or disadvantage by using Protoss.

1.2.4 AI work in real-time strategy games

In recent years Starcraft and similar RTS games has received a lot of attention from researchers. In this section we review some of the work which has been done.

Using Goal-Driven Autonomy in a Starcraft agent

Weber, Mateas and Jhala presents a complete agent in their article[25] capable of playing the game Starcraft autonomously. It does so by using Goal-Driven Autonomy (GDA). GDA is a research area in AI communities that aim to address the problem of how to respond when an agent encounters an unanticipated failure. One of the main ways to perform GDA is to enable agents to reason about their goals, enabling them to react and adapt to unanticipated situations. The agent was tested against both the built-in AI and human players online. It managed to outperform 48 percent of the human players on that specific ladder, and was capable of changing its strategies when playing against the same human player.

Creating a CBR database from automated replay annotation in Starcraft

Weber and Ontañón presents a system in their article[27] that focuses on automating the process of learning from expert Starcraft replays. They state that previous work has been using small amount of replays due to the effort needed to convert them to cases manually. Their work seek to remove this barrier by the use of goal ontologies. They base their work on a system called Darmok 2[16], which allows them to find plans from human demonstrations. Their experimental evaluation manages to use plans to set up the resource infrastructure, but is still unable to beat the built in AI of Starcraft.

Using Reactive Planning Idioms for multi-scale game AI in Starcraft

Weber et. al. presents a system in their article[26] that focuses on creating an AI agent that is able to reason about goals across multiple levels of granularity. They propose the need for this technique due to the nature of RTS games. In RTS games, one needs to reason about high level decisions such as long term goals and strategies while simultaneously managing short term goals for an army engaging the opponent. The usual way to deal with this is to abstract the responsibilities into separate layers, but they argue the need for reasoning across the layers. One example is when a unit both has individual priorities and responsibilities to the squad it is part of. Their system uses the reactive planning language A Behaviour Language (ABL)[14], and successfully implements a behaviour tree that is able to pursue concurrent goals. They show good results against the built-in AI, with a win rate of above 60 percent against all races.

Using reinforcement learning to create game AI in Starcraft

Micić, Arnarsson and Jónsson presents methods for reinforcement learning in their research report[15] with Starcraft as their domain. They focus on the area of managing troops (micromanagement), both as individuals and in squads. Their experiments are for learning when to transition between states in Finite State Machines (FSM), and the effectiveness of using reinforcement learning for tuning this transition. Their results show positive results with single units and small squads, but it gets harder as the complexity of the squads increase. They argue that reinforcement learning can be applied with more advanced techniques than FSM, but also show that learning needs to be done offline. They state that any noticeable learning during execution is still out of the question.

1.3 Goals and Research Questions

In this section we outline the goals we wish to achieve with this thesis. We present two issues which have received little attention from researchers, and how we intend to solve these.

We also present several questions which serve as a basis for testing and as a measurement of how successful our approach has been.

1.3.1 Case construction

A lot of work has been done in applying the CBR process to the domain of RTSs, especially as a tool for generating plans. However, one aspect of this, namely that of constructing a useful initial casebase, has only been touched upon briefly. Most Case-based planners for RTS games utilize small casebases which have been created through a manual process.

A casebase must have several properties to be considered useable:

- The casebase must contain enough cases to provide a solution even in rare situations the agent may face.
- The casebase must be created with minimal manual effort by the user.
- The casebase must have a representation which is suitable for use in a real-time environment.

The first item, casebase size, is an important one. Most research has been performed with manually constructed casebases, whose small number of cases can't cover all the situations an agent may find itself in.

The effort involved in creating the casebase should be minimal. Hand-creating cases is time-consuming and will likely not result in a good casebase. Thus, a casebase whose size is significant must be created by some automated process.

Finally, large casebases will come with a performance penalty. It serves us no good if searching the stored cases takes minutes or hours, by the time we have a solution it will be outdated. Therefore, the casebase must be represented in a manner which makes efficient lookup possible.

In order to combat these issues we propose a method for automatically extracting cases from expert replays, and investigate whether this is a viable approach to case construction.

1.3.2 Scaling issues

Given that our case construction method is successful we face a second issue. The casebase will contain a very large amount of cases, which will lead to decreased performance. We will therefore investigate issues in scaling up casebase size.

First, we need to know whether a large casebase causes performance issues which make the CBR planner useless in a real-time setting. New plans must be generated quickly if the agent is to respond effectively to the rapidly changing situations in the game.

Second, we will investigate how a larger casebase affect the outcome of games. We will determine whether the larger casebase leads to an agent which wins more games. If this is not the case, we wish to determine the cause.

Finally, if runtime performance turns out to be and issue, we will investigate possible solutions to this.

1.3.3 Research Questions

- What are the major challenges in creating a usable casebase
- What effect, if any, does the casebase have on the agents performance
- Does the size of a usable casebase require a sophisticated selection algorithm to meet the demands of a real-time environment

1.4 Contributions

This thesis will research the feasibility of creating a CBR planner for an agent playing Starcraft. Our main contribution is determining if it is possible to create a case-base from a large amount of replays. Previous work concerned with analysing player strategies in Starcraft[23] used 36 replay trace files to create the case-base, and got around 1500 cases. We wish to go further than this and find the feasibility of using a case-base with more than 10,000 cases. Our tests will investigate whether the performance of an agent improves with the additional cases.

A second contribution is that we intend to create a complete agent playing Starcraft. It is possible for other parties to create an agent based on our work. One possibility is to keep the planning with CBR we use and create an independent micromanagement module that uses our plans. Our selected architecture support switching out parts within the layers without affecting the overall run-time of the agent.

1.5 Research Method

Initially we will research existing solutions that uses CBR and tools used to create these. Our goal is to develop a good understanding of the feasibility of using CBR in Starcraft. We will also find out how previous solutions have solved issues with the size of the case-base, if they have considered it at all.

In order to answer the questions we have posed, we will implement an agent which plays complete games of Starcraft. By complete games, we mean that the agent will have to play games in the same manner humans do, and not in a restricted scenario. Thus, the agent must be capable of managing both resources and units, planning ahead and constructing bases.

In order to create a casebase to be used as input for the planner, we will need to create a system that allows us to translate expert replays into a format that is understandable by the agent. We will focus on time-efficiency, as the case representation could be changed at any point. This is to avoid having to spend too much time recreating the case-base when these inevitable changes occur.

In order to test our agent we will let it play against the built-in AI and use the data from these matches to reason about its performance. We wish to focus on how usable the case-base is in terms of time efficiency and the amount of wins/losses the agent manages to get.

1.6 Thesis Structure

The remainder of this thesis is structured as follows:

In Chapter 2 we present the research we performed during August through December 2011. This project led to this master thesis and is of importance for some of the choices we made. We follow with a thorough description of the CBR method, and give some examples of previous work in the field.

In Chapter 3 we present the CBR model we are using in our agent. We start with detailing the model of the domain and follow with the steps our CBR follows.

In Chapter 4 we present an overview of our system. First we present the tools we have utilized, and how they have affected us when we were creating the agent. We follow with a description of the major components of of the agent.

In Chapter 5 we refine our research questions to testable experiments, and define performance measures for the experiments. We present the method by which the experiments were conducted and finally the results we observed.

In Chapter 6 we present an evaluation of our research. We summarize our results and conclude with some possible extensions of our research.

Chapter 2

Background

Here we will provide an overview of the background of our thesis. First we will present the research we did during the fall of 2011, which is considered as a pre-study to this thesis. After that we will present a thorough overview of the case-based reasoning method.

2.1 Real-time strategy agent architectures

Prior to this master thesis we did a research project in fall 2011. The main goal of this project was to investigate the possible architectures that are suited for an RTS agent. We were then tasked with arguing which of these were most suited for an agent and instantiate an agent design based on this architecture. Finally we were to select an area of specialisation, which would be realized in this master thesis. This section will give a short summary of that project. For the complete project report, please read[22].

2.1.1 Pre-study

The project had six groups of two students, and we were encouraged by our supervisors to work together on a pre-study. We utilized a method called Structured Literature Review (SLR) to find a collection of research papers to base our arguments on. SLR is a method for a thorough and structured search for literature relevant to one or more research questions. The goals of this method are to ensure the quality and coverage of the literature related to some given research question(s), and to be reproducible. The interested reader can read Kofod-Petersen's tutorial[9], which gives a step by step recipe of the method.

At the end of the review we had narrowed the scope down to about 50 relevant articles. These served as a basis for the remainder of the projects investigation of possible architectures for an RTS agent.

2.1.2 Architectures

Our research showed that there are many different architectures that could be used in an RTS agent. We classified these in three main classes of possible architectures: Layered, Cognitive and Multi-agent systems.

Layered architectures

Layered architectures are a staple within software engineering, with applications in many sub-fields, including AI. While layered architectures may encompass many concepts, there are some features which uniquely identify them. First and foremost, layered architectures consist of several layers, where each layer has some clearly defined responsibilities. An example would be separating a planning layer from a reactive layer. Another feature of layered architectures is that any layer should only communicate with the layers directly above or below it.

Cognitive architectures

Cognitive architectures are based on the way the human brain works, by using a large amount of multiple processors that compete and collaborate with each other. The idea is that conscious content will emerge from this interaction, and create advanced behaviours. Due to this, there is some overlap with biologically inspired AI research. As it is inspired by the brain, many architectures are based on the different parts of the brain that are used in human reasoning. Examples include the basal ganglia, which is used by humans for a variety of functions such as voluntary motor control, cognitive emotional functions and procedural learning relating to routine behaviours or "habits", such as eye movements.

Multi-agent systems

Multi-agent systems (MAS) is a fairly new field which has attracted a lot of attention in the past years. The basic idea is to make several agents which control one or more entities each, by analysing sensor input data and control them via their actuators. The agents also need to communicate with the other agents and coordinate plans together.

2.1.3 Architectural reasoning

After analysing each architecture class, we ended up creating an architecture based on a layered one from robotics[2]. Our research showed that all three architectures satisfied our goals of planning, resource management and unit control. The main difference was in the inherent qualities of each architecture. We needed the architecture to be modular and extensible. While both cognitive architectures and MAS are modular and extensible, they are not inherently simple. Both fields requires time and work to understand the principles to it, and the architectures are often geared towards the specific methods each field employs. Layered architectures have simple communication protocols between the layers, while supporting an arbitrary complexity within each layer. This means that it is possible to use any method or technique to fulfill a layers responsibilities, without the need for any other layer to know of this complexity. This is the well-known principle of encapsulation, and it enables work to progress in parallel in the different layers.

2.1.4 Our architecture

As is presented in[2], the architecture is divided into three separate layers: planning, executive and reactive. In addition, the architecture has a separate module which makes global state available, in a read-only manner, to all layers. We also use a communication-module, which allows layers to cooperate, while still maintaining strict separation. We will explain briefly how each layer works here. Figure 2.1 shows the overall architecture we are using.

Planning layer

The planning layer is the top layer and the brain of the agent. It performs the high-level reasoning that is required in the RTS domain. This includes generating plans, distributing plans and fixing outdated plans. The plan generation can be

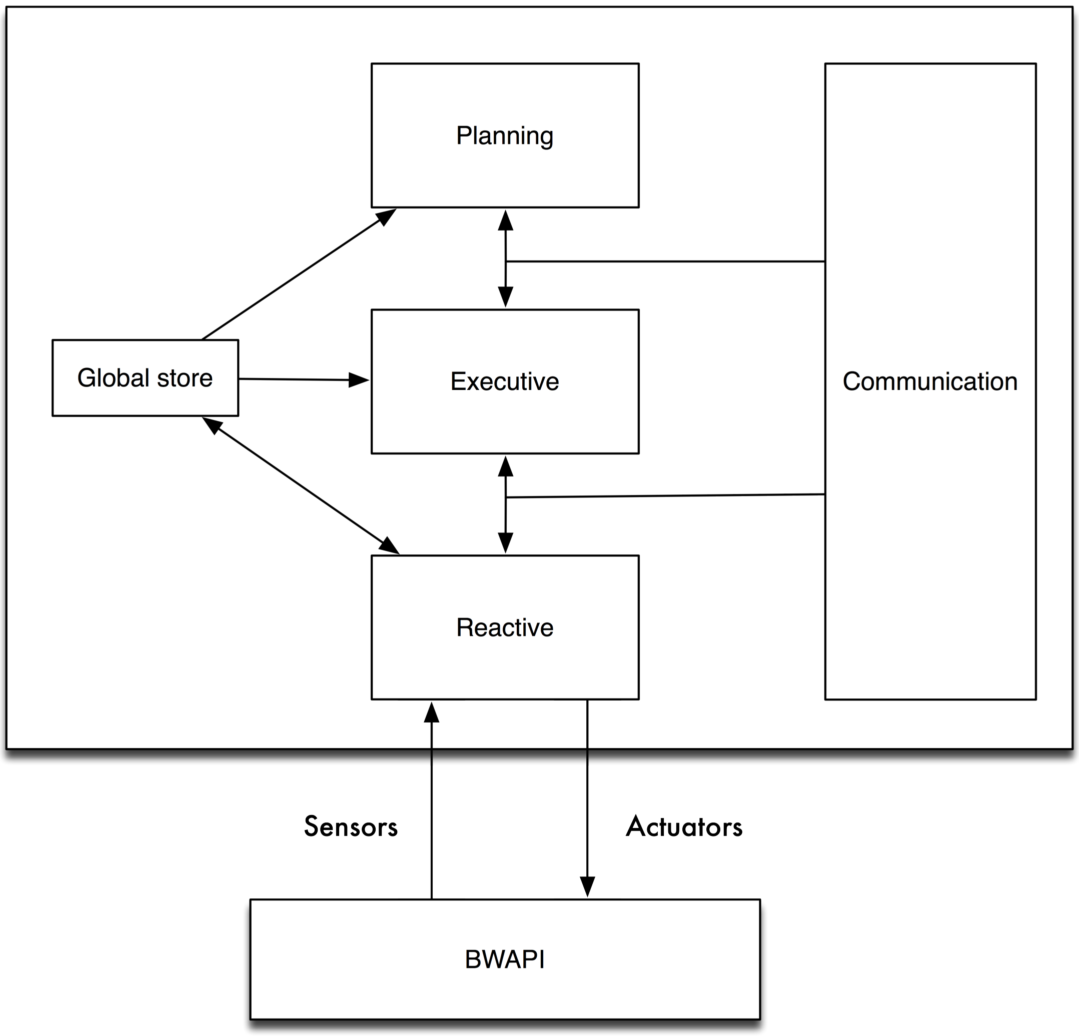

Figure 2.1: **Our layered agent architecture**

performed by any arbitrary method, as long as the generated plans comply with a given format.

Executive layer

The executive layer is the middle layer. It acts as an intermediary between high-level reasoning and executable tasks. Its tasks include plan decomposition, allocating resources to tasks, maintaining scouting information and keeping track of units and structures. It will give orders to the reactive layer such as ordering attacks, creation of buildings and units, and ordering scouts. It will also notify the planning layer if a plan seems impossible to execute, or if events should render the plan obsolete.

Reactive layer

The reactive layer is the bottom layer, and the only one that interacts directly with Starcraft. It translates tasks such as attacking and creating buildings into actions that units perform in the game. This is managed by creating behaviours that control individuals or a set of units. Behaviours are designed to be independent of each other, and can be linked together to perform more complex tasks. One example is that we can order units to move together to a rally point before attacking, in order to attack with a single large force, rather than two small ones. This does not affect the move behaviour or the attacking behaviour, as they do not know or take into account any such abstractions.

2.2 Case-based reasoning

Case-based reasoning is a subfield of machine learning which is concerned with learning from experience. CBR is based on the intuition that problem solving can be performed by remembering similar situations form the past and then reusing the solution from that situation in the new context. Ontañón et. al.[17] presents a CBR system that integrates learning and reasoning in this way.

A more intuitive way of explaining CBR is through examples from the real world. For instance, a lawyer arguing a case may remember a previous case, where the situations were similar, and reuse the reasoning from the previous case. Another example may be a doctor observing a certain set of symptoms in a patient. If he has seen similar symptoms in the past, he may reason that the diagnosis is the same and recommend a treatment. In both examples previous knowledge has been employed to create a solution to the current problem at hand.

CBR traces its roots to the work of Schank on Dynamic Memories[20], and the research on analogical reasoning by Gentner[6]. The first system that uses the ideas of CBR is the work done by Janet Kolodner on the system Cyrus[11] [12]. Cyrus contains two databases that are able to give information about the former U.S. Secretaries of State Cyrus Vance and Edmund Muskie. Cyrus is able to answer questions given in English that concerns these two secretaries. An example is given in her first article[11]:

> *It retrieves facts from its memory when queried in English. Following is a dialog with CYRUS:*
>
> - *When was the last time Vance was in Egypt?*
> - ON DECEMBER 10, 1978.
> - *Why did he go there?*
> - TO NEGOTIATE THE CAMP DAVID ACCORDS.
> - *Who did he talk to there?*
> - WITH ANWAR SADAT.
> - *Where was Muskie three weeks ago?*
> - IN EUROPE.
> - *Who did he talk to?*
> - TO NATO IN BRUSSELS ON MAY 14 AND TO ANDREI GROMKYO IN VIENNA.

2.2.1 Case representation

Representing cases is a very fundamental part of any system performing CBR, since the CBR module is only as good as the cases it stores. But before we explore the case structure we plan to use, let us define the meaning of a case. Janet Kolodner is one of the pioneers within CBR, and in the first chapter of her book Case-based reasoning[10] she defines a case as:

> *A case is a contexualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner.*

She also divides a case into two parts:

1. *The lesson(s) it teaches*
2. *The context in which it teaches its lesson(s)*

The first item is the cases *solution*, and the second item are its *features*. Based on this we need to first find the features of Starcraft's domain that have an impact on how we should play in order to win. Second, we need to define how our solution should be structured so our agent can carry out the actions it needs to fulfill its goals.

2.2.2 The CBR method

Here we will go into detail about the case-based reasoning method. While there are many different implementations, which we will show in section 2.3, they share the general idea of a CBR cycle. We will present the cycle structure that Aamodt and Plaza[1] proposes. It contains 4 steps:

- Retrieve
- Reuse
- Revise
- Retain

2.2.3 CBR Cycle - Retrieve

The "Retrieve" step is the first part of the CBR cycle. It can be divided into 4 subtasks:

- Identify features
- Search
- Initially match
- Case Selection

Identifying the features can be simply comparing the input descriptors of a domain. In our domain this means that we use the current gamestate in a gameplay. This gamestate contains information such as ours and enemy units. It is possible to describe the domain in more detail, by using contextual information. One example is to signify that an attack is currently being executed, which would mean that some of our units are occupied. Another example would be to use the opponent model from our previous case as input for buildings we believe the enemy has, if we have not been able to scout them. This approach is an important part of any knowledge intensive CBR system.

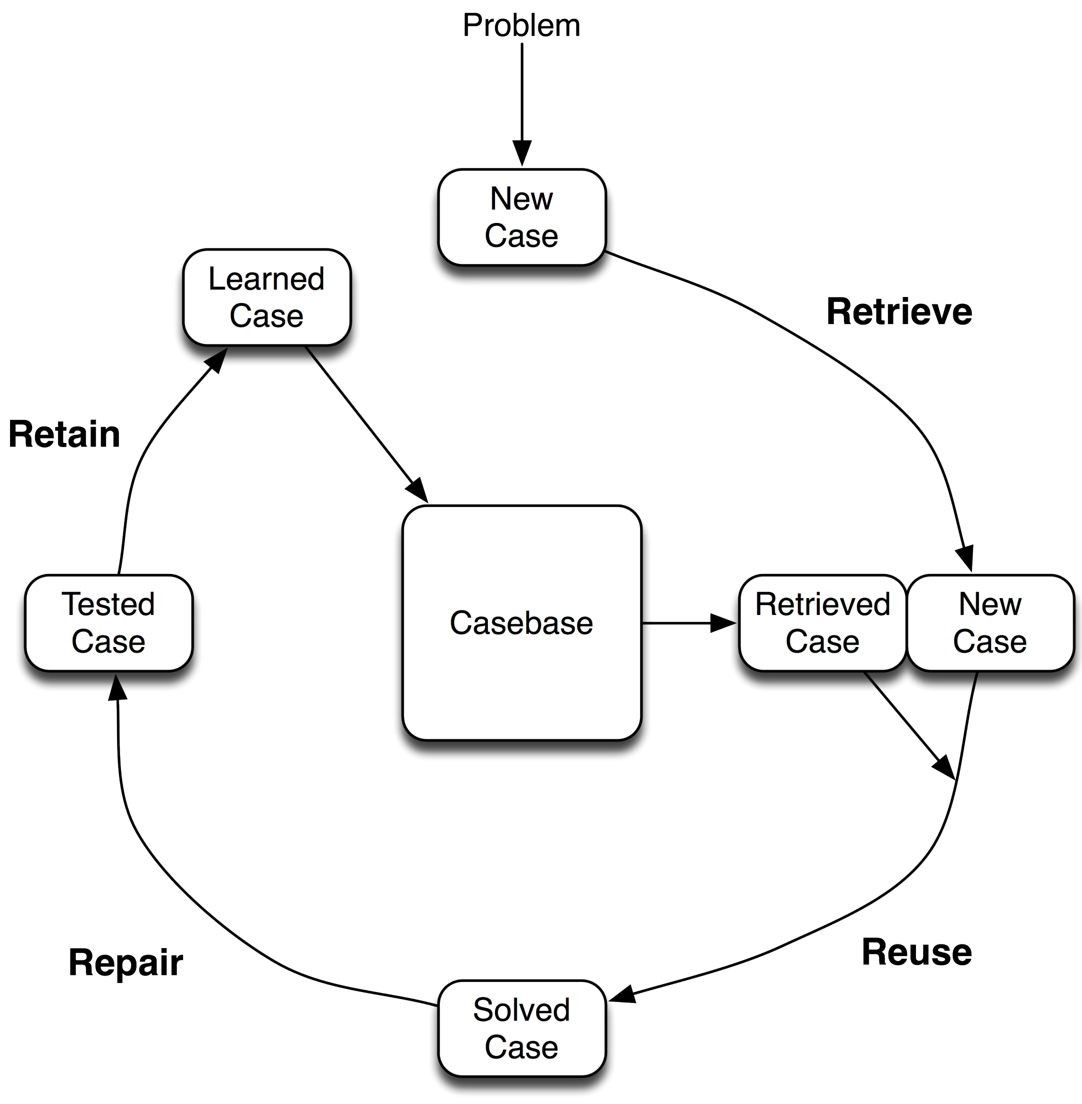

Figure 2.2: **The CBR cycle**

Searching a database for matching cases is the next task. The first objective is to find out what we are looking for. One example is that we require a case to use, and have a certain model of the world. This example will search for cases based on the features they have. Another example is that we are following a case, and want to find similar cases. This would require the search to base its comparisons on both the features and the solutions of cases. As for the searching itself, it may not always be feasible to search through all the cases we have. Some domains have strict time limits for the retrieval process. In others the case database could be very large, depending on the design and data availability of the domain. Efficiency can, and often is an important part of the search. Within the CBR community this is collectively known as the indexing problem[10]. One way to improve the situation is to provide search trees which will filter out uninteresting cases. Another way is to combine similar cases into more general cases. One way of creating general cases is proposed by Koton[13]. He creates *Generalized Episodes* (GE) which are used to combine cases with some similarities. The GE contains *norms*, which contains abstract general information that is common for all the containing cases. Next follows *indices* which are features that differ in some way. The *indices* can lead to the third part which can be *cases* or another GE.

Initially match is the task of finding a set of possible matches. There are a few choices of matching criteria, depending on the domain and database size. The simplest method is to do a similarity matching based on the features of a case. This provides a fast and easy way to get a certain similarity value between a current situation and the cases in the database. A more knowledge intensive way is to weigh each feature according to its importance. The importance of a feature depends the state of the environment, and a knowledge intensive approach utilizes this. An example is in Starcraft, where buildings can show strategies early in the game, while later we are more interested in amount of income bases. Weighting allows partial matching, where the focus is on a subset of the features which are more relevant to the current problem we need to solve. Another way is to consider contextual information, and find a deeper semantic similarity between a case and a current situation. This can be done by using goals and constraints to guide the matching process. The knowledge intensive methods are in general more time consuming, and the environment might be too time constrained to use them.

Case selection is the process of further elaborating on the cases it received from the initial matching. This may have been done in the initial matching already, by finding only one case to use, but more often we will have a set to consider. In order to select a case it will need to evaluate the matching cases in more detail. For instance, this can be done by generating explanations for why some features does not match fully and finding the consequences of the mismatch. Another way is to simulate a solution and see how well it performed. In our domain this is not feasible, as it would be too time consuming to simulate entire plans. A system can also request user input for selection.

2.2.4 CBR Cycle - Reuse

The “Reuse” step is the second part of the CBR cycle. It can be performed by copying or adapting a solution.

The simplest method is to copy the solution. Any differences are abstracted away, and the solution is used directly. This is the most trivial form of reuse, and is naturally not possible in most situations. If the differences are critical, one needs to adapt the solution in some way before reusing it.

Case adaptation can be performed in several ways. According to Aamodt and Plaza, one can either reuse the solution itself (transformational reuse) or the method that constructed the solution (derivational reuse). In transformational reuse, there exists a way to transform the current solution into a solution that we can use in our current environment. In our domain, this could be by adding technology buildings that are prerequisites for certain parts of the solution, or removing redundant items from the solution. In derivational reuse, one will construct a new solution based on the method that we used in the retrieved case. With the new domain parameters and goals it will “replay” the old solution in the new context, and thus generate a new solution.

2.2.5 CBR Cycle - Revise

The “Revise” step is the third part of the CBR cycle. This step is performed after a copied or adapted solution has been used. If the solution failed in some way during reuse, we have a chance to learn from those errors. Revision consists of two subtasks: evaluating the solution and repairing faults.

Evaluating the solution is generally time consuming and is performed outside the CBR module. It is done by performing the solution in a real environment or asking a teacher to evaluate. If the solution is successful, it will be retained. If not, we will progress to repairing the solution.

Repairing involves detecting errors in the solution and generating explanations for them. The first way to repair is by using the case database to find another solution to a sub-problem that failed. If our database is rich enough, it will be possible to find proper solutions for the problems we encountered, and creating a new case by merging the solutions. The second way is to modify the solution so that the failures do no occur. In the CHEF system[7] [8], the repair module adds steps to the plan to assure that the causes of the errors will not occur. The repair module needs to have certain domain knowledge that allows it to compensate and fix errors that are commonly occurring.

2.2.6 CBR Cycle - Retain

The “Retain” step is the fourth and last step of the CBR cycle. It is the process of deciding what parts of a case should be stored in the database, and how we should store it. It has three parts, extraction, indexing and integration.

The extraction step involves finding out which parts of a case should be retained. If we have a failure, we want to make sure we do not make the same mistakes again. This can be done by having a failure database we can use to check if a solution has pitfalls we want to avoid. It also applies to successes, as we need to decide if the entire case is relevant for the success.

Indexing is the process of deciding how to structure the database for future use and deciding what features to use. The trivial solution is to use all the features in this process. This is, however, not optimal if we have a large database or if time is restricted.

The integration step involves either inserting the newly generated case in the database, or updating an existing case with the modifications we have made during reuse. If we have a new case we want to add we need to modify the indexes in the database, to improve similarity matching in the future. If we need to update an existing case it comes down to how we wish to alter the existing cases. This is usually done by combining the features and updating the weighting.

The retain step is a good source of learning, as we will gradually refine the database to be better at matching and solving new problems.

2.3 CBR in Starcraft and other Real-time strategy games

In this section we will present various work on CBR systems in Starcraft and other real-time strategy games.

2.3.1 Creating a casebase from automated replay annotation in Starcraft

Weber and Ontañón presents a system in their article[27] that focuses on automating the process of learning from expert Starcraft replays. They state that previous work has been using small amount of replays due to the effort needed to convert them to cases manually. Their work attempts to remove this barrier by the use of goal ontologies.

They base their work on an existing system called Darmok 2[16]. This system can analyze human demonstrations (traces) and create plans from these. The system uses a special technique called *on-line case-based planning cycle*[17], proposed by Ontañón et. al. This technique interleaves planning and execution in real-time domains. In addition it uses an efficient plan transformation algorithm to adapt plans in real-time.

Their system is able to automatically annotate replays with goals from a goal ontology, in order to create traces that can be used in Darmok 2. The problem with this process is that general replays are very noisy, unless you specifically tailor an expert to play with the agent in mind. The goal ontology is used here to specify when certain goals are being pursued. Goals can be subdivided into smaller task oriented goals, and each goal has parameters that state when this goal can be pursued in an ongoing match.

When evaluating the system it was noted that it was not able to beat the built in AI. The agent was, however, able to set up a resource infrastructure and expand the technology tree. The reason for the failure against the built in computer was because it uses a strong early attack strategy which their system is not yet able to counter properly. A full evaluation of the system is intended as part of the future work.

This system is very relevant to our agent. Our system was inspired by the use of automatic replay analysis and creating plans from these. Their system touches upon our focus area, scaling CBR. As they expressed, creating a large case database is not possible with manual replay analysis. It is our intention to go test the CBR module with an even larger case library than they used in their system.

2.3.2 CBR used for micromanagement in Warcraft 3

Tomasz Szczepański and Agnar Aamodt presents a system in their article[21] for managing troops (micromanagement) in the strategy game Warcraft 3. This game is made by the the same company that made Starcraft, Blizzard Entertainment, and it shares some game concepts with that game.

They have chosen Warcraft 3 as their testing environment because it has a large emphasis on individual unit control. Compared to Starcraft, units in Warcraft 3 have much larger health, and often an inherent ability to heal themselves. This leads to strategies involving moving damaged units out of the fray and spreading the damage to several units. In addition the built in AI is fairly predictable, it has unit type priorities, and the observant player can make a note of which units it will focus on and simply move it back. This will cause the AI to follow that unit mindlessly while taking numerous hits from the lower priority units.

The system was trained by letting it play against the computer, and having an expert add cases when it did something wrong. After training the module, it was able to beat the built in computer every time. Additional tests were performed on human players of various levels of skill. It was shown that it outperformed the built in AI, but was beaten by the expert players in the end. They did not, however, manage to do so without loosing units. In addition the agent was used as a combat assistant with the human players. It worked by taking control of any unselected troops, while the human player controlled whichever units he had selected. This received positive feedbacks from the novice and casual gamers, but the expert players felt it was getting in the way of their plans.

While this system is not directly relevant to our current system, since they deal with the low level micromanagement instead of planning, it is still relevant to our future work. Their article shows that micromanagement using CBR is a very viable approach in real-time strategy games. In our system, it is a natural way to expand the use of CBR, by letting it learn how to control armies from replays. This work is, however, more than likely to be considered a separate master project in itself.

2.3.3 CBR using conceptual neighbourhoods in Wargus

Weber and Mateas presents a system in their article[23] which uses conceptual neighbourhoods to perform retrieval in a CBR system. Conceptual neighbourhoods are a hybrid between nearest neighbour methods and symbolic CBR[24], which use domain specific transformation rules to improve case comparison. They use a game called Wargus, which is a clone of the game Warcraft 2 by Blizzard Entertainment, to test their agent in. The CBR system is tasked with finding optimal build orders and executing them.

The system is based on A behaviour language (ABL)[14], which is a reactive planning language. The agent is a bundle of managers that each have certain responsibilities. These managers communicate with each other through the ABL's working memory.

The casebase was created by running several scripted builds against each other on several maps. The scripts were hand coded build orders with specific timing attacks to be executed. The map types ranged from small maps with open paths to large maps with bases separated with a forest that needed to be cut down.

The agent was tested against the built in AI and other well established scripts. An important feature in the testing stage was that they used both perfect and imperfect map information. This had not previously been tested to a great extent. The testing was done by comparing a nearest neighbour selector (NNS) to the conceptual neighbourhood selector (CNS). The CNS algorithm performed just as well as the NNS algorithm with perfect map information, but it performed better than the NNS when imperfect map information was used.

Our reactive layer was inspired by the use of ABL in this system. While we did not go to the extent of using ABL, we implemented our own version of it. Our reactive layer consists of independent behaviours that can perform anything from simple to advanced tasks. The nature of these behaviours make them very useful for expanding the reactive layer at our own pace. Adding behaviours is simply the operation of instantiating the behaviour and attaching it to the root node. Behaviours can also be tied together, without the knowledge of the subsequent behaviours, to perform more advanced tasks together.

Chapter 3

Planning with CBR

In this chapter we describe our research in CBR-planning. We also present our algorithms and our chosen model of the domain.

3.1 Modelling Starcraft

Here we describe the structure of our cases. We explain the level of abstraction we have chosen, and why we believe it is appropriate. We also explain the effect these choices have on other parts of the agent.

Figure 3.1 shows how we intend to structure our cases in a compact manner. Features are represented as Name : Type, where [] is a list and () is a tuple. Plan and opponent model are lists of actions to perform and expected actions respectively. A more detailed explanation of this is given in the following sections.

3.1.1 Case features

Deciding the features of a domain is an important task. Cheng and Thawonmas[4] presents a case structure for the RTS game "Warcraft". They made the decision to divide the features into three levels, Strategic, Tactical and Operational. Strategic and Tactical can be considered the description of the domain, while Operational corresponds to the tasks units can get. We based the decision on our own knowledge of Starcraft's domain, we have chosen the following data from Starcraft to be our features:

- Ours and opponents race

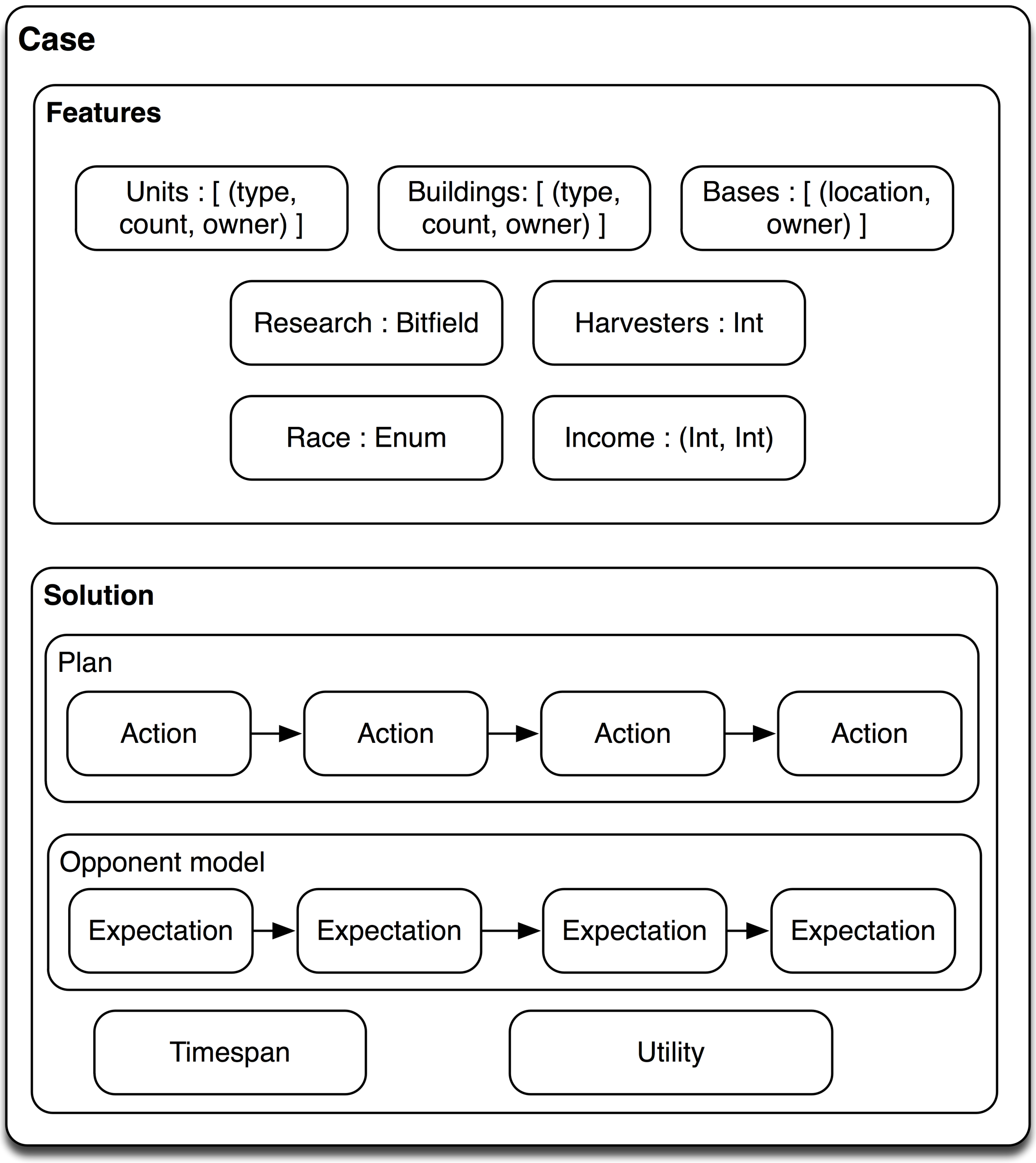

Figure 3.1: **Representation of a case**

- Ours and opponents military units
- Ours and opponents buildings
- Ours and opponents research
- Resources and income rate
- Ours and opponents harvesters
- Ours and opponents bases

Below follows a short explanation of the impact each feature has on how we need to plan our strategy:

- The race of the our agent and the opponent is in important factor in matching. In general, we wish to select amongst cases where the races match what we observe, since the available units and strategies for each race is varied. A note here is that we have trained the agent focusing on replays with at least one Protoss player, since our agent always plays the Protoss race. The variation is in the enemy race. There can be instances where the agent has similar cases for different enemy races, which can be used to determine the agents strategy. One example is if the enemy is focusing on close-combat units, which requires the same counter strategies by our agent. The problem is that we still face the possibility that the opponent switches to a race specific strategy that is hard to plan for with the wrong enemy race. Due to this we will only consider cases that matches the enemy race.
- The military units of the agent and the opponent is vital for the available strategies. If the opponent already has a strong force, while our agent has focused on economy, it would be hard to facilitate strategies which involve harassing the enemy to keep control of his movements. We anticipate that many cases will assume that the agent has a certain amount of units, and strategies can involve making a complementing unit that will make a good attack force.
- The current buildings we and the opponent have available can be important in any match. Some strategies need many of the same production building in order to build a certain amount of units within a certain time-frame. It also shows what units that are possible to build for ourselves and our opponent, which can have an impact on available strategies.
- The research that we have available is also important for strategies. While it can be hard to find out the opponents current research, beyond the weapon and armor upgrades we have available from units, it is still important to

calculate them. One example is the increased speed of "Zealots", which will allow them to catch up with ranged units that are trying to run away. Another example is similar troops with different weapon and armor upgrades. For instance, a "Marine" that has its weapon and armor upgraded once which means it will require 6 shots to kill a basic marine. The basic marine will require 8 shots to kill the upgraded marine. In a more extreme situation, a basic marine against a fully upgraded marine would play out completely differently. The fully upgraded marine requires 5 shots to kill the basic marine, while the basic marine requires 14 shots to kill the upgraded one. Either way, there is a major difference if a similar amount of troops meet each other in battle.

- The agents current resources and income rate is important in order to complete a case in its allocated timeframe. In the extreme case, an agent cannot execute a plan without gas, and if he has no possibility to create an income of gas (his vespene geyser has been blocked by the opponent), the current plan becomes unusable. In other situations, an agent could have a sudden drop in income, which causes him to be unable to complete the plan within a certain error margin of the allocated time. This could have a major impact if the agent wanted to make an early attack, but was delayed.
- The amount of harvesters you and your opponent have available can prove important. One example is that our agent can calculate the expected income rate of the opponent, and use this information to plan how many harvester he needs to destroy in order to cause a significant delay in the current expected build order he thinks the opponent will use.
- While the amount of bases is not vital to the agents success, it is important in some situations. It is coupled to the expected income an agent can possibly support, and running out of minerals at one base location can prompt the agent to move on to other areas. Another example is if the opponent decides to expand to obtain a better income, the agent can plan to attack him while he has committed himself to economy, or use the opportunity to expand himself.

3.1.2 Solution

The second part of the case is the solution. We have chosen these components to be part of the solution:

- A list of actions to be executed by the agent

- The timespan of the case, measured in frames
- The opponent model, which includes the units and buildings we expect him to build
- Utility, given by a win/loss ratio

Below follows a more detailed explanation of each part of a solution:

- The list of actions is the actual base for the plan we wish to execute. It will contain high-level actions like build certain buildings, train a certain amount of units, research different technologies or expand to an additional income slot. It also contains attack and defend commands, with different forms of granularity. Examples include harass, all-in and drop. The agent will decompose the actions into commands that can be sent into the game world, so decisions on building placement, squad movement and attack strategies beyond the high level ones are up to the actual implementation to figure out. This separates the responsibility and ensures that the module can be changed without major impact to the agent.
- The timespan that the case has is important if the agent is to make timed attacks, or assume that the opponent can only build a certain amount of buildings and/or units while the plan is executing. It is also a good measure for how well the agent is performing, if he manages to execute a plan in less time than originally thought for instance.
- The opponent model is useful to the agent. If the opponent does exactly what we expect that he will, then there is no reason that the agent should not execute a plan as it was previously done, save if he manages an attack or defence more poorly than it was in the case. It also provides the agent with "things to look for" when scouting the opponent. If our plan shows that the opponent should have three production buildings by that time, and he only has one, then we need to figure out where he is putting his resources. On the other hand, if the agent finds a different building than he expected, that piece of information may require the agent to use another plan. This provides us with a natural way of replanning, without wasting computation time on replanning after a certain amount of time.
- The utility of a case is calculated from the ratio of the wins and losses of the cases this case can lead to. The value is used to decide the probability that we will select this case if it has been matched. To show that we have won many times with a higher utility is an important measure when selecting cases.

3.1.3 Composite cases

According to Aamodt and Plaza[1], a case is a specialised episode in the library, containing specific domain knowledge. An important concept in CBR is Memory Organisation Packets (MOPs). Proposed by Schank[20], a MOP is a general knowledge structure to account for the diverse knowledge contained in episodic events. A good example is the one given by Schank: "The sand dollars and the drunk".

> *X's daughter was diving for sand dollars. X pointed out where there were a great many sand dollars, but X's daughter continued to dive where she was. X asked why. She said that the water was shallower where she was diving. This reminded X of the joke about the drunk who was searching for his ring under the lamppost because the light was better there even though he had lost the ring elsewhere.*

As we can see from the example, two cases from different domains can have similar solutions. In the example the solution for both cases would be to search nearer the place where they believed they could find the item they were looking for, instead of looking where it was easier to search. In our domain, however, we have the opposite problem. We have many cases that can have similar features, but different solutions. While it is certainly possible to find all the cases that match well, and then choose, it would be simpler if cases were bundled together. We therefore propose, based on Schank's theory, the use of composite cases. A composite case is a case that is constructed from very similar cases. By combining two or more cases into a general case, we will have a faster database, but chiefly we will have several solutions with different utility. As we can see from figure 3.2, this will give us a decision tree when we find a case that matches with our gamestate. The decision now rests on how well a solution has performed, with several to choose from.

3.1.4 Sample case

Figure 3.3 shows a simple case of the initial actions our agent will perform, as well as the indices as they were recorded from a replay. In the figure, A means Agent and E is the Enemy.

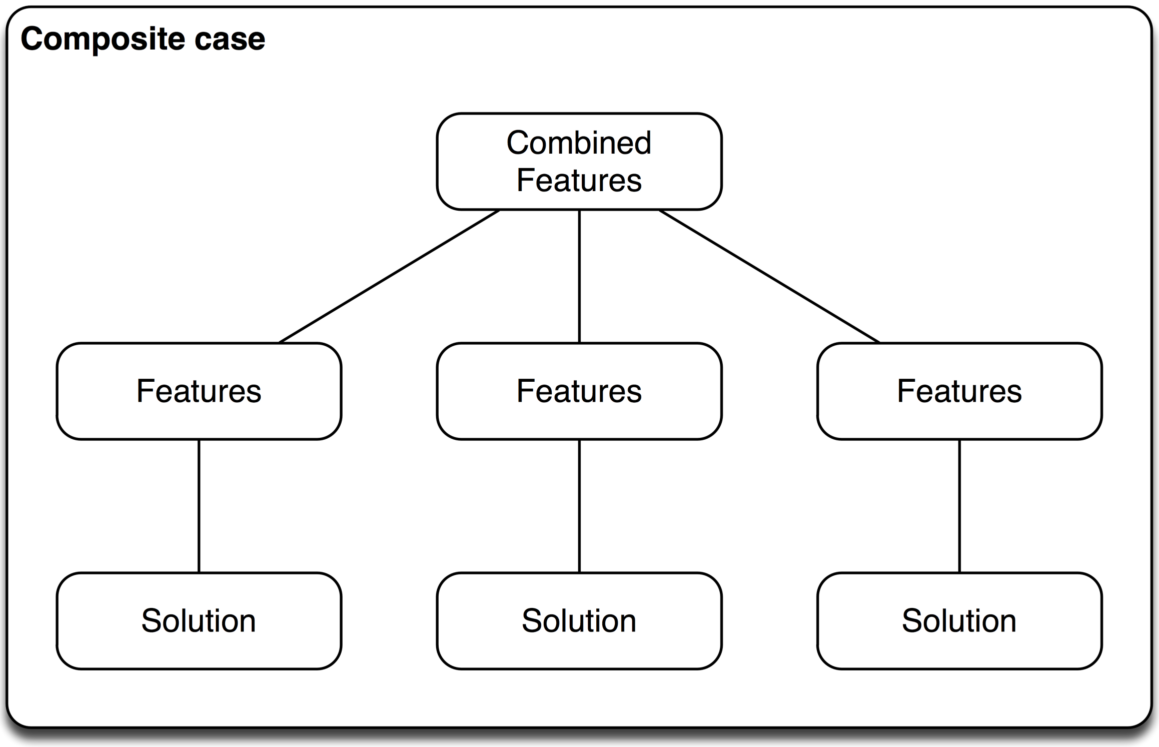

Figure 3.2: **Composite case tree**

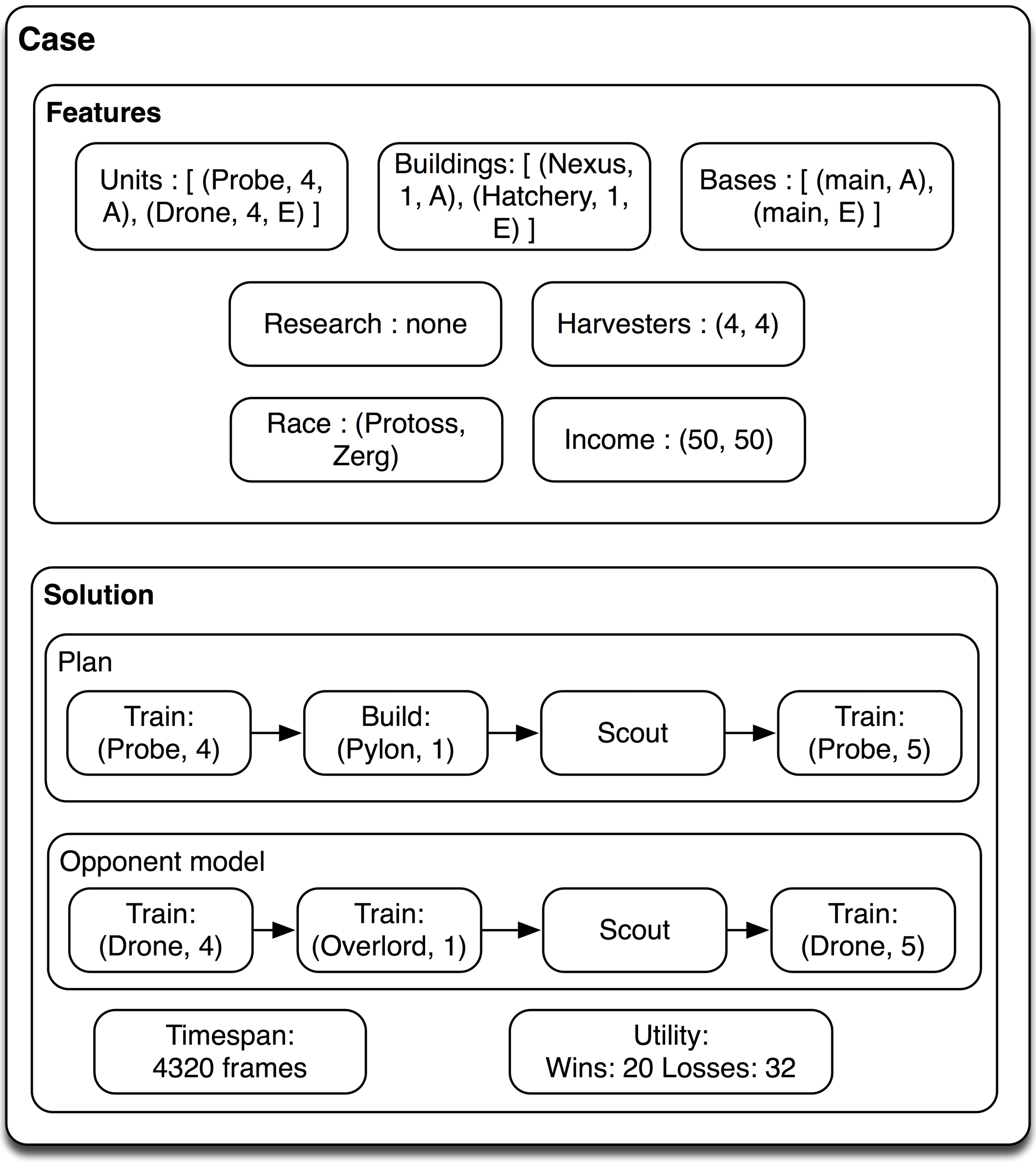

Figure 3.3: **Sample start-case Protoss vs. Zerg**

3.2 Case selection

Here we will explain the selection method of our cases. We will begin by explaining how case matching is performed, and then move on to describe our composite cases/case clustering. Next we will detail how we select cases from the library. Finally we make a note of how our system handles unknowns, i.e. things covered by fog of war.

3.2.1 Case matching

Our features can be represented in a syntactic way since we have numeric values for every feature. By syntactic we mean that we compare the values of each feature based what we can measure from them. If we have 20 Zealots in our army, while the case has 10 Zealots, then we have 50 % similarity. We do not consider the deeper semantic implication this might have. In addition we also implement a more knowledge intensive approach by weighting every feature we intend to match. This makes it possible to do matching on some parts of our cases while disregarding others. By using a filter it is possible to enforce that every feature we match is above a certain threshold.

The first equation in (3.1) is the general one, when both values are non-zero. As a note, we do not operate with negative numbers, so it is never possible to divide by zero. If we have at least one zero value we still would like to know how similar they are, an example is that zero to one unit is more similar than zero to ten units.

$$Similarity(a,b) = \begin{cases} \frac{a}{b} & \text{if } a \leq b,\ a > 0,\ b > 0 \\ \frac{a+\epsilon}{b+\epsilon} & \text{if } a = 0,\ \epsilon > 0 \\ 1 & \text{if } a = 0,\ b = 0 \end{cases} \tag{3.1}$$

In equation (3.1) a and b are values from a feature in the case. One example is that we are finding the similarity between the harvester amount. In that situation, a is the amount of our harvesters from the gamestate, for example 15, while b is the amount of our harvesters in the case gamestate, say 20. The similarity here would be 75 %. If the case gamestate has the lower value, we switch the values of a and b in the equation. This means that if we have 20 harvesters and the case has 15, the similarity is still 75 %.

Next, we use a filter. If any value a is below a certain threshold value f, we truncate it to zero. This allows us to disregard cases which have a similarity below a certain threshold. The filter is shown in equation (3.2):

$$Filter(a, f) = \begin{cases} a & \text{if } a \geq f \\ 0 & \text{if } a < f \end{cases} \tag{3.2}$$

The complete matching function is shown in equation (3.4). For each feature, a, we compare with the case value, b. To this we apply a certain weight, W_a, specific to feature a. It is then processed by the filter we presented in equation (3.2), and finally divided by the total weight (3.3). As a note, in our implementation f is always zero, meaning all cases are considered.

$$W_{total} = \sum_a W_a \tag{3.3}$$

$$Match(a, b, f) = \frac{W_a \cdot Filter(Similarity(a, b), f)}{W_{total}} \tag{3.4}$$

Below follows an explanation of each feature we are currently comparing. When we state that two values are compared to each other, we are implying that we use the equation shown in equation (3.1). Each feature has a certain weight applied to it.

- Resources and resource rate.

This is done by comparing the current minerals, gas and their respective rates to the ones found in the case. Each value has an equal impact to the total similarity.

- Harvester count

This is done by comparing ours and the enemy's harvester count with the one found in the case. Each count has an equal impact to the total similarity.

- Research

This is done by comparing each research we and the enemy possess with the ones found in the case. Each research that matches to the ones in the case is added to the matched value, while every research counts toward the total. An example is if we have Protoss Ground Weapons upgraded to 3, while the case have them upgraded to 1. We would now add 2 (1 for our research, 1 for case) to the matched value, while we would add 4 (3 ours total, 1 case total) to the total. The similarity follows the equation in equation (3.1), with matched as a and total as b. The enemy's and our research have an equal impact to the total similarity.

- Ours and enemy's units and buildings

We compare buildings and units in the same way. First, we match on a unit to unit basis. Only identical unit and building types are considered matching in this stage. Then we compare the total amount of minerals and gas value of the units and buildings. Both these results count 50 percent towards the total similarity. We have a separate weighting for the buildings and units, and one for whether they are ours or the enemy's.

- Bases

Finally we match the amount of bases we and the enemy have. The enemy's and our bases have an equal impact to the total similarity.

3.2.2 Case clustering

In order to achieve better searching efficiency it is important to minimise the search space. Several techniques have been devised for this purpose. Our approach works by clustering cases with similar features as shown in figure 3.4. Depending on the threshold for similarity, this can lead to a dramatic reduction in the number of feature-vectors we have to search.

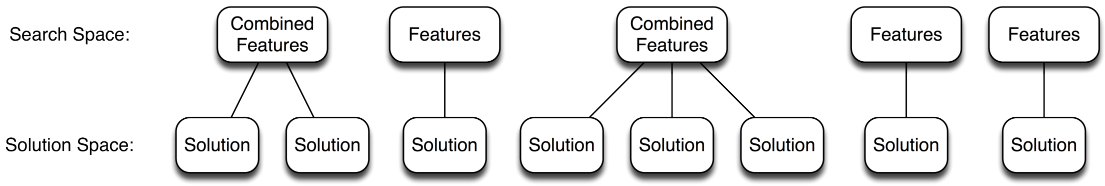

Figure 3.4: **Casebase with clustered features**

As we explained in section 3.1.3, a composite case is a case created by combining two or more cases' features. This is done by setting each feature in the composite case to the average of all the cases it encompasses. When search among composite cases we match on the combined features, and select one of the cases it contains.

3.2.3 Selection method

Our selection method is divided into searching and finding a subset of matching cases, and then selecting between them.

Searching for cases is done by comparing the input gamestate to each case that is not part of a composite case, and then comparing to each composite case. Any case above a certain similarity threshold is kept for the next part of the matching. If there are less than 5 cases above this threshold, we choose the 5 best matches we have.

The second part is to select one case we will use in the agent. This is done by giving each case a certain probability of being selected, by the formula found in equation (3.5). If we select a normal case, it will be returned to the agent. If we select a composite case, we will return one of the cases contained within, using the same probability function on each case.

$$Probability = \frac{Similarity \cdot W_{amount} \cdot (R_{winratio} + \epsilon)}{Total} \quad \epsilon > 0 \tag{3.5}$$

In equation (3.5) *Similarity* is the similarity between the gamestate and the case. W_{amount} is the amount of cases this probability is referring to, that is, if its a composite case. If it is a normal case, W_{amount} is 1. $R_{winratio}$ is the winning ratio of this case. We have added an epsilon to ensure that cases that leads to only losses have a small probability of being selected. This is to ensure that we cannot avoid any strategy, and leads to a more unpredictable and human-like agent. Total is the total amount of all probabilities, to ensure each probability is between 0 and 1.

From this equation we can observe two things. First, composite cases are in general more likely to be selected than normal cases. This according to our design, as a composite case shows general tendencies towards performing similar strategies. They also contain more refined data, in the form of more accurate win ratios. The second observation is that cases that have better outcomes are selected more often. If the case library is updated with the agents own replays, this will give us an automatic tuning of the selection strategies of the agent, which is desirable.

3.2.4 Handling unknowns

When we create our cases we utilise perfect map information, and always have an accurate representation of the enemy's plan and forces. When we play our own matches we do not have this advantage, and must resort to less efficient ways to find out what the enemy is up to. This creates a discrepancy between our cases and the gamestate we are able to provide them, as our current information could be outdated, or incomplete. The process of handling these unknowns is an important aspect of our agent. We have a few possible strategies to counter this problem, but they have not been implemented yet.

First, we already have an opponent model that we can use to predict the enemy's current buildings and units. If we do not have adequate scouting information when we are selecting a new plan it is possible to use the predicted buildings in the gamestate. This enables us to find better matches if the opponent is following the predicted plan. The drawback is that we can enter a negative cycle where we follow obsolete plans due to assumptions. This is a necessary sacrifice in our environment, but it can be reduced by actively finding out what the opponent is up to, which is a good plan in general.

Second, we are able to select weights when matching cases. If we have not been able to see what the opponent has been building yet it is possible to set the weights of the enemy's units and buildings to zero. This allows us to perform partial matching on the gamestate, which can be useful early in the game when we have not been scouting. By not considering the enemy's buildings and units we effectively raise the matching percentage. This is because those features would pull down the total matching percentage, since they match empty lists with units and buildings towards the case's known opponent information.

3.3 Case adaptation

Here we will detail the issues which may arise when trying to use a case, such as missing dependencies, and when a the situation has changed enough for an entire plan to be rejected.

3.3.1 Missing dependencies

During case selection we do not exclude any plans that cannot be executed directly. The idea is that while we may lack a critical building for a plan, if the remaining parts are equal enough we should use this plan. In order to solve this problem we will do a dependency check each time we start a new plan. This is primarily to allow our agent to perform parallel tasks, given that any parallel task does not postpone the current first task of the plan. This dependency check will also add any missing buildings that are required to perform our plan without trouble.

This is equivalent to the adaptation strategy presented by Kolodner([10];Ch. 11), *commonsense transformation*, which is a transformational adaptation strategy. We have access to any buildings or units requirements, and can use these to make simple additions into a plan without the need to use a more complex strategy. An obvious drawback is that we will need more time to complete a plan, but given that plans are selected based on how similar they are to our gamestate, it should not be a severe drawback.

3.3.2 Plan rejection

According to Woolridge[28], agents can have several forms of commitment. Blind commitment is when an agent actively tries to perform its goals until they are finished. Single-minded commitment is when an agent tries to perform its goals until they are done, or it no longer believes they are possible to finish. Open-minded commitment allows the agent to reason about the goals themselves, and change them if it finds better goals.

The concept of commitment to plans is important for the success of the agent. While it is simple to create an agent that performs its plans until they are finished, it would not be optimal if the plan was doomed to fail due to some unforeseen circumstance. This is an example of Blind commitment, and would not suit us in our dynamic environment. What we want is an agent that deliberates on its plans. The problem now is to find out when we should deliberate on our plans. If we deliberate too often, we would spend too much time on doing it, and possibly switch plans regularly. This would lead to an agent that performs many fractioned plans that does not lead to a good result. If we deliberate insufficiently, we risk using obsolete plans.

One solution to plan rejection is to use the opponent model we have from the CBR module. The idea behind the opponent model is that if the opponent performs the same actions we have predicted, then our plan is a counter to his strategy. The problem is now decomposed into finding out when there is a sufficient discrepancy between the opponent model and the opponents actions. One measure we are using is when the opponent has created a building that gives it access to a different technology than what we expected. This could have a major impact on our strategy, and is worth replanning for. Another measure is if the opponent has expanded when we did not expect it. This could lead to new strategies which could involve using our tactical advantage in units, or expanding ourselves.

Our agent is currently using Blind commitment, but has the information required to make it able to reason about its plans.

3.4 Case retention

Here we will explain how we store and learn from our own experiences. First we will present the method we have selected for storing our own experience. We will then proceed with showing how we learn from this storage method, and continue with discussing possible advantages and drawbacks to the selected method. Finally we will present an issue with the retain process.

3.4.1 Case storing

Whenever we play a match, our agent gets concrete data on how well certain cases have performed when the agent uses them. Our approach to storing these new cases is based on the previous work with the replay tracer we have already created. When the agent has played a match, a replay from that match is available. That replay is then analysed into a trace, which is used to create cases. These cases are merged with the current case database, and then merged with the composite cases we already have, if they meet the similarity thresholds. This does mean that entirely new cases are created each time we play a match, even though we followed previous cases, with possible adjustments. In the end this approach offers us a very simple and effective way to store our own cases.

3.4.2 Learning by refining the casebase

The learning part of our CBR module is integrated with the storage algorithm. The agent will store all the new cases that it gets from its own replay, with the result of the match. There is a good probability that some of the cases can be matched with existing composite cases, or create entirely new ones. This is because we already use cases that are in the database, and thus our strategies will try to perform the same strategy over again. Even if we do not match with composite cases, we will create new branches that the agent could follow in an execution. If we do match with previous cases, we will update the win-loss ratio that composite case had. The selection algorithm for cases, see equation (3.5), will use the win-loss ratio in the probability for selecting a certain case. By automatically tuning this ratio with our own cases we achieve a self learning agent.

3.4.3 Pros and cons of case storing method

A good advantage of this approach is that we can offer a pre-tested way of creating cases from our own experiences. The tracer will analyse the replay as if it was an expert that was playing it, and create new cases without any bias to the actual reasoning behind each action. By creating new cases we avoid having bias towards the previous cases we used. This means that we can possibly create entirely new cases that expands the knowledge in the case database.

Another advantage is that we do not have any difference between the initial training cases and the cases from the agents own matches, except for how the match was played. Our training cases assume perfect information, and uses this to create additional information to the agent that it would not have access to when playing itself. The only way to overcome this would be to use the replays to seed additional information to each case. This would mean that we would have to create an entirely separate module which takes in cases while analysing replays.

Another advantage is that the agent fully commits its resources to the actual game it is playing, without the need to carefully store data for the database. This simplifies our agent, and gives it better performance.

The most prominent drawback is that the storage method has to be performed offline. This is a natural feature, since our tracer must run the replay via BWAPI, and that requires more resources and execution time than the agent has available in a match. However, by doing the database update offline we are able to let the agent play an arbitrary amount of matches on his own, and updating the case database when we see fit.

3.4.4 External issue with replays

To our dismay, we discovered late in the project that replays from our own matches were corrupted by BWAPI. The agent would perform the initial actions as usual, but after two minutes the harvesters would stop harvesting, and creating buildings. This first appeared when the agent had more time consuming tasks to perform, so replays saved in the early part of the projects had no issues. The problem with the replays is that they store actions to be simulated in the game, so if one action is corrupted, the remaining ones will also be corrupted. The only way to fix this would be to store information from each match ourselves, but it would not be trivial. Since we did not want to sacrifice the overall progress on the agent, we decided to leave case retention out of the agent, while focusing on testing scaling issues when creating the training set.

3.5 Notes on case revision

The observant reader will notice that we do not explicitly perform case revision. The reason for this is that it has been merged with case adaptation and case retention. The adaptation process will fix dependencies automatically. These are then automatically used in case retention. If a replay leads to a failure, the database will make sure that we perform that strategy less often.

The process of trying to repair cases is a lengthy one with our agent. While it is possible to for a human to look at a replay and pointing out where a player in a replay did something strategically wrong, it is very hard to translate this into rules for a repair module. Another problem is that a case could contain a valid strategy, but our agent fails to perform it correctly. This could be due to certain building placement, or simply controlling attacking units poorly. Repairing these faults is not possible without changing the strategy altogether, or changing the agent. We believe our approach of case retention will perform better due to these circumstances.

Chapter 4

System overview

In this chapter we give a short overview of our system and its dependencies, to inform the discussion in the following chapters.

4.1 The tools

Our agent and tools are written in C++. C++ was chosen because it generates high-performance binaries, while still allowing us to work with several layers of abstraction. In addition, the framework we use to interface with Starcraft is written in C++. By using C++ we avoid the added complication of porting the framework to our language of choice.

4.1.1 Brood War API

Brood War API (BWAPI) is a free open-source C++ framework which gives us access to the same information a player has from the game itself. It has a standard add-on library (BWSAL) that gives the user several modules that does semi-advanced operations, such as automatic building placement and build dependency resolver. The use of this framework greatly lessens the initial start up time of a project, and gives us more reason to use Starcraft as our testing domain. While it is fully possible to create an agent that can play the game in one days work, an agent based on more advanced AI methods still require quite some work. The framework allows us to get started on the more advanced aspects of the agent while eliminating the need to worry about trivial problems when connecting and communicating with the game and its resources.

4.2 The main components

Our system consists of three components, working together to allow our agent to play a successful game of Starcraft. They are as follows:

- Tracer
- Analyzer
- Agent runtime

The first two components are data-mining tools, which allow us to generate a case database from replays. The final component is the agent runtime, which is responsible for using the cases and playing the actual game.

4.2.1 Data-flow in the system

Data flows from our set of replays through the Tracer, which feeds into the Analyzer and is then saved to our database of cases. The Agent runtime uses the database to play a game, which generates a new replay. The data-flow then loops to the beginning, as we see in figure 4.1.

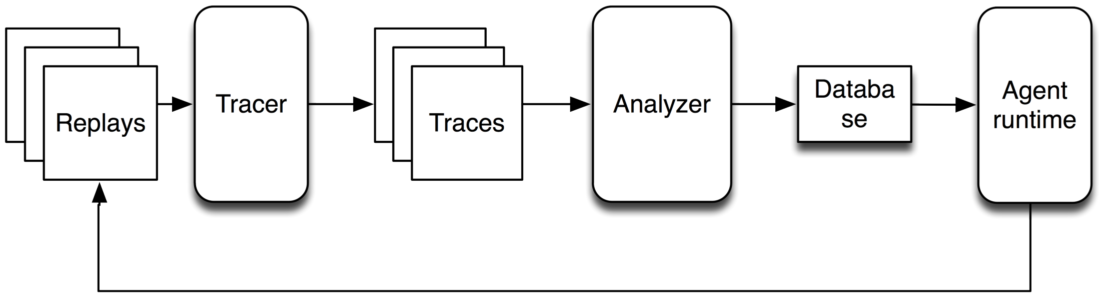

Figure 4.1: **The flow of data through the system**

We now outline the responsibilities and functions of each module.

4.2.2 Tracer

The tracer is a fairly simple tool, born from the pragmatic desire to make data-mining more efficient. We initially analyzed replays directly, as BWAPI was playing them, but this turned out to be a huge limitation.

First, BWAPI does not allow us to skip forward or jump back in time. This made detailed analysis very difficult, as we could not search for the "initiating factors" of important events such as attacks.

Second, analyzing in real-time is a very time-consuming process, and we soon found that analyzing a single replay could take as long as 30 minutes. If we were ever to scale up this solution we would need several computers analyzing replays in parallel. This was not an option, and we went in search of alternate solutions.

The answer to both these problems was the tracer. It runs through replays, and stores enough information to make efficient analysis possible. These traces are generated once, and analysis can then be run efficiently on the traces.

Internally, the tracer interfaces with BWAPI to view replays and saves the state of the game for each frame. This naturally leads to a significant amount of data, and we compress by only saving deltas when something changes. We store data such as when structures are built, units are trained and research initiated. We also store data on a per-unit basis, for instance its location at different times, and who the unit is attacking. Together, this information allows us to recreate situations at any point in the game.

4.2.3 Analyzer

The analyzer takes its input from the tracer and searches for relevant events which occurred during each game. It uses this information to generate cases, which we later use in our agent runtime.

From the traces we are able to recreate a detailed gamestate, which we use to search for events. Events are actions ordered by either player, which we feel are significant enough to be reenacted by our agent while it plays a game. It then separates these into sequences of actions, which serve as the solution of a case. We also record the gamestate at the start of the sequence, from which we can identify similar cases in the runtime.

The analyzer feeds these cases into our case database.

4.2.4 Agent runtime

The agent runtime is by far the most complex component of our system. Here, we will only give a short overview of its functions, interested readers are directed to appendix B, where we give a much more detailed description. The agent runtime was also the main focus of our specialization project[22], which contains an even more detailed description as well as the reasoning behind our design.

The agent runtime is designed with a three-layered architecture. The layers are:

- Planning layer
- Executive layer
- Reactive layer

The planning layer is our main area of interest, as our thesis is based around planning with CBR. The planning layer is responsible for monitoring the games progress and suggest plans which lead to victory. How it achieves this will be explained in detail in the following chapters. It then communicates this plan to the executive layer.

The executive layers main responsibility is to execute plans received from the planning layer. This includes allocating resources, ordering attacks and maintaing good scouting coverage of the play area. The executive layer also monitors plan progress, and reports to the planning layer if a plan seems impossible to execute or if it fails due to unforeseen events.

The reactive layer is designed to control individual units or groups of units. It receives an allocation of units and a goal these are supposed to achieve. It then generates behaviours designed to fulfill the goals, and reports the outcome back to the executive layer. The behaviours can be modified and extended at runtime, and are largely inspired by work done in the field of hierarchical task networks (HTNs)[5].

Chapter 5

Experiments and Results

In this chapter, we refine our research questions to testable experiments, and define performance measures for the experiments. We present the method by which the experiments were conducted and finally the results we observed.

5.1 Architecture and Model

All the tests are performed by playing the agent against the built-in AI opponent of Starcraft.

Our agent is configured with different casebases, and then plays a succession of games, while tracking different performance measures.

5.1.1 Casebases

Our casebase is derived from 956 replays scraped from several websites[1] which maintain databases of replays. As a note, we initially had 4200 replays as a base for creating traces, but only 956 passed our inclusion requirements. Examples of inclusion requirements are the ability to find a winner in the end, and only two distinct players in the game. We filtered the replays such that at least one of the players had to be Protoss for the replay to be used. If the game was played as Protoss vs. Protoss we would generate cases from both players viewpoint. These replays were run through the replay tracer and then the analyzer to generate a

[1]Sites include www.gosugamers.net, www.iccup.com and www.teamliquid.net

casebase. After filtering corrupt replays and cases, our source casebase contained 23106 cases.

From this source casebase we generated several new casebases with different characteristics. A separate tool would resize the database or run our clustering algorithm. The clustering algorithm was configured to combine cases with more than 90% similarity.

Table 5.1 lists the casebases which we use in our experiments. Cases and clusters are the size of the searchable feature-vectors. Their sum is the size of the search-space. Solutions are the total number of plans the agent can select from. Finally, solutions per cluster is the average number of solutions contained in the case clusters. We will be referring to this table during the rest of this chapter.

id	cases	clusters	solutions	solutions per cluster
1	23106	0	23106	0
2	2300	0	2300	0
3	230	0	230	0
4	1316	3518	23106	6.2
5	928	405	2300	3.4
6	186	19	230	2.8

Table 5.1: The casesbases used in the experiments

5.1.2 Tracking performance measures

If the agent is configured to run in test-mode, a module in the agent will track several different measurements of the agents performance. The module will track every measurement in every game, whether it is needed for the given experiment or not.

The module tracks these measurements for each game played:

- Outcome of the game, one of (Win, Loss, Crash)
- Races in the game, ours and the opponents.
- The map we played on
- Which casebase was in use
- Which cases were selected during the game
- The match between selected case and current gamestate

- The time, in milliseconds, to select a case

5.1.3 Additional notes on experiments

Because Starcraft must run in realtime and we need several games to produce statistically significant results, running the experiments is very time-consuming. With the capability of tracking every measurement we therefore decided to base several experiments on the results from a single run, if the input was compatible. For instance, given a large database we can track both the performance of case-selection and the agents win-rate in a single run.

The measurement module is also designed to work incrementally. Despite our best efforts to create a robust agent, both it and BWAPI contains bugs. Some of these lead to fatal crashes, and we decided to design the experiments such that they could be halted at any point and then resumed later. For non-fatal bugs we merely record that the game could not be finished.

5.2 Experimental Plan

In this section we detail the tests we wish to perform, and what the measure of the tests are.

5.2.1 Effect of database size on runtime performance

Test details

In this test we will run the agent with differing database sizes, and measure the time to find a match with each. The database will only contain normal cases, that is, it will have no case clustering.

Over the course of 3 runs we will use databases of 1%, 10% and 100% of the source database. the 10 % is created by selecting a tenth of the 100 % database, while the 1 % is created by selecting a tenth of the 10 % database. Each run will contain 100 games.

For each run, we will measure the average time to select a solution, the minimum and maximum time to select a solution and standard deviation.

Test reasoning

We perform this test to show how the database size affects the runtime performance of our agent. It is important that the agent is able to perform its tasks within a certain time limit, since a plan could become obsolete if we spend too much time deliberating on it. It also gives us a measure of how often we are able to create new plans when the agent is playing a game, which is important if there is a need to change plans dynamically.

We also wish to find out how large the database can theoretically be. If there is a low threshold of the size we can have, we must be more selective when choosing which replays to use. Having a the highest theoretical size of the database means that we limit the ability for the agent to update the database with its own replays.

Test results

The test results we achieved are shown in table 5.2. In this table, Total matched refers to the number of times the CBR module was called upon to find a solutions. Avg is the average time to select a case over all runs. Avg / case is the average time to match a single case. Std.Dev, Min and Max are respectively, the standard deviation of time to select, the minimum and maximum time to select. Times are in milliseconds. Run is the identifier of the run in this experiment, while Casebase identifies a casebase listed in table 5.1.

Run	Casebase	Total matched	Avg	Avg / case	Std.Dev.	Min	Max
1	1	743	8.82	0.038	7.42	0.0	71.0
2	2	734	569	0.247	574	70	3835
3	3	902	6007	0.260	4005	603	19,550

Table 5.2: The results from experiment 1

Discussion of results

We believed that the runtime performance of the agent would be somewhat linear based on the size of the database. This is because the matching algorithm runs over every case it has once, and then selects one of the best candidates. The matching algorithm is linear, as it only depends on the amount of information each feature contains, and not on the other cases in the library. We thus expected the matching time per case to be fairly constant, and the average retrieval time to scale linearly with the case size. Our results show that the average time retrieve a case from the database increases more than linearly. The time to perform matching on a single case is on average 0.038 ms in the smallest database, while it has increased to more than six times that amount with the medium and large databases. This is an interesting and unexpected result, which needs to be investigated.

Another important aspect of these results is that the agent uses too much time to deliberate. When we play against the built-in AI, it does not matter if the game freezes a few second when matching. The gamestate does not change during that time. This is an effect of Starcrafts internal architecture.

5.2.2 Effect of case clustering on runtime performance

Test details

In this test we will run the agent with differing database sizes, and measure the time to find a match with each. In this test we will enable case clustering in order to see how this changes the performance.

Over the course of 3 runs we will use databases of 1%, 10% and 100% of the source database. the 10 % is created by selecting a tenth of the 100 % database, while the 1 % is created by selecting a tenth of the 10 % database. Each database will have composite cases as well as normal cases that did not match well enough for composite cases. Each run will contain 100 games.

For each run, we will measure the average time to select a solution, the minimum and maximum time to select a solution and standard deviation.

Test reasoning

We perform this test to see if our case clustering method has any notable improvements on the effectiveness of the agent. This is an important result if it does give an improvement, as it allows us to further increase the database size.

Test results

The test results we achieved are shown in table 5.3. The structure of this table is identical to that in 5.2. See its description for an explanation of the headings.

Run	Casebase	Total matched	Avg	Avg / case	Std.Dev.	Min	Max
1	4	505	8.91	0.043	8.36	0	70
2	5	738	507.00	0.380	519.00	60	3,515
3	6	668	11,004.00	2.87	8,328.00	470	30,263

Table 5.3: The results from experiment 2

Discussion of results

The average matching time is fairly equal to the time we had without clustering, except for the large database. The main difference is that our databases now consist of considerably fewer cases that we match with, as each composite case is matched once, while each case that it consists of is not used directly in the matching. This results in a large increase in the average matching time per case. The large database now consists of 1,316 single cases, and 3,518 composite cases, which gives us 4,834 total cases when matching, compared to 23,000 without composite cases. The medium database requires almost 9 times as much time on average to match each case compared to the small database. The large database uses almost 67 times the average matching time of the small database. As we explained in the previous test, we expected a linear relationship between the average case retrieval time, and a fairly constant average matching time per case. This is not true and needs to be investigated.

The most important result here is that case clustering increases the time required to retrieve a case from the large database. The medium and small databases have as good as equal retrieval time. We expected case clustering to reduce the time required, since we only have to consider a subset of the total database. Our databases also show that case clustering is possible, and more effective on the large database, so the fact that it increases the time is very unexpected.

5.2.3 Effect of database size on win performance

Test details

In this test we will run the agent with differing database sizes, and measure the impact this has on the agents winning rate. The database will only contain normal cases, that is, it will have no case clustering.

Over the course of 3 runs we will use databases of 1%, 10% and 100% of the source database. the 10 % is created by selecting a tenth of the 100 % database, while the 1 % is created by selecting a tenth of the 10 % database. Each run will contain 100 games.

This experiment is designed to measure what effect scaling up the database has on the selected strategies, and whether the agent improves with a larger database.

For each run we will be measuring the wins, losses and the ratio between these.

Test reasoning

We perform this test to show if we get a better agent by scaling the database. This is an intuitive notion, since more experience often yields better understanding in other fields. By testing this in our agent we prove if this is true for CBR in Starcraft as well.

Another important reason behind this test is how well it is able to perform in terms of winning based on the size of the database. A large database could mean that the agent does not find adequate strategies since it has so many to choose from. The concept of having an optimal size of the database when it comes to winning games is something that can be shown from this test. This could lead to the idea that one can exchange cases in the database with cases that perform better.

Test results

The test results we achieved are shown in table 5.4. Run and Casebase are as for tables 5.2 and 5.3. Wins and Losses record how many games were finished with that outcome. Crashes lists the times a game could not be finished, due to some error in our bot. Finally, the Ratio Win/Losses is calculated as $Ratio = \frac{Wins}{Wins+Losses}$.

Run	Casebase	Wins	Losses	Crashes	Ratio Win/Losses
1	1	1	78	21	1.28%
2	2	2	77	21	2.59%
3	3	5	77	18	6.49%

Table 5.4: The results from experiment 3

Discussion of results

As we can see, we have fairly poor winning results against the built-in AI. The most important result we can derive from this is that a larger database does increase the winning percentage. The increase is fairly small, but compared to the other percentages we have a good increase. The real question is whether this is because our plans are poor, if we select plans that are not suited for the situations or if the agent does not execute plans in a good way. One observation we made during the testing was that the agent was not able to direct his units to the enemy's bases, save the initial one. This lead to poor use of military units if the agent managed to kill the initial base, and then stopped using those military units.

5.2.4 Effect of case clustering on win performance

Test details

In this test we will run the agent with differing database sizes, and measure the impact this has on the agents winning rate. In this test we will enable case clustering in order to see how this changes the performance.

Over the course of 3 runs we will use databases of 1%, 10% and 100% of the source database. the 10 % is created by selecting a tenth of the 100 % database, while the 1 % is created by selecting a tenth of the 10 % database. Each database will have composite cases as well as normal cases that did not match well enough for composite cases. Each run will contain 100 games.

This experiment is designed to measure what effect scaling up the database has on the selected strategies, and whether the agent improves with a larger database.

For each run we will be measuring the wins, losses and the ratio between these.

Test reasoning

The reason we choose this test is to see if our composite cases give any notable difference in the winning performance of the agent. Our composite cases are designed to allow the agent to select more accurately between cases, since they offer a win/loss ratio of the cases it contains. This should, in theory, give it more discrimination between cases in when it needs a plan, and thus provide better plans. We wish to see if this theory is correct.

Test results

The test results we achieved are shown in table 5.5. See 5.4 for an explanation of the table.

Run	Casebase	Wins	Losses	Crashes	Ratio Win/Losses
1	4	1	89	10	1.12%
2	5	3	79	19	3.79%
3	6	4	76	20	5.26%

Table 5.5: The results from experiment 4

Discussion of results

As we can see, the results are fairly equal to the test without case clustering. The agent manages to increase his winning percentage with larger databases, but the percentages are much lower than what we wanted them to be. Until we can get the agent to play better it is hard to see if case clustering has a good effect on the win/loss ratio. Our results so far leads to the conclusion that case clustering is not increasing our winning performance yet.

5.2.5 Improvement from learning

This would be a run of 100 games, learning from those 100 games, running another 100 games and recording improvements. However, replays become corrupted, and we cannot run this test. The reason we wished to run this test is to see if our agent becomes better at case selection when it refines its own database. This is an important concept in CBR, which we discussed in section 2.2.6. An agent that refines its database should become better at similarity matching in the future. In our system this means that the agent will choose winning strategies more often, and avoid loosing strategies. To be able to prove this with testing results would mean that our agent could have great potential of automated learning, which is a desirable property in an agent.

5.3 Experimental Results

In this section we discuss how the results interact across the experiments.

5.3.1 Timing

As we see from the experiments measuring case selection performance, things do not look good for scaling up the casebase. In addition, the case clustering algorithm which we believed would lead to a significant performance boost actually increases selection time by nearly 50%.

The selection algorithm is linear in the number of feature-vectors it looks through. If we use the 1% casebase (with about 9ms. selection time) as our baseline, we should observe about 90ms. for the 10% casebase. Instead, we see nearly 600 ms, which is closer to the 900ms. we expect for the full casebase. This, however, takes nearly 6 seconds to select a case.

After these surprising results, we manually reviewed the timing of the selection algorithm. To our dismay, we discovered that the majority of the time was spent copying the casebase. On disk, the source casebase is about 50 Mb. and is expanded to nearly 150 Mb. in memory. This is copied every time we attempt to select a case. Our more careful review showed that copying this database can take as long as 4-6 seconds, while the remainder of the selection algorithm only takes about 200 ms.

We also performed these tests with the clustered casebase. Due to the increased complexity caused by clustering, the copying takes even longer in this case. In this situation the actual matching was reduced to about 100 ms.

We believe it should be possible to redesign the selection algorithm to avoid this copying, and if so, this would lead to a significant improvement in selection times.

Chapter 6

Evaluation and Conclusion

6.1 Summary

In this paper we present the issue of creating CBR modules to be used in RTS games. We then show an area that has not been explored to a great extent namely the use of large scale casebases. We present our goal of creating a database from automatic replay analysis, and how to manage the scale of the database.

Further we present a brief overview of our previous work on creating a viable agent architecture for Starcraft. We show the reasoning behind our choice of a layered approach, and how the agent will solve the tasks of the RTS domain. We continue with a detailed overview of the CBR method and show some systems that have been created using CBR in our domain.

We then show the CBR system we have created and explain the design decisions we made to tailor it to Starcrafts domain. We also explain any discrepancies between the earlier description of the CBR method, and why these were made.

Since our system is complex, we present an overview of the different components and the dataflow between these. The most important of these are the tracer that stores Starcraft replays in a more accessible format, traces, the analyzer that extracts cases from these traces, and a brief overview of the agent.

We then present our results of our own experiments, and why we selected them. We ran four different tests, finding the runtime performance with a normal and a clustered casebase, and finding the winning performance with a normal and a clustered casebase. We proceed with discussing the results across the tests.

6.2 Evaluation

6.2.1 CBR for planning - scaling issues

Our work has shown how well CBR can be used for planning in Starcraft. Our main focus was to find out how an agent with a large database performs. The results of our testing shows that the agent should be able to have a good performance with larger databases after some redesigning of our selection algorithm. With this redesign, the effect of case clustering should be more apparent. With our large database, 94 % of the cases where eligible for clustering, and we reduced the size of the part database of the database that is relevant for matching to 21 % of the original size. The effect of this should be even more apparent with larger databases. In our opinion, the agent has a lot of potential, although it is still a work in progress.

6.2.2 Creating a casebase from replays

Our work with creating the database using automatic replay analysis has been a success. The use of traces greatly lessens the time required to create the casebase. Our Tracer does require a fair amount of time to generate the traces from replays, as it needs to simulate each game via BWAPI. The creation of all our traces too about three days, but we only required 2 hours to create the large casebase of more than 23,000 cases. This allows us to avoid worrying about making changes to the case representation, or refining the analysis module.

6.2.3 Creating an effective agent for Starcraft

We also wish to elaborate on the winning performance of the agent. As we have shown in our tests, the agent still performs fairly poorly against Starcrafts built-in AI. We believe that there are a few causes to this. In our opinion, the main problem is how the agent executes plans. Due to time constraints, we were not able to implement the agent to play as well as we had hoped. As a consequence of this, the executive and reactive layer does not have some features that could help with plan execution. One issue is that the agent has a hard time finding every building the enemy owns, and does not press its advantage when the first base has been destroyed. Another issue is that the agent does not spend all its resources. During the initial few plans, the main constraint is the resources, but as we progress, the agent accumulates much unspent resources. The main constraint then becomes how fast it is able to produce units, and we have not made the agent evaluate its plans on those premises.

Another issue is that our replays are of experts playing Starcraft. The built-in AI has a tendency to perform a strong rush strategy, and our cases are more tailored towards longer matches. While we have replays of some rush strategies, the agent would probably improve it it received more training in these scenarios.

6.3 Discussion

At the outset of this thesis, in section 1.3, we described two major areas we wished to investigate. In this section, we discuss how well our research has accomplished its goals, and whether our research questions have been answered.

6.3.1 Case construction

One of our goals was to determine how well an automated case construction tool could perform.

One of our first discoveries was that the most time-consuming step in this process is playing through replays in BWAPI. Since replays are stored in a compressed format by Starcraft it is impossible to analyze these directly, because no information about state is stored. We have solved this problem by using a separate trace tool which runs the replays and saves necessary state. We recommend that anyone attempting to analyze replays use this approach.

6.3.2 Performance in scale

Our initial research showed that very little work has been done in determining how well CBR scales up.

We were successful in creating a large database of cases, and performed several measurements of its performance. Our experiments showed a tendency to improved play by our agent when it used a larger casebase. Unfortunately our agents micromanagement is not robust enough to use plans effectively, and the results are skewed because of this. We believe the plans are improved, however, we can not make a conclusion one way or the other.

We have also measured the runtime performance of an upscaled casebase. This is easy to measure, but there are no absolute limits above which the solution becomes unusable. One indicator is provided by the rules of the AIIDE tournament[1]. It states a bot is disqualified if:

- ≥ 2 frames exceed 10 seconds
- or ≥ 10 frames exceed 1 second
- or ≥ 200 frames exceed 55ms

By these rules, our solution would struggle. Selecting cases is a time-consuming process, and is performed often enough that we would violate the second rule. Despite this, the agent should be able to play without noticeable delay versus a human player. Due to this we believe that scaling up a CBR solution solution shows promise, but would benefit from more research and testing.

6.3.3 Answers to research questions

In section 1.3 we posed three research questions we wanted to answer. We now provide the answers here.

We have determined that automated case construction from expert replays is a viable solution to case construction. The process is time-consuming, especially running the replays through Starcraft, but the resulting casebase is well worth the effort.

An agent with a large casebase has shown a tendency to outperform one with a smaller set of cases. We believe the effort in creating the larger database will pay off in improved plans utilized by the agent.

A planner based on case-based reasoning is a complex piece of software. A large casebase needs a smart selection algorithm to be able to perform in a real-time strategy game such as Starcraft.

6.4 Contributions

We have made three major contributions by creating this agent. The first is that we have researched on the viability of creating and using a large casebase for

[1]http://skatgame.net/mburo/sc2011/rules.html

planning in Starcraft. Other parties may use our results to further test the limits of CBR in RTS, or other similar domains.

The next contribution is the CBR case creating system we have made, the Tracer and Analyzer. These subsystems are independent of the agent, and can be further refined, or used to create cases for other areas in Starcraft.

The last contribution is the complete agent we have made. While it is a work in progress, other parties could use our system and put in their own ideas and subsystems. The architecture has been designed to facilitate change of its internal modules. While we have made a good planning system with CBR, there are areas to improve in order to make the agent better. This could include working more on creating better behaviours, or completely redesigning the reactive layer. We had to create our system from scratch and can acknowledge that it is a lot of work.

6.5 Future Work

Our goal was to create a complete agent that could play Starcraft. While our agent can play Starcraft without any constraints on the games, there are still areas that could be improved. Here we will present some of the ideas we have for further work on our agent.

6.5.1 Smarter planning

While our planning layer manages to generate plans, we have a few ideas on how it can be improved. First, the use of context information when selecting plans could be very helpful. We have made one such approach to alter plans that try to create expansions while we already have an expansion that is about to be built. The issue is also apparent when we are performing an attack. As it is now, the most common final action of a plan is to perform an attack. We generate new plans when we issue this command, and many of the combat units that are available at the plans beginning might be occupied or dead by the end of it. By improving the information we supply the planner with, it is possible to generate more valid plans.

The second idea is to use the opponent model when scouting to a greater extent. If we are able to show that a plan has become invalid, we can alter our strategies to perform better in the long run.

6.5.2 Smarter selection algorithm

Our experiments have shown that the size of the casebase has a significant impact on case selection performance. We have proposed one method which improves the selection time by reducing the search space, however we believe there are other methods which may further improve this.

6.5.3 A goal-oriented representation of plans

The current representation of plans is very straight-forward. We represent plans as a list of actions to be performed directly. This leads to a lot of adaptation of plans, which in turn leads to an increase in the size of the casebase as new cases are learned. We believe that representing plans as goals would alleviate this problem.

As an example, the sequence of actions *Train(5, Probe)*, *Build(2, Pylon)* would be changed to *Have(5, Probe)*, *Have(2, Pylon)*. The agents executive layer can then check if goals are fulfilled or need to be expanded, but the results of the plan are invariant to the state of the game. This would also reduce the need for adaptation algorithms, which should improve performance further.

6.5.4 Improving micromanagement

The reactive layer was designed to be able to plug in new behaviours at will. Creating good subsystems to control units can be a separate thesis in itself, and our work here is only basic functionality. One area we wanted to improve ourselves was the attack algorithm. Currently it is overzealous and will not stop attacking the enemy until it has either won or lost all its units. Creating methods to evaluate the viability of an attack would be most helpful, along with a retreat behaviour.

We also want to improve the target selection within the attack behaviour. The current system is designed to attack nearby units, based on how valuable they are. It does not consider its own positioning, and whether a target far away is poorly defended. Our agent often goes straight into the fray, heedless of the opponents defences and ranged units. An improvement here would go a long way in preserving units.

6.5.5 Improving the executive layer

One apparent problem with the agent was that it did not properly store where the enemy was situated. Our idea is to scout more effectively and store relevant information such as base locations, known army and defences. This information would form the backbone of a better attack algorithm.

References

[1] Aamodt, A. and Plaza, E. (1994). Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–59.

[2] Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An Architecture for Autonomy. *International Journal of Robotics Research*, pages 1–38.

[3] Buro, M. and Furtak, T. (2003). Rts games as test-bed for real-time ai research. In *Proceedings of the 7th Joint Conference on Information Science (JCIS 2003)*, pages 481–484.

[4] Cheng, D. and Thawonmas, R. (2004). Case-based plan recognition for real-time strategy games. In *Proceedings of the Fifth Game-On International Conference*, pages 36–40.

[5] Erol, K., Hendler, J., and Nau, D. (1994). Umcp: A sound and complete procedure for hierarchical task-network planning. In *Proc. AIPS*, volume 94, pages 249–254.

[6] Gentner, D. (1983). Structure-Mapping : A Theoretical Framework for Analogy. *Cognitive Science*, 7:155–170.

[7] Hammond, K. J. (1989). *Case-Based Planning: Viewing planning as a memory task*. Academic Press Professional, Inc., San Diego, CA, USA.

[8] Hammond, K. J. (1990). Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228.

[9] Kofod-Petersen, A. (2012). How to do a Structured Literature Review in computer science.

[10] Kolodner, J. (1993). *Case-based reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[11] Kolodner, J. L. (1983a). Maintaining Organization in a Dynamic Long-Term Memory. *Cognitive Science*, 7:243–280.

[12] Kolodner, J. L. (1983b). Reconstructive Memory : A Computer Model . *Cognitive Science*, 7:281–328.

[13] Koton, P. (1988). *Using Experience in Learning and Problem Solving*. PhD thesis, Massachusetts Institute of Technology, Laboratory of Computer Science.

[14] Mateas, M. and Stern, A. (2002). A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4):39–47.

[15] Micić, A., Arnarsson, D., and Jónsson, V. (2011). Developing Game AI for the Real Time Strategy Game Starcraft. Reykjavik University.

[16] Ontañón, S., Bonnette, K., Mahindrakar, P., Gómez-Martín, M., Long, K., Radhakrishnan, J., Shah, R., and Ram, A. (2009). Learning from Human Demonstrations for Real-Time Case-Based Planning. In *IJCAI-09 Workshop on Learning Structural Knowledge From Observations (STRUCK-09)*.

[17] Ontañón, S., Mishra, K., Sugandh, N., and Ram, A. (2010). On-Line Case-Based Planning. *Computational Intelligence*, 26(1):84–119.

[18] Russell, S. and Norvig, P. (2010). *Artificial intelligence: a modern approach*. Prentice hall.

[19] Schadd, F., Bakkes, S., and Spronck, P. (2007). Opponent modeling in real-time strategy games. In *8th International Conference on Intelligent Games and Simulation (GAME-ON 2007)*, pages 61–68.

[20] Schank, R. C. (1983). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, New York, NY, USA.

[21] Szczepański, T. and Aamodt, A. (2008). Case-based reasoning for improved micromanagement in real-time strategy games. *Positioning paper NTNU-IDI*.

[22] Tornes, D. and Eriksson, J. (2011). A layered architecture for a starcraft agent.

[23] Weber, B. G. and Mateas, M. (2009a). Case-Based Reasoning for Build Order in Real-Time Strategy Games. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 106–111. The AAAI Press.

[24] Weber, B. G. and Mateas, M. (2009b). Conceptual Neighborhoods for Retrieval in Case-Based Reasoning. In *Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR)*, pages 343–357.

[25] Weber, B. G., Mateas, M., and Jhala, A. (2010a). Applying Goal-Driven Autonomy to StarCraft. In *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*. The AAAI Press.

[26] Weber, B. G., Mawhorter, P., Mateas, M., and Jhala, A. (2010b). Reactive planning idioms for multi-scale game AI. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 115–122. IEEE.

[27] Weber, B. G. and Onta, S. (2010). Using Automated Replay Annotation for Case-Based Planning in Games. In *Proceedings of the ICCBR Workshop on Computer Games*.

[28] Wooldridge, M. (2002). *An introduction to multiagent systems*. Wiley.

Appendix A

Starcraft

Central to this project is the game "Starcraft". Here we will present the game at a level of detail intended for a reader that is not familiar with the game itself or the Real-time Strategy genre.

A.1 The game

Starcraft is a military science fiction real-time strategy (RTS) game developed by Blizzard Entertainment. The game was released 31st of March 1998, and is one of the best selling games (11 million copies as of February 2009) for personal computers. It deviated from the common way of implementing RTS games at that time, which became a huge success. The game was highly praised by critics and game players alike. Its expansion, Starcraft Brood War (SC:BW) has a pro-gaming league in South-Korea. Although the game is largely replaced by Starcraft II, it is still the de facto benchmark that new RTS games are compared to today. This is because of its extensive testing and balancing for more than 10 years.

A.2 The Real-time Strategy genre

In RTS games, the player has control of units and structures and must make tactical decisions to destroy the opposing players assets. The players have to accumulate resources by making buildings and/or a dedicated resource gathering unit (worker), and then expend these resources in different areas. Examples can be to make a large army, or to invest in more buildings and technology that will unlock better units.

The standard way to play these games is that each player is given an equal amount of resources, a small amount of gathering units and a structure to make more. From this modest start, the players have to decide how to best make units and buildings that will ultimately lead to victory. There are other ways to play the game though, but since these are not relevant to the agent we are making we will not elaborate on them.

The complexity of the genre lies in the need to make a high level strategy to defeat ones opponent, while handling the dynamic part of controlling the army. The player needs to balance his attention in these areas in order to continually control every aspect of his gameplay.

A.3 Starcraft's success

The reason Starcraft was so successful is because it deviated from the normal way of making RTS games at the time. Previously, players had two races/factions to choose between. The problem was that these two races had different names, but equivalent creatures, with identical attributes. A good analogy is Chess, each player controls his pieces, but the only difference between the pieces of each player is the color, white or black. A white Queen has the same options as the black Queen when it comes to moving it. Starcraft deviated from this principle by using 3 races. Each of the races had very different features, and supported different strategies, depending on the opponent. The strength of this method came from the way Blizzard balanced each race, so that there were no single strategy that would work every time against a certain opponent race. While we have not found a study that supports this claim, there are still strong indications that it is true. Starcraft has a Pro-gaming league in South-Korea, and over the past 10 years they have had the best players in the world competing there. Any imbalances that has been discovered there has been quickly patched by Blizzard. This has been an ongoing process for 10 years, and we are confident that there exists few imbalances today.

A.4 Gameplay

Starcraft has 3 races that one can choose among. First, the “Terrans”, who are human colonists. They are the most versatile race, and provide a good mix of all-round infantry and mechanical military technologies, such as tanks and nuclear weapons. They also have personal cloaking (invisibility) abilities for selective units, and a high damage single target ability for their unit “Battlecruiser”. While there are many tactics for Terrans, they are particularly good at playing a defensive game. This is because they can completely block off entry points (called “Ramps”) to their main base, while providing a way through for their own forces by lifting production structures off the ground. Combined with the long range area-of-effect damage of their unit “Siege Tank”, a player would in most cases be forced to make aerial units to break their defences.

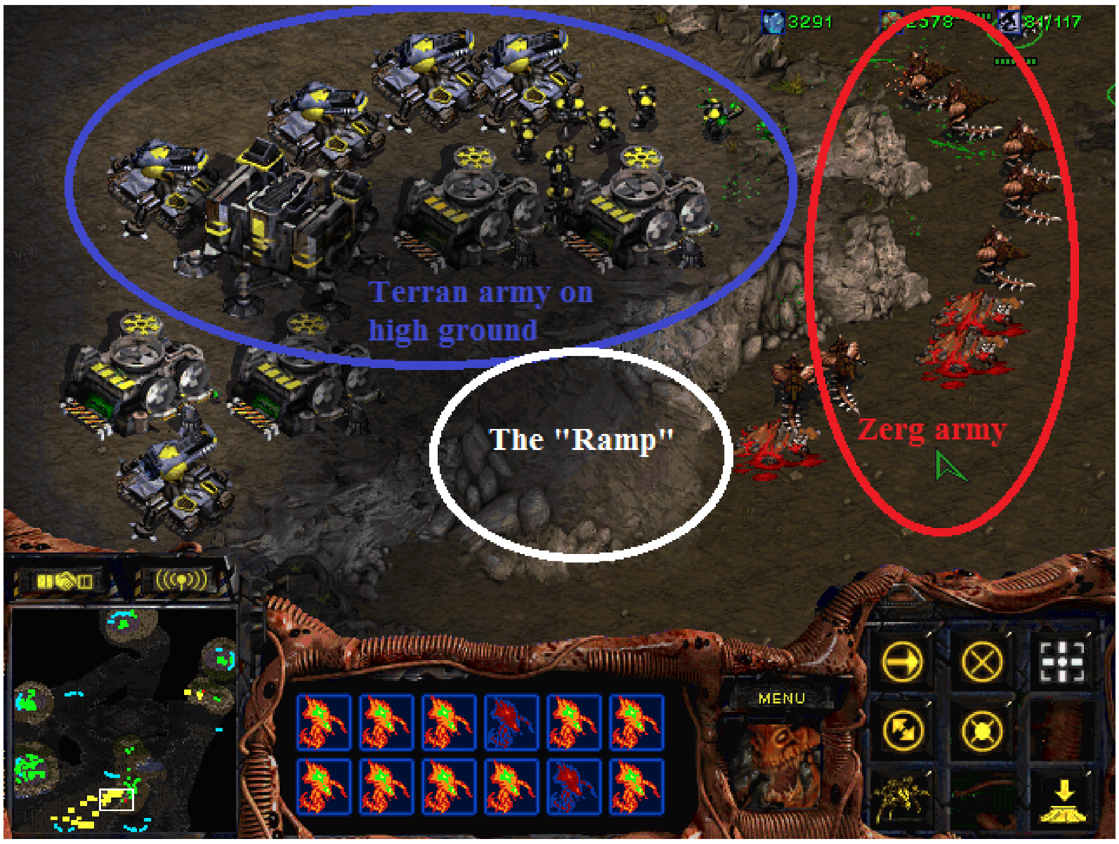

Figure A.1: **Terran holding high ground with tanks against Hydralisk**

Second is the psionic alien race called the “Protoss”. In general they have more expensive and more powerful units, but they have higher production time than the other races. They possess advanced technologies such as personal shields for all units and can warp in structures. This allows them to warp in many structures simultaneously with only one worker. In battle they have many exotic abilities and units, such as a "Psi Storm", which does massive area-of-effect damage, the permanently cloaked assassin "Dark Templar", to the massive flying unit "Carrier", which launches small fighter ships that does the fighting for it.

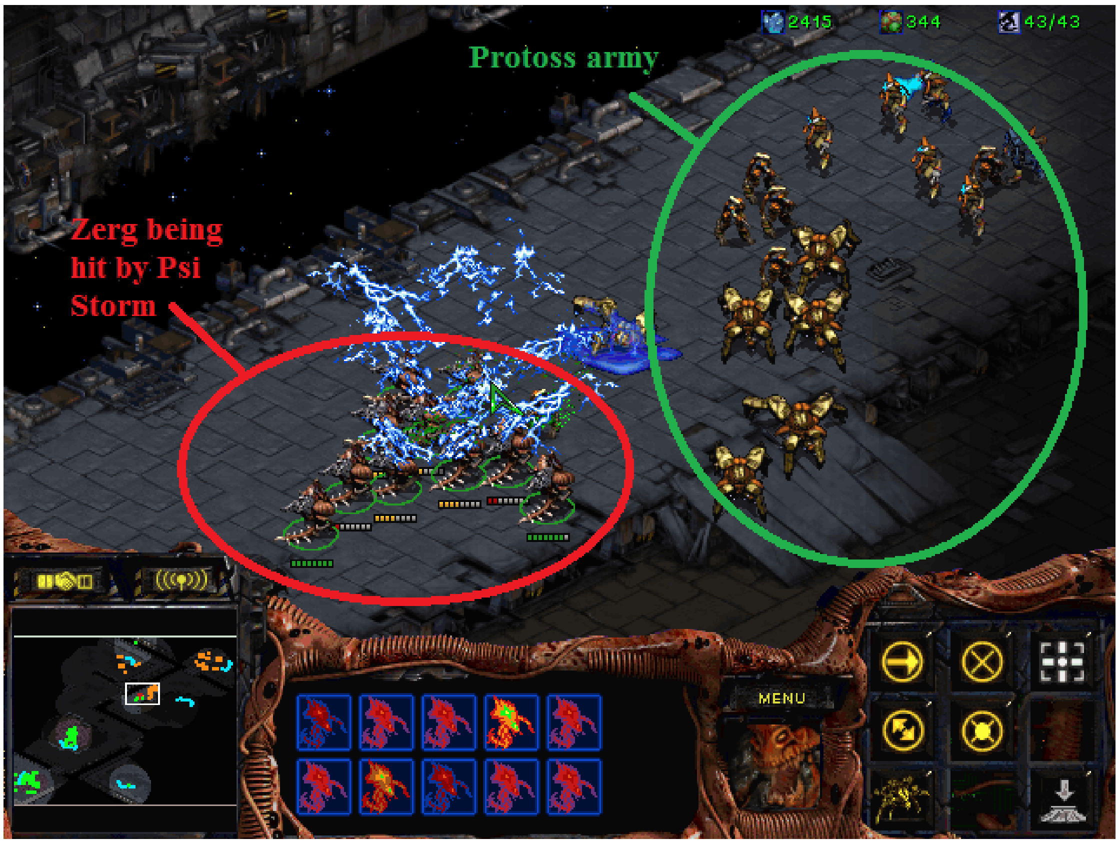

Figure A.2: **Protoss forces fighting a group of Hydralisk with Psi storm**

Last is the insectoid alien race "Zerg". They are composed of entirely organic structures, which consumes the worker that creates them. They are very different from the two others races, as all units can be produced from their main building, the hatchery. Units are not queued up and produced one by one at a normal production building, but they use a unit called "larva" that are generated automatically (up to a max of 3 per hatchery). A larva can morph itself into any unit the Zerg have access to create. They do need the unit specific building in order to be able to spawn each unit though. A player can then simultaneously make units at every hatchery (if he can afford it), and spawn an attack wave using the same time that it takes to make a single unit. Combined with cheap and expendable units this gives the Zerg a very offensive style of play, where an attack wave can be quickly followed up with another before the opponent is able to recover from the last. Units available include the small and agile “Zergling”, the all-round shooter “Hydralisk”, to the massive ground unit “Ultralisk”. Alternatively the player can make the highly agile flying unit, the “Mutalisk” for a hit-and-run tactic.

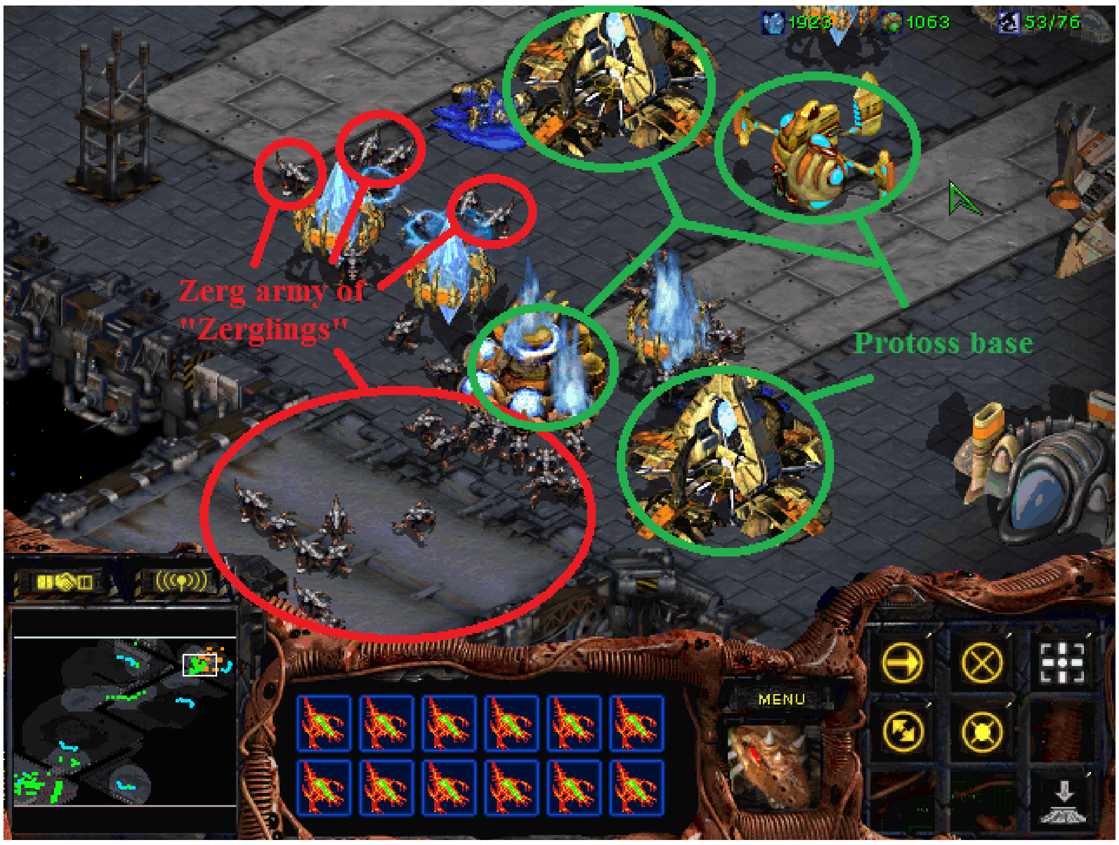

Figure A.3: **Zerglings overwhelming Protoss base**

A.5 Resource management

In Starcraft there are two resources that can be harvested. First there are minerals, which are required for every unit and building in the game. It can be obtained by harvesting it with a worker from the mineral fields that are scattered around the map. Second is vespene gas. This requires a special harvesting building to be built upon the gas node first, and can only be accessed by one worker at a time. In general resources are bundled together in groups of 8 mineral patches with one gas node. Every player starts at one such location, shown in figure A.4, and can make additional bases at other bundles.

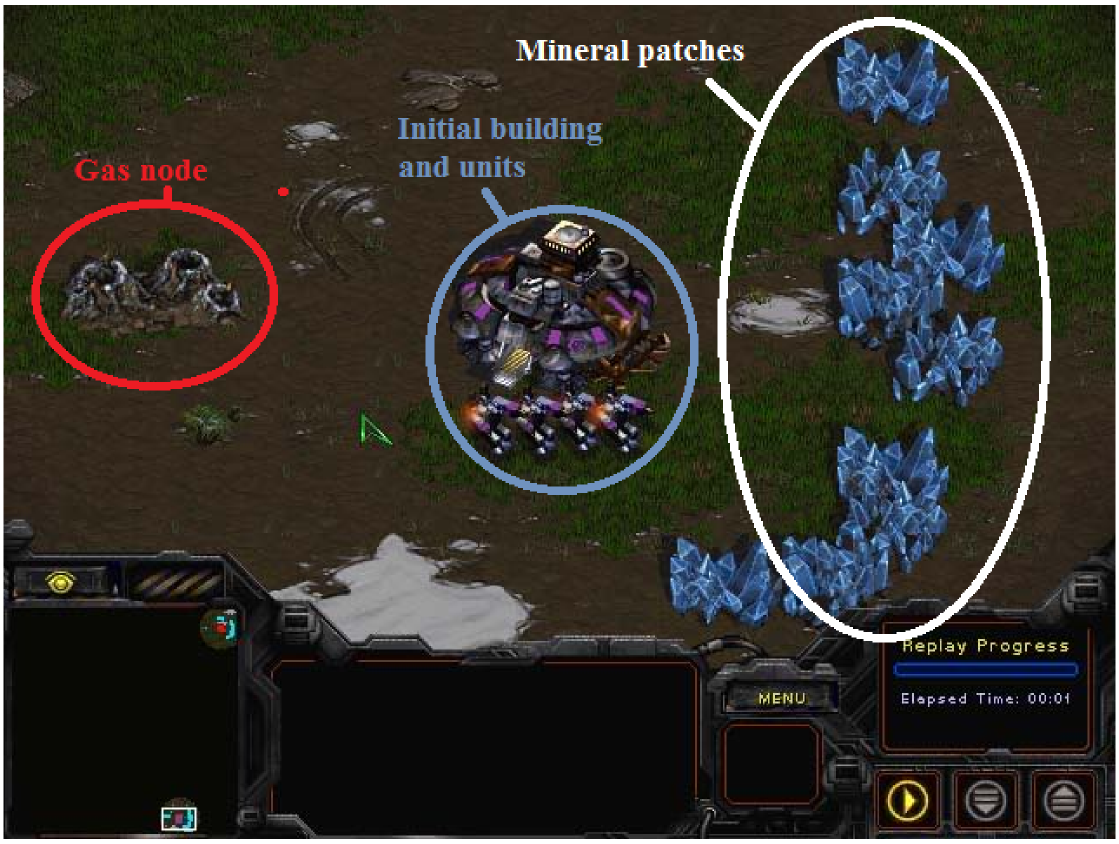

Figure A.4: **Initial starting location**

In addition, the player has a number of supply points. Each unit uses a fixed amount of supplies from 1/2 to 6 points. Once a unit dies, its supply points are returned. This is a mechanism to ensure that each unit type is balanced, powerful units require more supply points per unit than weaker units. It can be increased by making the special unit or building that increases it for each race. It is, however, limited at a total of 200 supplies, for example 40 units that use 5 supply points each, or 200 units that use 1 supply point each. This ensures that an agent must take care to avoid reaching this limit, as it cannot produce more units if it does. It also leads to certain strategies that involve destroying the enemy's supply buildings or units, in order to stop them from continually producing units.

A.6 High-level decision making in Starcraft

The high level decision making, or macro-management, is a very vital part of any agent. In Starcraft, one needs to predict the strategy of the other player, and plan to make buildings and units that somehow counters his strategy and ultimately leads to a victory. When starting a game, an agent has a selection of build-orders, similar to chess openings, each of which has pros and cons. A build order is the selection of the initial actions you want to perform. The agent can select anything from setting up a strong resource infrastructure to making an early tactical attack. There are no dominant strategies here, but the success of a build order depends on many factors, including the map, the opponents race and strategy. Since an agent cannot always know what the opponent will do (without sacrificing units early to scout), the environment offers a very complex decision making process which has to be managed.

A.7 Low-level army management in Starcraft

The need to manage individual troops and coordinating the army, or micro-management, is at the opposite end of macro-management. Micro-management refers to the detailed control of individual units of squads. For instance, the player can move a damaged unit out of the line of fire, or spreading a squad out to avoid area-of-effect damage. The agent needs to react in real-time to ensure that its decisions are being executed correctly, and change its strategy on-the-fly if the opponent makes an unexpected move. Pro-gaming has shown us that a player with good micro-management can overcome a foe with a more powerful army while sustaining minimal losses, and decide the outcome of the game that way. The agent has many different units available with different strengths and weaknesses, from the small and agile Zerglings, to the slow and powerful Battlecruiser.

A.8 Brood War API

Brood War API (BWAPI) is a free and open-source C++ framework which allows agents to control and retrieve information about nearly all aspects of SC:BW. Using BWAPI, programmers may test their AI techniques and perform research in a robust RTS environment.

Recently, tournaments for AI agents have been started, much like tournaments have existed for human players since SC:BW's inception. Several such tournaments are hosted by universities for the sole purpose of advancing AI research. Some examples include the annual AIIDE[1], organised by AAAI[2] and hosted at universities around the world, and CIG[3], organised by IEEE.

BWAPI is a flexible framework, and allows for many interesting aspects of AI research. For instance, BWAPI may limit information to what would be available to a human, posing challenges for the AI in opponent modelling, and requiring it to send out scouts to reveal the opponents intentions. Another possibility would be analysing replays by professional players to discover optimal build-orders or learning strategies.

A.9 Starcraft as an agent environment

The reason we chose Starcraft as our environment is because it is highly dynamic, combining the need to plan on a higher level (macro-management) with the more dynamic part of controlling the army in order to defeat the enemy (micro-management). In addition, the environment offers imperfect information by adding a "fog-of-war", which means it hides the information of the areas of the screen that the agent currently has no units or structures in. It is similar to the way a commander must have scouts that report to him about enemy movements in their sector. Without scouts he only knows how it was based on previous intel. We also have access to a free project, Brood War API, which gives us access to the same information and actions a human player has available. This allows us to focus on the creation of the agent itself. All in all, Starcraft offers an ideal test-bed to make complex intelligent agents which can be used to solve other similar complex problems.

[1]Artificial Intelligence In Digital Entertainment
[2]Association for Advancement of Artificial Intelligence
[3]Computational Intelligence in Games

Appendix B

Agent architecture

In this section we will describe our architecture. It consists of three layers which has separate responsibilities. They are the planning layer, which decides the high-level decisions, the executive layer, which generates behaviours to fulfill each plan item, and the reactive layer, which directly controls our units and structures via the API. For more information concerning our reasoning behind the choice of architecture, please read our pre-master project[22].

B.1 Planning layer

The planning layer is our top-most layer. It handles decisions in the most abstract form, which generally consists of a plan for the immediate future gameplay. Here is a list of the responsibilities this layer has:

- Plan generation
- Re-planning (exchanging plans at run-time)
- Opponent modelling

Plan generation is the primary task of this layer. A plan is a set of actions we want the agent to perform, for instance "train x amount of units of type y", where x and y can be an arbitrary value. The CBR module has the primary task of generating proper plans based on the current gamestate. A plan is generated when the planner requests it, and it has a certain format so we can change the modules when we please.

The task of re-planning is vital to any agent that wants to have a dynamic playing style. Consider the following event, an opponent builds aerial units, while your agent only has ground based attacks. This would allow the opponent to engage your army without the fear of retaliation, and would surely lead to defeat. The task of re-planning is to question a current plan, in light of new information, to see if it still is valid. Re-planning can be initiated by any significant event in the game game, e.g. if the agent is attacked, a current plan fails, or if the opponent model (explained below) differs from the observed actions of the opponent. In either case, the planner will have to decide whether the current plan is valid or not.

Opponent modelling is a very important for any agent[19]. The idea is that we predict how an opponent will play, based on our current information, and generate plans that will somehow counter that strategy. In our agent an opponent model is made by the CBR module. We will use a predicted opponent build order, which can be used when we are scouting the opponents base. If we fail to find an expected item, we need to find out where the opponent is using his resources. It can also be used to signal a significant event in the game, such as finding an unexpected unit or building when scouting.

Our planning layer communicates only with the layer directly below it. It will issue these types of messages:

- New plan
- Revised plan

A new plan is the most common message. We execute one plan at the time, so when the planning layer either has no plan, or the executive layer informs that the current plan is finished or failed, it will generate a new plan and pass it down.

As we mentioned previously, our agent can revise its plans. This is done if the current plan is believed to fail, in the light of recent information. The agent will then scrap the current plan and generate a new one which will handle the new situation. The message passed down will thus order the executive layer to scrap its current plan, and any active execution that has been issued from the previous plan, and start with the new plan.

B.2 Executive layer

The executive layer is our middle layer. It is responsible for executing the plans it receives from the planning layer in a sensible way, and will instantiate and track any behaviours that are created to fulfil a plan item. Here is the list of the responsibilities this layer has:

- Subdivide plans into items that correspond to actions
- Allocate resources and units which are requested in plan items
- Perform plan items in parallel wherever possible
- Track plan progress

The main task of this layer is to subdivide plans into items that correspond actions. This is a fairly simple task, as we have chosen a plan granularity that roughly corresponds to a one-to-one mapping between the two. The main task thus becomes to track the progress of a plan. In order to do this, the executive layer will divide a plan into an action, then create an appropriate behaviour that will fulfil this action and instantiate it. Some behaviours, like training units, can be considered done in the plan when the behaviour is created. We do not yet consider the even that the unit was cancelled or its production building was destroyed. Other behaviours, like creating buildings, requires the behaviour to send a success of failure message to the executive layer before we can consider it done. This is because other plan items, such as a building or training a unit, may require that previous building to be able to execute.

An important task of the executive layer is to allocate resources and units that are requested in plan items. The first and simplest case is to allocate resources. The agent knows at any time how many of its respective resources, minerals, gas and supply, it has available. The work is to plan to spend resources before instantiating a behaviour, else the resources could be used before while the behaviour runs. One example is a worker trying to build a building. It will require its resources to be available when it has finished moving to the location of the building site, else it will fail. Our solution is to flag resources as taken before instantiating a behaviour, and to release these resources when they have been spent in the game. The second case is when a plan item requires units. We have a unit manager that tracks all the units the agent owns, and whether they are occupied or not. In addition we have a harvest manager that controls all the workers that are mining minerals or gas. Most behaviours requires units, one example is attack behaviours. Attacks must have combat units to perform its attack, and the planning layer will have a suggested amount of units that had a certain effect in a similar attack. While the agent might have access to the units it is recommended to use, some units may be occupied in other vital tasks, such as scouting and base defence. The executive layer has to decide if and where it should deallocate units from, and then instantiate the attack behaviour.

In order to have even a resemblance of effective gameplay, the agent must work on tasks in parallel. Our initial observation was that the agent was always beaten badly by the built in AI, because it perform every task sequentially. This is a fairly complex task of the executive layer, since there are many events to keep track of. First, it must find dependencies in its current plan. In most cases, this is that a unit/building/upgrade requires another building in order to be built. One example is that the unit "Dragoon" requires not only its production building, but also an additional building the "Cybernetics core", in order for it to be unlocked. When the layer encounters such situations, it has to make sure that the previous items has finished before starting executing the next. Another important task is to plan ahead with resource management. If it is vital that the layer starts a dependent unit/building/upgrade immediately after its prerequisite building is complete, then it needs to make sure that other tasks does not spend too many resources while the prerequisite building is in construction. We do not distinguish between the importance of plan items, save that they come in a recommended execution order.

The final task is to track plan progress. This has been discussed some above, but the important part is to make sure that every plan item is executed correctly, and warn the planning layer when it has finished its current plan.

The executive layer communicates with the layers above and below it. It will issue these types of messages:

- Plan finished
- Instantiate behaviour
- Release units

The first message is the only message it will send to the planning layer. This message will signal the planning layer that the current plan has finished executing, and it is ready to receive a new one. We only have one plan at the time, so this message will make the planning layer remove its current plan and create a new one. We plan to add logic to handle concurrent plans, but this is a low priority.

The next message is to the reactive layer. It will order it to create a new behaviour of a certain type, with a certain amount of resources and units available. Examples include attack behaviours with a combat force, a build command with a worker assigned and the resources necessary, or a scouting command with its respective units.

The last message is usually sent just prior to the instantiate behaviour message. It will order the reactive layer to release the given units from the behaviour that controls them. This will happen every time an early scouting command is issued, which must use a worker that is currently harvesting. Other examples include deallocating defensive units to an attack group, or sending workers to a new base location to harvest.

B.3 Reactive layer

The lowermost layer is the reactive layer. It is the only layer that directly communicates with the game environment, via the API. Its primary task is to manage the behaviour tree/forest that is created to perform the tasks from the executive layer. Here is a list of the responsibilities this layer has:

- Create behaviours for the behaviour tree
- Report the status of behaviours as they finish

The primary task of the reactive layer is to create behaviours that will fulfil a task from the executive layer. Our approach gives us a lot of freedom when it comes to deciding how to perform a task. Each behaviour has only knowledge of itself and its children. This allows us to wrap entire behaviours with new logic, without the need to change the individual behaviours. One example is an attack command. While the only current behaviour for this command is an attack behaviour, we can easily add logic previous and after this to, for example, initially move to a rally point, and retreat to base when the attack behaviour is completed.

The second task is to report the status of behaviours as they finish. We have a single root node which will receive and store any messages from the behaviours lower in the tree. The reactive layer will only check the message buffer of this root node, and forward any messages to the executive layer. The way this is done is by adding a default logic which causes any behaviour to check the message buffers of its children, put any messages in its own buffer, and then delete the child's message. It is possible to override this method should the parent wish to add a more customised message to its own.

The reactive layer communicates only with the layer directly above it, the executive layer. Here are the messages it will issue:

- Behaviour status

The message concerning behaviour status is the only one we are sending at the moment. It will contain an integer which identifies which behaviour it belongs to, and whether the behaviour was successful or not. It is also possible to append additional information to the message. We currently do this when a ”Nexus” or an ”Assimilator” building has been completed. The first is the main building of our race, which signals that a new base has been constructed. The second is the unique building that needs to be created to harvest the resource vespene gas. Both of these buildings will require the harvest manager to allocate its own harvesters to the new resource location.