CHALMERS

Making and Acting on Predictions in StarCraft: Brood War

Bachelor of Science Thesis in Computer Science and Engineering

Henrik Alburg
Filip Brynfors
Florian Minges
Björn Persson Mattsson
Jakob Svensson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, June 2014

Making and Acting on Predictions in StarCraft: Brood War

Henrik Alburg
Filip Brynfors
Florian Minges
Björn Persson Mattsson
Jakob Svensson

Examiner: Jan Skansholm

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden June 2014

Abstract

Making predictions and strategic decisions is not just a problem in real life, but also in the complex environments of most real-time strategy games. Due to the lack of complete information, such games can be used as testbeds for AI research in this area. This thesis presents how an AI agent, in the game StarCraft, can make use of a Bayesian network to predict the state of the opposing player's technology tree. It is also shown how the predictions can be used to assist the AI agent in making strategic decisions. Bayesian networks are used for the predictions, and the agent's army composition is generated dynamically based on the predictions and the observed opponent units. The agent is tested against StarCraft's built-in AI. The results show that it is possible to accurately predict the state of the opponent's technology tree, and that the predictions have a positive effect on the AI agent's win rate.

Sammandrag

Att göra förutsägelser och fatta strategiska beslut är inte bara ett problem i vardagliga situationer, utan också i de komplexa miljöer som finns i många realtidsstrategispel. På grund av aspekter som exempelvis inkomplett information kan sådana spel användas som testmiljö för AI-forskning inom komplexa system. Denna kandidatrapport presenterar hur en AI-agent i spelet StarCraft kan använda Bayesiska nätverk för att förutsäga motståndarens teknologiträd. Rapporten redogör också för hur förutsägelser kan användas som underlag för beslutsfattande. Bayesiska nätverk används för förutsägelser, och AI-agentens armé anpassas dynamiskt baserat på förutsägelserna och de observationer som görs av motståndaren. Agenten testas mot den inbyggda AI som finns i StarCraft. Resultaten visar att det är möjligt att noggrannt förutsäga motståndarens teknologiträd och att dessa förutsägelser positivt påverkar matchernas utgång när de används i beslutsfattandet.

Acknowledgements

"There are only two types of people, those who can extrapolate from incomplete data."
- Unknown

We would like to thank our supervisor Andreas Abel for his support throughout the project. We would also like to thank the department of Language and Communication at Chalmers University of Technology for their continuous effort to improve our writing, as well as Astrid Hylén and Pär Persson Mattsson for their valuable feedback on the report.

The Authors, Gothenburg 2014-06-05

Contents

1 Introduction

Artificial Intelligence (AI) has reached the level where computers are able to beat top human players in games such as Chess. In 1997, the supercomputer Deep Blue beat the reigning world champion Garry Kasparov in a six-game match of Chess [1]. In order to win, Deep Blue made use of a problem solving technique called *brute-force* which systematically checks all possible solutions and picks the best one. However, solving problems by using brute-force is often impossible for intelligent agents in real-life situations due to the complex environments in which they act. The number of solutions to evaluate is simply too large even for most modern computers, and with increasingly complex environments the number grows even larger [2]. As a result, the situation might have changed before the AI is able to find the optimal, but no longer valid, solution. Further, complete information is not available in many real-life situations. A chess player will know for certain where the pieces are on the board, but the driver of a car can never be sure if there is an obstacle behind a house corner or not.

An AI designed to tackle more complex problems, such as those encountered in real life, must be able to handle incomplete knowledge. At the same time it must also use techniques that quickly generate functioning solutions before the situation has changed. In order to handle these requirements, many AI agents strive to find a near-optimal solution, one that is good enough for its purpose, instead of trying to find the absolute optimal solution.

When investigating different algorithms and methods for letting AI agents interact in complex environments, it is useful to have a platform where it is possible to test the different techniques. As an alternative to developing specialised simulators, some researchers look at modern games as a possible arena for testing the AI agents [3]. Games put players in situations where they need to make advanced decisions, often quickly and while dealing with limited resources.

One genre of game that focuses on long-term strategic planning while still forcing the player to constantly make short-term decisions is *Real-Time Strategy* (RTS). In RTS games, events happen continuously, as opposed to most board games which are turn-based. One such game, *StarCraft: Brood War*, is used in this project. The RTS genre and StarCraft is explained more thoroughly in section 2.1 and 2.2.

1.1 Background

The behaviours of AI agents are heavily affected by the architecture with which they are built. Some agents make use of scripts; lists of commands that are executed sequentially. Scripting is a method that is commonly used in the game industry since scripts are understandable, predictable, and easy to implement and extend [4]. However, scripts are static and are thus often easy for players to exploit. Other agents are modelled by finite state machines, where the behaviour is defined by predefined states which the agent can transition between (see Figure 1). Finite state machines are practical to use in cases where the behaviour is well-defined [5, 6]. However, they are also static in the sense that a finite state machine is only transitioning between predefined states. Yet another method is machine learning, the science of getting computers to act without being explicitly programmed. By applying machine learning, developers try to give the AI agent an instrument to learn from experience. One example is case-based reasoning, a technique which will be discussed more in detail in section 3.2. Many previous works have explored the implementation of case-based reasoning in RTS games, but very few have managed to create a successful AI agent executing the behaviour [7, 8].

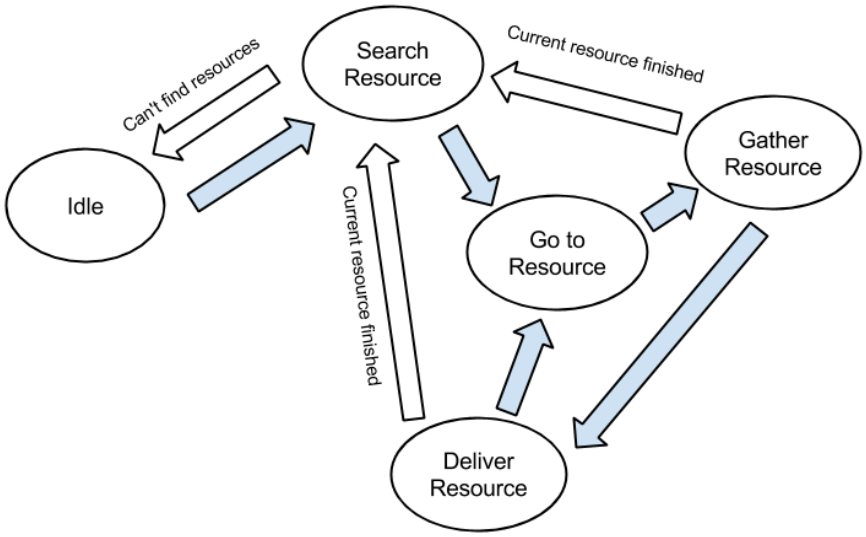

Figure 1. Example of a finite state machine. Graphics adapted from Martinez [9].

One problem in many RTS games is also that the AI agents cheat in one way or another. The goal of most game companies is to earn money by selling games, and it is often both easier and cheaper to tune the difficulty of the AI by letting it cheat in a controlled manner, rather than to spend development time on creating an AI that plays the game on the same terms as a human. The native AI in StarCraft, for example, has complete knowledge of the map and always knows where the player's units and buildings are situated [10]. During the recent years, many major game titles have also focused more on multiplayer modes, rather than a single-player mode [11]. In doing so, the players get matched against other humans instead of playing against AI agents. This has resulted in even less emphasis, from the game companies, on AI development. Although the commercial development of AI for RTS games has stagnated, it is a very active area of academical research [12].

According to a survey made on real-time strategy game AI research, some of the greatest challenges that an AI agent must be able to handle are [12]:

- **Resource management** - The gathering of resources, as well as determining what the resources should be invested in.
- **Decision-making under uncertainty** - An RTS game is a game of incomplete information; first of all because it is only partially observable, and secondly, because it is not possible to know for certain what the opponent is planning.
- **Spatial and temporal reasoning** - Spatial reasoning is a key regarding the placement of armies and buildings, since terrain may be used to optimise the chances of success. Temporal reasoning is necessary in order to know when to invest in e.g. building construction or upgrades, since the effect of these decisions are not experienced in the short term.
- **Opponent modelling and learning** - The ability of finding ways to exploit the opponent, as well as detecting when the opponent is exploiting oneself. Learning also includes

becoming better before, during or after a game.

- **Real-time planning** - The complex nature of RTS games and the large number of possible states means that standard planning algorithms cannot be directly applied to the problem.

Challenges like these can often be divided into three levels of abstraction: high-level strategical problems such as decision-making, planning and learning; mid-level tactical problems such as spatial reasoning; and low-level problems such as resource and unit management [12]. The focus of this project lies on the highest level of abstraction, strategy. In particular the challenges of making decisions under uncertainty, and learning and adapting to opponents.

1.2 Purpose

The purpose of this project is to develop an AI agent capable of playing the game StarCraft: Brood War. The project has two main goals:

1. The AI agent should be able to make accurate predictions of what the opponent is doing, based on observations of the opponent.

2. The AI agent should adapt its strategy based on predictions and observations of the opponent.

1.3 Scope of the project

As the goal of this project focuses on developing an AI able to make predictions and adapt to the opponent, an already existing AI agent will be used as a code base. By doing so, we focus on reaching the goal instead of spending time on developing basic but necessary functionalities like resource gathering and combat management. A more detailed description of the agent that we used as a base can be found in section 3.5.

There are three playable races in StarCraft, and in order to not let the scope of the project grow too large, the AI agent will only play as one of them; Protoss. This is because many of the gameplay mechanics and strategies differ between the races, and constructing an agent able to handle all three races would require more time than is available in this project.

2 Fundamentals of StarCraft

This section presents a description of real-time strategy games and some of the properties that make them good testing environments for AI development. Additionally, the specific environment used in this project, StarCraft, is explained in order for the reader to understand the critical concepts discussed in later parts of the report.

2.1 Real-time strategy games

Real-time strategy games are games where the player controls units, manages resources and finally creates an army to defeat his opponent [13]. As the genre name might suggest, these games are played in real time and decisions have to be made quickly since the state of the game changes continually. Every single decision may have lasting impact, and it is therefore important to make correct decisions from the start.

The different buildings, units, and technologies in RTS games can be seen as nodes in a tree-like graph. Buildings will often unlock more advanced buildings, which in turn allows the player to create stronger units, or units with unique abilities. There are also different kinds of *technologies*, or *upgrades*, that can give units special abilities, or just make them stronger or faster. As the game progresses more buildings will be unlocked and it will be possible to create a larger variety of units and buildings. Since this resembles the mathematical data structure called *tree*, it commonly goes under the name *technology tree*, or just *tech tree* (see Figure 2).

The worlds of RTS games are often partially observable, meaning that a player will only have vision in the near vicinity around his buildings and units. This obfuscation is called *fog of war* and brings uncertainty and an additional level of strategy into the games. Fog of war opens up for the possibility of surprising the opponent, by e.g. secretly preparing an army and attacking him.

RTS games are very complex, as they have a practically infinite search space. There can be millions of possible ways to reach a game state, such as a victory condition. The time is virtually continuous and every action affects the environment, further increasing the complexity. In addition, RTS games often have some element of probability causing the world to appear non-deterministic.

2.2 StarCraft

One of the most popular RTS games is *StarCraft*, and with the expansion pack *Brood War* the game has grown even more in popularity. Since its release in 1998, over 11 million copies have been sold worldwide [14]. In StarCraft the players can choose to play as each of three unique, but very well-balanced, races [15]. Since each race requires different types of strategies, the game has been versatile enough to become a well-received televised e-sport, and this is one of the primary reasons for the success of StarCraft. In South Korea, where competitive leagues and tournaments with prize money are being held, professional players are even able to make a living out of playing StarCraft [16].

What differentiated StarCraft from other RTS games at the time of its release was its different races [17]. The three races are called Protoss, Terran, and Zerg and though they are mechanically alike, they are played with different play styles. Zerg is a race of insect-like aliens, where units

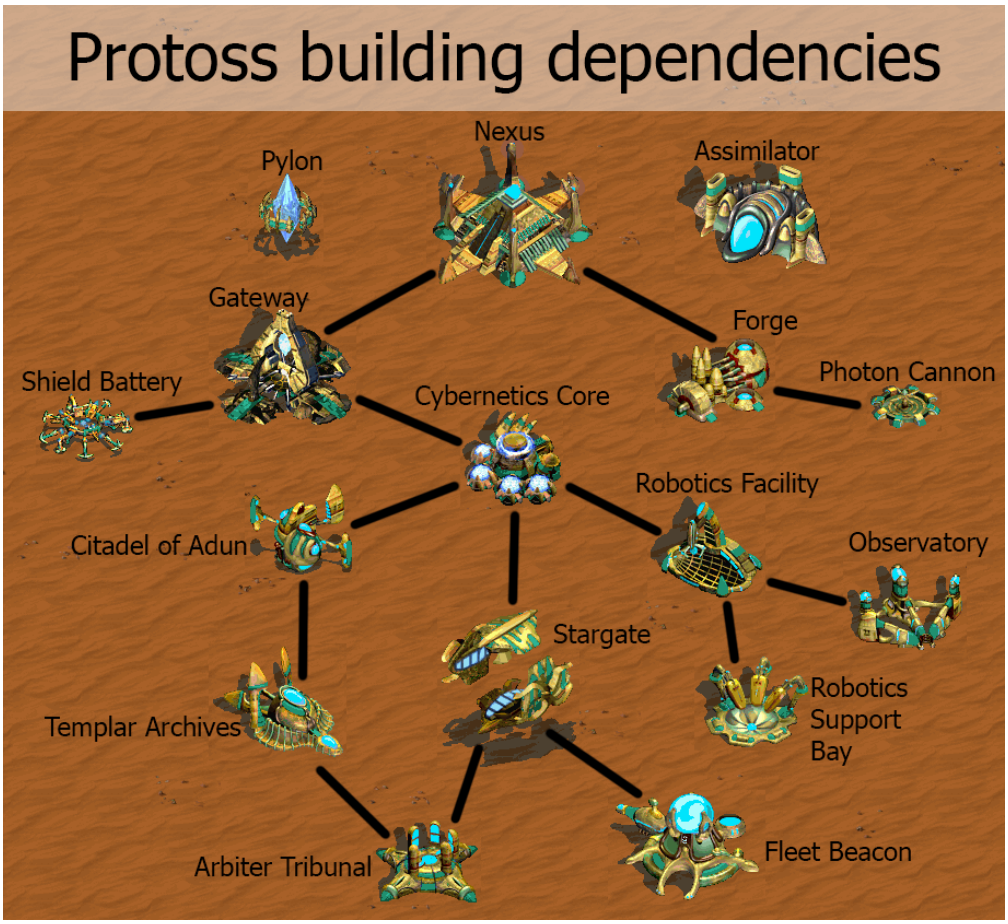

Figure 2. The building dependencies of the Protoss race form a tech tree. The Nexus, Pylon and Assimilator can be constructed at any time, while all other buildings have prerequisites.

are cheap and produced in masses. Protoss is another alien race that instead relies on advanced technology and fewer, but more powerful, units. Terran is the human race and has a play style that is something of a mix between the two other races; they combine infantry units with powerful tanks and starships in order to create strong defences. Even though the play styles of the races differ, the game is still considered very balanced.

Below is a list of important StarCraft concepts that might be needed in order to understand the report:

- **Races** - The three races are called Protoss, Terran and Zerg. They may also be referred to as P, T and Z.
- **Matchup** - A matchup is a certain race versus another, for example Protoss vs Terran (PvT). This makes a total of 6 different matchups; PvP, PvT, PvZ, TvT, TvZ and ZvZ.
- **Resources** - There are two types of resources in StarCraft; minerals and vespene gas. Resources are gathered by workers from mineral patches and vespene geysers, respectively. They take the role of money and are used to pay for units, buildings, and upgrades.

- **Workers** - Workers are a certain type of unit that are able to harvest resources and build buildings. Every race has one unit type that is a worker. Protoss has the unit Probe, Terran has the unit SCV and Zerg has the unit Drone.
- **Base building** - The base building is the building every race starts with. It is able to create workers and retrieve the resources which the workers have harvested.
- **Base** - A base is a location where the base buildings are usually placed. There are commonly eight mineral patches in a half circle called the mineral line and one vespene geyser close to a base.
- **Expand** - When a player is expanding it means that he is about to occupy a new base. A new base building will be created as close to the mineral patches as possible.
- **Supply** - Supply is needed to be able to create army units. Every unit has a supply cost, ranging from 1 to 6, and in order to create larger armies the amount of available supply must be increased. In order to increase the supply limit, certain supply-providing buildings or units (each race has a different supply provider) have to be built.
- **Scout** - To scout is to try to see what the opponent is doing. It is often associated with exploring the enemy's base in order to observe what types of buildings the opponent possesses.
- **Spell** - A special ability which can be used on units, either friendly or hostile, or be cast on the ground.
- **Spell-caster** - A unit which is able to cast spells.
- **Tech tree** - What buildings unlocks new buildings is often represented by a tree, which is referred to as the tech tree (see Figure 2).
- **Technology** - A technology is an added behaviour to a unit, such as a spell.
- **Upgrades** - An upgrade changes a property of a unit, such as making it faster, giving it greater shooting range or making its attack stronger.
- **Tech** - To tech is to unlock new units or buildings. This is often done by constructing buildings that unlocks new buildings or army units.
- **Build Order** - A build order is a sequence of buildings and units which should be built.
- **Opening** - Openings are build orders which are used in the beginning of the game before a player starts adapting to the opponent. They are often decided before the game starts.
- **Native AI** - The built-in AI that is shipped with the game.
- **Pylon** - A Pylon is a Protoss building which is required in order to construct other buildings. Each Protoss building (except from the base building and Pylons) needs power and must be close to a Pylon for it to function.

The objective of each game is to destroy all of the opponent's buildings. In order to create the stronger and better army, a player will need to take several things into consideration, starting from the very first second. A player starts with four workers and a base building (see Figure 3). Workers gather minerals which are essential to create more workers, new buildings and army units. At this point a player often follows predefined build orders, setting the tempo of the game.

Figure 3. The starting state of StarCraft: Brood War while playing as Protoss.

Resource management is one of the key concepts of every StarCraft game. Generally, resources can be invested in three areas; economy, tech and army units. A player might use the resources to create more workers so as to get more resources in the future, or he might invest in an army overwhelming the opponent before he can set up a proper defence. The third option is to construct buildings allowing the player to research some special technology or to create a special army unit, defeating the opponent with units that are stronger. These are the three main factors that must be balanced in a strategic manner, in order to put oneself in a good position to win the game.

A large amount of workers is the imperative for a strong economy and if a player can keep a better economy than the opposing player, he will generally win in the later parts of the game. Each base location will eventually run out of minerals, which means that the player has to *expand* to new base locations in order to keep a steady flow of mineral income throughout the game. Another reason to expand is if a player needs large amounts of gas, since the gas income from one base is quite low.

Although workers' responsibilities are to gather resources and construct buildings, they do have attacking capabilities. However, they are easily killed by army units. Thus, investing in an army early while the opponent focuses on creating worker units for his economy to thrive could result

in an annihilation of the enemy, leading to an early victory. If it is not successful, the attackers' economy will be behind and winning later in the game will be much more difficult.

If the economies of the two players are roughly the same, the more strategic aspect comes from the choice of technology buildings. Different technologies or army units are often weak or strong against other technologies or units. This means that it is crucial to focus on acquiring the technologies or units that are strong against what the opponent is getting. Often when a player follows a build order, the build order will end with the player unlocking some special tech. If the opponent can predict what that special tech is, it is possible to prepare a counter for it. By predicting and preparing for what the opponent is doing, a player can thus gain an advantage.

In the end, all these things need to be taken into consideration for every move a player does. There is no perfect strategy because the different paths in the tech tree are all well-balanced, and on top of that, a player can play more economically than his opponent to get an extra advantage. To win a game a player must therefore know what his opponent is going to do, using either predictions or assumptions, and react accordingly.

Of course, there is more to StarCraft than just making the correct strategic decisions. Aspects such as controlling the army, both individual units and larger army groups, are often a critical part of the game. Although these facets might not have a direct impact on the concepts of strategy and prediction, it is important to understand that a lot is happening in the background, which might alter the overall result.

3 AI development in StarCraft

This section discusses AI development in Starcraft and RTS games. The two AI techniques *Bayesian networks* and *case-based reasoning* are introduced, and some academic research papers exploring them are presented. These papers have been important for the direction of the project and also gives an understanding to what has been done in this particular field. In addition, we present some of the most successful and well-known agents from the competitive StarCraft AI scene. These were chosen, first of all for what they can add to the project, but also to give the reader an idea of how AI agents can function in StarCraft. Lastly, UAlbertaBot, the agent which was chosen as a base for our agent, is described in further detail.

3.1 Bayesian networks

A *Bayesian network* is a statistical model that makes use of Bayesian probability to represent dependencies between random variables. The network is a directed acyclic graph, where each node represents one of the variables. An edge from one node to another represents a direct dependency between the corresponding two variables, and knowledge about the state of one variable may propagate through the network to affect the knowledge about other dependent variables [18]. This is because each node has an underlying conditional probability table which, given the state of all dependent variables, specifies a probability distribution of the variable's value. Data for the tables can be gathered either by expert knowledge or observations [19]. By using data from observations, it is possible to generate a probability distribution that represents the actual situation.

An example of a small Bayesian network is depicted in Figure 4. In this example, the nodes 'Sprinkler', 'Rain' and 'Grass wet' are the three random variables. Without any information regarding the state of 'Grass wet', it is hard to accurately answer whether it has rained or not. However, if it is known that the grass is wet, the probability of 'Rain' changes. By using Bayesian networks to model directly dependent random variables, it is possible to use known information about some variables to gain more accurate probabilities about others.

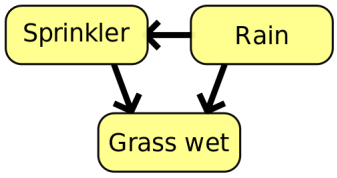

Figure 4. An example of a very simple Bayesian network. Wet grass depends on whether the sprinkler is active and whether it has rained. Rain also influences whether the sprinkler is active. These variables depend on each other and can be modelled using a Bayesian network.

3.2 Case-based reasoning

Case-based Reasoning (CBR) is an approach to problem-solving that does not rely on massive calculations, making it scale well into complex domains. Instead, CBR utilises experience, or specific knowledge from previously solved problems, in order to create solutions for new ones [20]. New problems are first matched against previously encountered, similar, problems called cases. The cases are stored in a case base which contains all experienced cases, and when a new problem is encountered and solved it is added to the case base. The idea originates from how humans reason when they encounter a new problem. An auto mechanic, for example, might

use case-based reasoning when repairing a car, by recalling how he restored a car with a similar problem in the past.

The problem-solving method CBR is divided into four processes called *retrieve*, *reuse*, *revise* and *retain* as illustrated in Figure 5. When a new problem is to be solved, one or more previously encountered cases similar to the new problem are retrieved. The solution of the previous case is then reused as a proposed solution for the new problem. The proposed solution is revised by testing the solution on the problem and evaluating the result. If the proposed solution failed, it is altered with the help of general knowledge about the problem domain. Lastly, useful information from the experience is retained for future reuse by updating the case base with a new case, or by modifying existing cases.

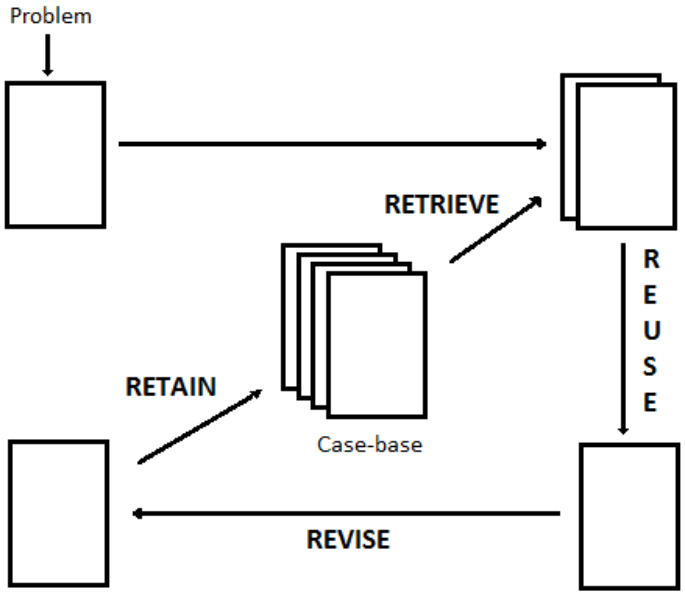

Figure 5. The case-based reasoning cycle. Graphics adapted from Aamodt and Plaza [20].

3.3 AI research in RTS games

Both Bayesian networks and case-based reasoning (explained in sections 3.1 and 3.2) have been used in various research projects. Below are summaries of the research papers that were the most inspiring and provided us with useful insights.

Synnaeve et al. used a Bayesian model to predict which opening their opponent was using [21]. Their technique could be used for build orders in general, but the focus was on the openings. Data mining was applied to a large set of replays, labelled with the different openings. Ten percent of every matchup were excluded from the data mining, and used as a cross-validation set. Synnaeve et al. reached good results with this method.

A Bayesian network was used by Oen to predict what the opponent was doing [22]. The network was updated between games and the results showed adaptation in the network, and that the network handled incomplete information better over time. The main conclusion was that the network did its job, though the agent itself did not use the information as well as intended.

The Čertický brothers showed how case-based reasoning can be used to create an army composition that is able to counter the opponent's army [23]. Their cases were composed of the opponent's army composition, upgrades and technology, while their solutions consisted of the player's unit composition, upgrades and technology. They created a database containing their cases and solutions, using a simple similarity function that compared the cases with their observations. The researchers noted the importance of scouting in order to adapt quickly, since the game is constantly changing.

Eriksson and Tornes showed how case-based reasoning may be used for the whole gameplay [7]. With a huge database of cases, they read the state of the game and picked the closest matching case. The cases were generated from almost 1000 replays, resulting in approximately 23 000 cases. They are able to show that CBR could be used with good performance and a large database, using an appropriate similarity function. However, their agent did not perform very well against the native AI. This is mostly because the functionality for executing the plans given through CBR was fairly limited.

Kasper van Mens discusses both the advantages and difficulties of using case-based reasoning in an RTS environment [24]. The difficulties of adaptation and evaluation in RTS games are pointed out; there are so many aspects to take into consideration, and it is hard to afterwards determine which of them went wrong. The report also reasons about automatically creating a case base instead of manually annotating replays.

These reports have applied both CBR and Bayesian networks with good results. However, when implementing the techniques in an agent and actually playing the game, the results have been far worse. This is probably related to the focus areas of the projects, which often have been to explore the techniques rather than to create good agents.

3.4 Existing StarCraft AI agents

While Blizzard Entertainment has not provided an official way of letting a program interact with the StarCraft gameplay, the free and open source C++ library named Brood War API (BWAPI) allows for precisely that. BWAPI gives programmers the exact same view of the game as a player in-game, meaning not all information is available and units can be given commands. This opens up for the possibility to write independent AI programs that interact with and plays StarCraft in a way similar to humans.

Since BWAPI's creation in 2009, AI development in StarCraft has gained popularity and several AI agents have been created. Some of these have been implemented as actual competitive agents while others have been used to test different AI techniques. There are also tournaments for these AI agents to compete in. The most notable is the *AIIDE StarCraft AI Competition*, which has been run since 2010 [12]. From 2011, only open-source agents are allowed to compete in the AIIDE tournament, and this has helped both newcomers and other competitors to develop their AI agents as it is possible to use another agent as a code base.

The majority of today's most successful AI agents use predefined strategies. Although they might choose between some strategies in a dynamic and smart way, the strategies themselves are still hard-coded. The agents apply different AI techniques, but these are seldom used for strategy. Instead they are used for areas such as combat and production, gameplay facets with more short-term goals. The agents are of course designed to beat each other, but not necessarily outsmart one another, and this makes strategy less of a priority. Below, we present some of the most well-known agents, most of which are affiliated with universities from around the world.

Skynet

Skynet [25] is one of the best StarCraft agents, and by far the best one that is not affiliated with a university [12, 26]. Like many other AI agents it plays Protoss. Skynet's strategies (hard-coded build orders) are flagged as either 'aggressive' or 'defensive', which makes it possible for the army to know whether it should attack or defend. It does not use any combat simulator, and instead relies on comparing the players' army supply and size to determine if it should attack or not.

Skynet has several openings that it chooses from. It is chosen depending on the result against the given player on the given map, trying out the different build orders before sticking with the winning ones. Skynet is able to tweak the hard coded build orders a bit by using prioritisation for resources, where e.g. upgrades all have a priority value.

What makes Skynet so successful is its good use of the Protoss units. It has very good control of both early-game units and more powerful units with abilities, available in the later stages of the game. This makes Skynet strong throughout the game. However, the agent has a hard time when it encounters a type of unit called *Dark Templar*. The Dark Templar is cloaked, and in order to attack it, one requires a detector unit to make the Dark Templar visible. This weakness indicates that Skynet have problems adapting to the opponent.

AIUR

AIUR [27] is an acronym that stands for *Artificial Intelligence Using Randomness*, and Aiur also happens to be the name of the Protoss homeworld. The agent tries to use randomness in order to not become predictable. It starts off by randomly choosing one of five moods, where the mood will determine the strategy. It also randomises different facets in these strategies but they are not dependent on what the opponent has done during the current game. The strategy used during a game is chosen at random, but the more successful strategies have a higher probability of being selected. This is similar to the behaviours of other agents, though Aiur's choice includes more randomness.

Berkeley Overmind

Some credit should be given to the winner of the first official StarCraft AI tournament; Overmind [28], created by University of California, Berkeley. Overmind always used the same strategy, which was to get a flying unit called Mutalisk and produce as many of it as possible. A Mutalisk is able to attack both ground and air units making it an exceptional choice if one had to pick only one unit to use. It is also fairly easy to control, since a flying unit does not take the terrain into account when moving, nor are they able to collide with each another.

Mutalisks are, however, not available in the early parts of the game. This forces Overmind

to defend itself using other units until the Mutalisks are available. The agent tries to defend itself with Zerglings, cheap melee units, and static defence buildings while saving up its resources so as to get the Mutalisks as quickly as possible. Overmind loses most of its matches in the early stages of the game. However, when the agent is able to start creating Mutalisks, the games are often in Overmind's favour.

Overmind's Mutalisks uses artificial potential fields for the movement, meaning that they create attractive and repulsive forces on targets and threats. This, along with a battle simulation where Overmind can calculate its own and the enemy's losses, made it perform well. The Mutalisks were able to take out key targets while still keeping themselves alive, harassing the opponent almost constantly.

Although Overmind has a very special strategy, it does not compete in tournaments any more. The main reason for this is the early rushes used by other agents. They often overwhelm Overmind before the Mutalisks are finished and since some other agents collect data on which openings are successful, they soon use rushing strategies every game, making it almost impossible for Overmind to create the Mutalisks.

UAlbertaBot

UAlbertaBot [29] was developed by David Churchill, a Ph.D. student at the University of Alberta, and the agent emerged as the winner in the 4th annual AIIDE StarCraft AI Competition [26]. UAlbertaBot has a modular design that makes it easy to modify parts of the AI, and thanks to this, UAlbertaBot is also used as the code base for a game programming course in which the students create their own AI agents [30]. The agent's default tournament behaviour is to rush the opponent as early as possible and win the match before the opponent becomes a threat. This behaviour is possible since UAlbertaBot uses a combat simulator to determine whether a battle will be won or lost, and the AI will only attack when the agent is certain that it will win the battle. In addition, the agent makes use of a module that, given a production goal, searches for a build order that reaches that goal.

Regarding the strategic decisions, UAlbertaBot is not very dynamic. The agent scouts the opponent early on to find its enemy position, and as soon as the first military unit is trained UAlbertaBot starts attacking. By constantly building more units and expanding to new bases when needed, the agent can reinforce its attack until the opponent is destroyed. In the few cases where the opponent survives this early rush and starts attacking back, it is difficult for UAlbertaBot to handle the situation. Since UAlbertaBot is optimised to win the game as early as possible, its management of early-game units is very good while the management of the more advanced late-game units is unoptimised. The AI has the lowest average game time of all the competitive bots, mostly because of the early rushes and its effectiveness in combat [26].

3.5 The design of UAlbertaBot

As can be seen in section 3.4, there are many agents with varying quality, both regarding gameplay and design. In order to base this project upon an existing agent, it must have a design which makes it possible to modify the strategic behaviour. After considering several agents, UAlbertaBot was chosen because of its modular and extensible design, as well as its documentation. Therefore, the structure of this agent is explained more in detail.

UAlbertaBot makes use of a sequencing design pattern called *game loop*. Every frame, a number of different classes are updated by calling their update methods. A class diagram of the different classes can be seen in Figure 6. This design makes it easy to add rule-based behaviour in the appropriate update method of any given class.

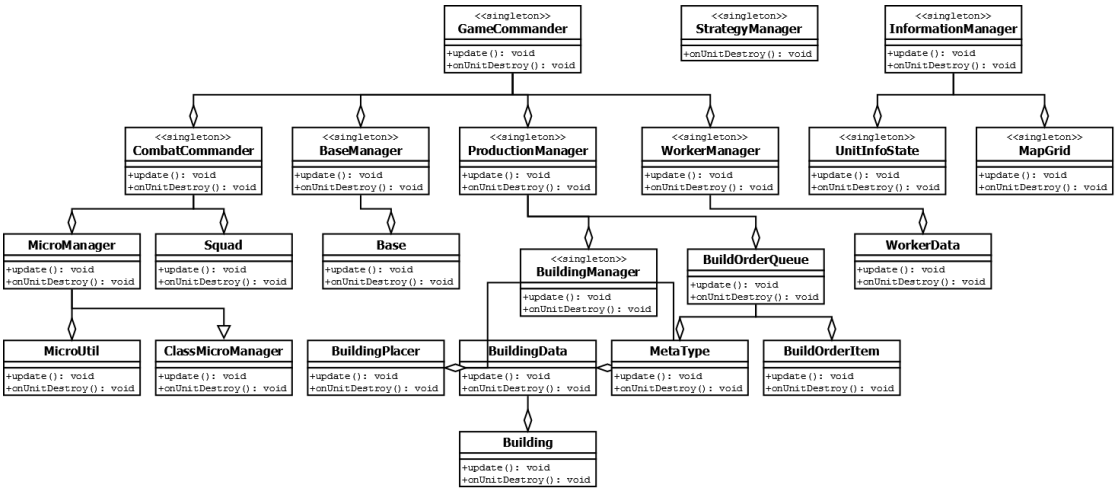

Figure 6. Class diagram of UAlbertaBot.

The structure is built around the two concepts *micro* and *macro*. Micro is the control of units; everything from telling the army to attack, to sending the first scouting unit. The main loop calls a class called *CombatCommander*, which main purpose is to assign all non-worker units to attacking, defending or scouting tasks. Each unit is assigned to managers which handle more specific actions, such as attacking, moving, and positioning the unit. Some of the unit managers found in UAlbertaBot are *ScoutManager*, *RangedManager* and *MeleeManager*. These managers have a rule-based behaviour that will execute any order coming from CombatCommander, and delegate the orders to the appropriate units.

The other concept, macro, includes everything from handling the collection of minerals, to following build orders and placing buildings at appropriate places. The main classes that handle macro are called *WorkerManager*, *ProductionManager* and *BuildingManager*. The WorkerManager makes sure that all workers are allocated to tasks, such as constructing buildings, mining minerals, or harvesting gas. The ProductionManager supervises the build order and delegates the creation of items from the build order to the appropriate classes, and is also responsible for finding a new build order once the old one is finished. The BuildingManager handles building-related tasks, such as placing buildings without blocking paths.

Two managers are not called in the main update loop; *InformationManager* and *StrategyManager*. Both of them are global, which means that every other manager might access them or take orders from them. InformationManager saves necessary data that the other managers might need, mostly data related to enemy units and bases. The StrategyManager's primary task is to generate build-order goals, and to some extent also decide if it is time to attack.

In the start of a game, UAlbertaBot uses one of three predefined build orders. These are early-

game oriented and are inspired by build orders of professional players. UAlbertaBot chooses build orders based on how effective they have previously been against the specific opponent [31]. After the initial build order, the agent continues with a goal-driven behaviour that determines what units to make. This leads to a very simple behaviour where UAlbertaBot creates the same types of units, given by the goal, over and over again.

The module that UAlbertaBot uses to search for build orders is called BuildOrderSearch (BOS). BOS takes a starting state as input, and then uses a heuristic depth-first search algorithm on the starting state until a state that fulfils the goal is found [32]. The algorithm is an *anytime algorithm* and can therefore be halted at any time to return the best solution found so far, or to continue the search at a later time. This allows BOS to search for longer durations without interrupting the game. The search is efficient for smaller build goals, but takes increasingly more processing power when several unique unit types are involved.

The combat simulator used by UAlbertaBot is an external project called Sparcraft [33]. Sparcraft is a scaled-down combat simulator which is built to be very fast. Given any situation, the simulator returns either 'success' or 'defeat', making it very easy to determine when to attack and when to fall back. Unfortunately, the simulator excludes some units, such as the very important spell-caster units. The reason for this exclusion is that these types of units are not controlled with the regular attack command, but rather with spells. This sometimes results in bad predictions, where a defeat will be predicted for a battle which could have been won.

UAlbertaBot uses its own pathfinding for units instead of relying on the pathfinding provided by the game engine. This, along with the rule-based unit micromanagement that UAlbertaBot uses for all its units, often outmaneuvers the opponent. These techniques work well for the units normally used by UAlbertaBot, but they do not work well for all unit types. This makes the behaviour undefined for more advanced units.

4 Method

This section describes how we created our agent. We start by presenting how UAlbertaBot was modified to fit with the project's needs, and continue with the new modules for predictions and decision-making. Lastly, we introduce a separate agent used for testing. The tools that were used during the project are briefly described in Appendix A.

4.1 Modifications of UAlbertaBot

When creating our own agent, we based it on UAlbertaBot, and named it Odin after the wisest of the Norse gods. Although UAlbertaBot already was a complete bot and had many desired features, there were several parts of the agent which needed to be modified in order for it to work with our goals.

As one of the goals of the project was to adapt the strategy based on what the enemy does, Odin needed to be able to manage a large variety of units, not just the few early-game units that UAlbertaBot handles well. To solve this issue, separate manager classes were added for the spell-casting unit *High Templar* as well as for the scouting unit *Observer*. The High Templar manager is responsible for letting High Templars use the spell *Psionic Storm*, summoning lightning bolts that deal damage to all units in a small area, in a controllable manner. Friendly units should be unharmed while still dealing as much damage as possible to enemy units. The objectives of using Observers are to reveal cloaked enemy units and to scout the enemy. To achieve this, the manager for Observers handles the Observers in a manner where one always travels around to scout the enemy bases. The remaining Observers try to locate and follow the enemy army, exposing any invisible units in it.

Special behaviour for *Reavers* and *Carriers* was defined within the generic manager for ranged units. These two units must be handled in an exceptional way because they deploy the "sub-units" *Scarabs* and *Interceptors* in order to attack. The special behaviour includes positioning and attacking, as well as building the Scarabs and Interceptors. The default behaviour for ranged units is used for the rest of the Reavers' and Carriers' behaviour. In addition, the way in which a worker scouts the enemy base in the beginning of a game was changed slightly, so that it explores more of the enemy base before going to harass the enemy worker units.

On a higher level, the way in which strategies are chosen was also changed. Several openings were added to the collection of openings used by Odin in the beginning of a game, and are now optimised against specific races instead of players. The opening build orders used by Odin are listed below, and a detailed description of them can be found in Appendix D.

- **Zealot Rush** - A build order optimised to get out an early Zealot attack to win the game before the opponent has time to react. This is the default opening used by UAlbertaBot in competitions and the only one not specifically created for Odin.

- **Gateway Fast Expand** - An opening designed to be both economical and aggressive, by allowing the player to quickly expand to a new base while still having some military safety.

- **Nexus First** - A very greedy build order giving a player economical advantage but making the player vulnerable to early rushes.

- **Arbiter** - A build order with the goal of getting out Arbiters and a small army of Zealots to catch the opponent off guard with cloaked army units.

- **Forge Fast Expand** - An economic opening that still allows for some safety against most early rushes. Being called the "Bisu Build" after one of the best Protoss players of all time, it is mostly used against Zerg opponents.

- **Two Gate Observer** - A build with the goal of giving the player the possibility to scout the enemy or see cloaked units.

- **Two Gate DT** - A build that punishes players who have not teched for detectors, by quickly attacking the enemy with two cloaked Dark Templars.

- **Gate Core Expand** - A build order that, instead of focusing on the economy, opens up for quickly teching to more advanced units and buildings.

- **Two Base Carrier** - A build with the goal of getting out Carriers, a very powerful Protoss unit, as quickly as possible. This opening is easily countered if spotted early.

The set of predefined build order goals was removed to make place for a special goal which adapts to the enemy situation, called *Army Composition Generator* in this project. This goal is explained further in section 4.3. The adapting goal tended to be larger than predefined ones, resulting in performance issues with the search algorithm for build orders. Because of this issue, the behaviour was also changed so that the goal did not have to be re-searched every time one of the agent's buildings or workers were destroyed. The original behaviour sometimes caused too frequent searches, with the result that no search had the time to return a build order.

The manner in which the algorithm is used was also changed. The algorithm now starts searching for a build order as usual, but if it takes too long time to find one, it moves units that the agent is already able to create directly to the build queue. If this does not help, the size of the goal is reduced before trying to search again. If the same search has timed out in this way three times, it is restarted with a completely new goal. At the same time, construction of Probes and Dragoons are started from each Nexus and Gateway respectively. This is done in order to not just stay idle while the new search is being carried out.

Checks for deadlocks were also added in order to avoid the production getting stuck for too long. Such deadlocks can happen when the BuildingPlacer is unable to find a place for a building. If the building is a prerequisite for the next item in the build queue, the production becomes stuck. To prevent this from happening, Odin checks if there is a problem finding a place for a building every 240th frame, and if so, a Pylon is added to the top of the build queue. Since the problem of finding a place for a building mostly happens when there are too few Pylons, building a new Pylon often solves the problem.

In order to further react to the situation, a system with an attack state and a defend state was created. The state which Odin is in depends on several parameters, including the economy, the army strength, and the defending capabilities of both the enemy and Odin. The behaviour

of Odin depends on the state he is in and is described further in section 4.3.

A feature that UAlbertaBot lacked completely was the ability to analyse replays. Thus, a module for gathering data from replays was created. The collected data is saved to a file at the end of each replay, making it possible to use it in other parts of Odin. How the data is used is described further in section 4.2.

4.2 Implementing predictions

Predicting the opponent's current and future behaviour is, in StarCraft, a problem of incomplete information. Most likely, a player will not know everything about the enemy, and this makes it impossible to accurately model the opponent only from what can be observed. Under the assumption that the player does not know for sure what the opponent will do, a precise prediction of the opponent's future state is impossible as well. By using a probabilistic model it is possible to model this uncertainty in a better way, and therefore it was decided that we use Bayesian networks for the task. A description of Bayesian networks can be found in section 3.1.

Design and usage of the Bayesian networks

The tech trees of the races were used as templates for the Bayesian networks, and since there are three races the result was three Bayesian networks. Each network had the basic structure of the corresponding race's tech tree (see Figure 2), with a few alterations. The networks were comprised by nodes of not only the buildings, but also the units and a node representing the game time (see Figure 7). The purpose of this network design is to calculate the probability that the opponent is in possession of a specific building or unit, given the observations that have been made so far and the amount of game time that has passed. Because there are three different networks, one for each race, the correct network must be loaded at the start of the game. In the cases where the opponent's race is Random, the network cannot be loaded until the enemy has been found and its race is determined.

In order to not make the network too complex each building and unit node has two possible values; 'absent' or 'present'. If a unit or building has been observed, the corresponding node's value should be set to 'present'. The node modelling the game time is called *TimePeriod*, and has 25 possible values. The time in StarCraft is measured in frames, and when the game is played in normal speed, one second is 24 frames. The first value, 'period01', represents that the time is in the interval 0-999 frames, the second value, 'period02', represents that it is in the interval 1000-1999 frames, and so on. The last value, 'period25', is a special case and represents that the game time is over 24 000 frames. The game time is an important factor in predicting what the opponent might be doing, since more advanced units are unlocked as the time goes on. The probabilities for late-game units to be present should therefore be low during the early parts of the game, to later increase.

The Bayesian network is used by setting the value of the TimePeriod node, and setting the nodes for all observed enemy units to the value 'present'. After the network has been updated with this information, the likelihood of the enemy possessing a unit or building is the probability that the value of the corresponding node has the value 'present'. In order to get as accurate values as possible, the network should be filled with up-to-date information about what buildings and units the enemy owns, and this means that scouting the enemy for information is important.

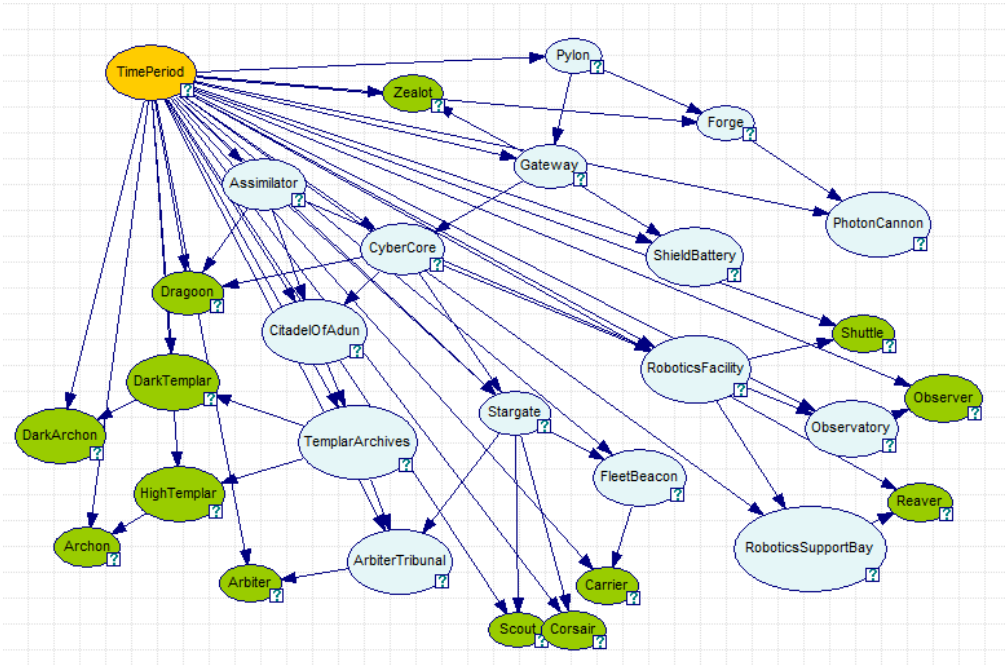

Figure 7. The Bayesian network used for the Protoss race.

Implementing the networks

The implementation of the Bayesian networks was done in several steps. In the first step, a program called GeNIe was used (a description of GeNIe can be found in section A) to graphically design the networks. Since GeNIe stores the Bayesian networks as XML files, it is easy for the AI agent to parse the information in the networks during runtime.

Initially, the Bayesian networks contained no data, so the second step consisted of collecting the data that was to be used in the networks. This was accomplished by performing data mining on StarCraft replays. Two sets of replays were used; replays from matches played by human professional players, and replays from matches played by the native AI. The human replays were downloaded from an e-sports community site called TeamLiquid [34], well known for their team of professional StarCraft players, while the native AI replays were recorded during the development of Odin. Odin was configured to collect and store data whenever it runs a replay, and was set to the task of gathering data from the replays. The data consisted of the units and buildings that had been constructed before each time period had ended. Since Odin exclusively plays as Protoss, only replays where one of the players played as Protoss were used, and the collected data only concerned the opposing player. That is, no Protoss data were gathered from PvT or PvZ matches. Of the 442 human professional replays that were used, 64 were PvP, 228 were PvT and 198 were PvZ. Of the 387 native AI replays that were used, 196 were PvP, 114 were PvT and 77 were PvZ. The collected data were converted to probabilities and loaded into the Bayesian networks by using GeNIe.

In order to let the AI agent use the Bayesian networks, they had to be loaded and updated at runtime. This was achieved by using a general purpose C++ library called Dlib, which pro-

vided support for reading, writing and updating Bayesian networks [35]. However, Dlib does not have support for reading the xml files generated by GeNIe, and a parser had to be created. Once the parser was done the Bayesian networks could be used by the AI agent.

4.3 Implementing strategic decision-making

Only having well-estimated information does not improve the behaviour of the agent in any manner, unless it is used appropriately. Thus, from the previously gathered and predicted data, strategical decisions have to be made in order to gain advantages over the opponent and ultimately to win the game.

Army composition generator

In order to be able to create an army that counters the opponent's army, decisions regarding what units to create must be made. For this purpose, the *Army Composition Generator* (ACG) was created. As the enemy situation is not fully known, the ACG puts together an estimation of the enemy's army by looking at the units that actually have been observed, as well as the units which the enemy likely has. Predictions of the latter are made using the Bayesian networks.

In order to make the ACG easy to tweak and optimise, two constants were introduced. The first constant, the *unitThreshold*, is a value that the prediction of a unit must exceed, before the ACG considers the unit. The second constant, the *unitStartValue*, is the weight with which a predicted unit is counted in the ACG. A weight of 1 would mean that the prediction is worth the same as an observation, and a weight of 0.1 would mean that the prediction is weighted as 10 % of an observation.

The estimated list of units in the enemy army is then used to generate a list of units that counters the enemy army. The counters are predefined, and a list of all counters used by Odin can be found in Appendix C. Each unit has two different counters, one which is cheap and easy to tech for, and one more expensive and farther into the tech tree. The end result of the ACG is then used to create a build order goal that holds units that counter the enemy army. The expensive counter unit is only added into the goal if it is teched for already, in any other case, the cheap counter unit is added. However, if the economy allows for it, then the prerequisites needed for the expensive unit is added to the goal to prepare for the future.

Attack and defend states

An attack state and a defend state was added for Odin to further react to the situation on a higher level. Odin's behaviour changes depending on which state he is in. When in the attack state, Odin builds units according to the army composition generator and attacks the enemy continuously. However, when in the defend state, Odin prepares for the enemy to attack by placing units within its own base. In addition to building units according to the ACG as in the attack state, Odin also builds *Photon Cannons* to defend the base.

Which state Odin is in depends on the army, defence and economic potentials of the two players. Each of these potentials are calculated from several factors, such as how far into the tech tree the player has reached, the size of the army, the number of buildings that can produce army units, the number of defensive buildings and the number of expansions that the player has. Whenever the army potential of the enemy is greater than Odin's army potential, Odin enters the defend state in order to survive until the army is powerful enough to deal with the enemy.

In order to account for possibly missing or outdated information, an uncertainty factor is used to modify the enemy's army potential. This factor is simply a value that the opponent's potential value is multiplied by, before comparing it with Odin's potential value. The usage of this uncertainty factor affects Odin's performance, in that the agent plays more aggressively with lower values of the uncertainty factor, and more defensively with greater values. A value of 1 makes Odin estimate the enemy's army potential to be precisely what has been observed (thus not modifying the potential at all), a value less than 1 would make Odin underestimate the opponent, and a value greater than 1 forces Odin to overestimate the opponent's army potential.

Case-based reasoning used for decision-making

To handle the decision-making of the agent, we intended to use case-based reasoning. The problem to solve was determining what the agent should do in the different situations that might appear during a game of StarCraft. A case was to be defined as a tuple containing the different army, economy, and defence potential values, as well as the player's and the opponent's army compositions. In this approach, a solution would be a sequence of actions of what Odin should do when encountering a situation, and the CBR would be responsible for generating this sequence. Depending on whether Odin is in the attack or the defend state, the solution should reach different goals. The goal of the attack state is for example to attack the opponent, and so the solution should specify how the attack is to be carried out.

However, this way of handling decisions was not implemented in Odin. The reason behind this is mainly the way in which UAlbertaBot is designed. UAlbertaBot is constantly executing update methods throughout the game, and it acts based on the current state of the game and the different events that occur. This design makes it difficult to implement the actions that were to be sequentially executed, without completely redesigning the agent. Creating a completely new structure for Odin would then undo all the benefits of using an existing agent as a start, and would thus take too much time to fit in the scope of this project.

4.4 Build Order Bot

To test if Odin successfully fulfilled the project goal, a test environment was set up. A test environment that is capable of this needs to have the property of reproducibility, meaning that it must be possible to reproduce the same conditions and presets in order to verify any results. A key to accomplish this is to use the same map and the same opponent every time. The problem with this approach is that a good AI agent needs to be unpredictable, while a test environment needs a predictable one. To solve this problem a *Build Order Bot* (BOB) was created.

BOB is a modified version of UAlbertaBot which can follow a set of attack orders. An attack order consists of a time represented by number of frames, a set of unit types and the amount of every unit type that should be sent (see Figure 8). When BOB has an attack order, it will not attack with any units until both the required number of units of each type is available and the timing is right. When BOB has no attack orders left it sends units in a continuous flow to attack the enemy. This feature lets BOB reproduce similar behaviour in multiple games, while UAlbertaBot and other AI agents often can have more varying attack timings. BOB is based on UAlbertaBot, meaning that the underlying mechanics of Odin and BOB are equivalent. This property is also very important when testing the performance of Odin, since the two bots will control the army in the same way.

```
8000
6
4
13000
6 22
5 3
```

Figure 8. An example of two attack orders; one where 4 units of type 6 (Dragoons) should attack the enemy after 8000 frames, and one where 5 units of type 6 and 3 units of type 22 (Dark Templars) should attack after 13 000 frames.

To further promote reproducibility, BOB also relies on longer predefined opening strategies. Instead of just covering the opening, they actually cover most of the game. To avoid the build order from changing to something else when critical units dies, and thus altering the game play, BOB tries to rebuild all critical units as soon as they are destroyed. By this it can potentially continue to play in the same way every game, despite how the opponent may interrupt it. It also mitigates the problems that may arise when searching for Build Orders with UAlbertaBot, as described earlier in this report.

5 Tests

In this section we explain how the goals of the project were tested and we describe the test results. There are separate tests and results for the prediction and strategic parts. All tests are done using real StarCraft games, the maps used can be found in Appendix B. To determine when a match is over, tournament rules were used; a player has won if all of the opponent's buildings are destroyed, but if the time has gone past 86 400 frames (one hour) the in-game score is used to determine the winner [36]. AI agents may also not slow down the game by spending too much time per frame on calculations. The games were played on the fastest possible speed meaning that a single game would take between 4 minutes up to the hour. Most games were played under supervision.

5.1 Predictions

To test the accuracy of the predictions Odin can make with the help of its Bayesian networks, 39 games were played against the native AI agent. The native AI agent's race was set to be chosen at random, which resulted in 16 games against Terran, 13 against Zerg and 10 against Protoss. The predictions of the Bayesian networks were logged every 1000 frames until frame 25 000. No data after frame 25 000 was logged, since the network only contains 25 time periods (as described in section 4.2). The games were saved as replays, allowing for the logged data from the network to be compared to the actual game state. Because the network deals with probabilities, it is also necessary to define a threshold for what counts as a correct prediction. An initial threshold was set to 75 %. A prediction of the state of a node in the Bayesian network is thus only considered accurate if the prediction determines the node's correct value with a certainty of 75 % or more. The obtained results are plotted in Figure 9, 10, 11, 12. The exact data can be found in Appendix E.

Figure 9 shows the prediction accuracy of nodes for the current time period. We also tested if we could use our Bayesian network to predict what kinds of units were present in future time periods. This was accomplished by setting the time period node to a value ahead in time and logging the Bayesian network throughout the game. This was done for up to 6 time periods ahead. While not all results are included in the graph, a comparison of prediction accuracy for the current time period, 3 time periods ahead, and 6 time periods ahead can be seen in Figure 10. The results shown are a generic summary of predictions in StarCraft, where the prediction accuracies against the different races were weighted equally.

Since the native AI does not play in the same way as professional StarCraft players, we also set up a similar test using data from the native AI. In the test, we first trained the Bayesian networks with 387 replays of Odin playing against the native AI, before using the networks to predict the technology tree of the native AI in actual games. The prediction accuracy was higher in this test, as can be seen in Figure 11 and Figure 12. From the graphs it can be concluded that prediction accuracy declines during the games first 15 000 or so frames and rise slightly after that. It is also evident that a network that is trained against the native AI performs better than one that is trained against replays of professional players, when they are both validated against the native AI. From Figure 11 we can also conclude that both Terran and Zerg yield prediction accuracies over 80 %, with the latter one staying at over 90 % for most of the time, whilst Protoss prediction accuracies even drop down to under 70 %. Overall, the prediction accuracy of the network that is trained against the native AI is above 80 % and performs almost as good when it is predicting the current time period compared to a time period as far as 6000 frames ahead.

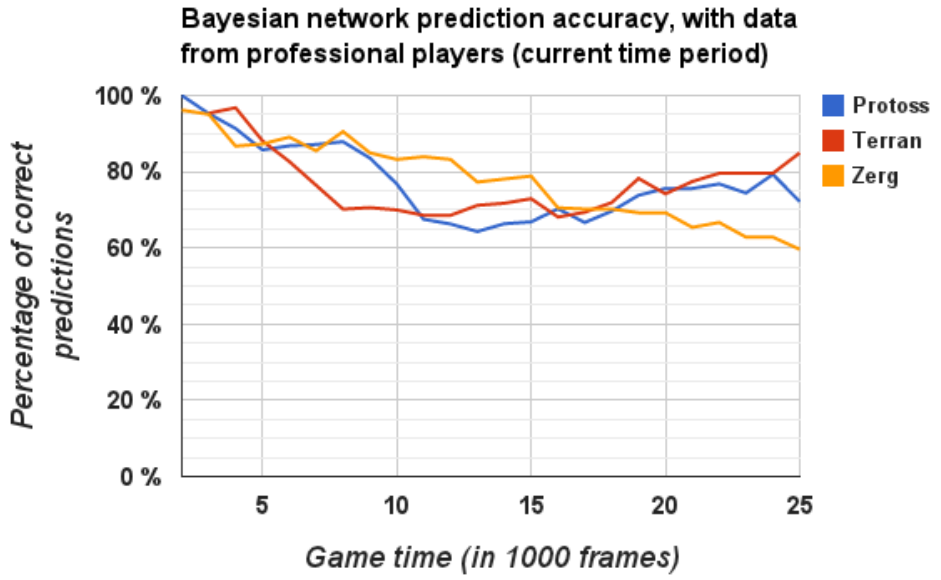

Figure 9. Percentage of correct predictions for the three different races in StarCraft. The data in the Bayesian networks is gathered from replays of professional StarCraft players, but the predictions were tried out on the native StarCraft AI.

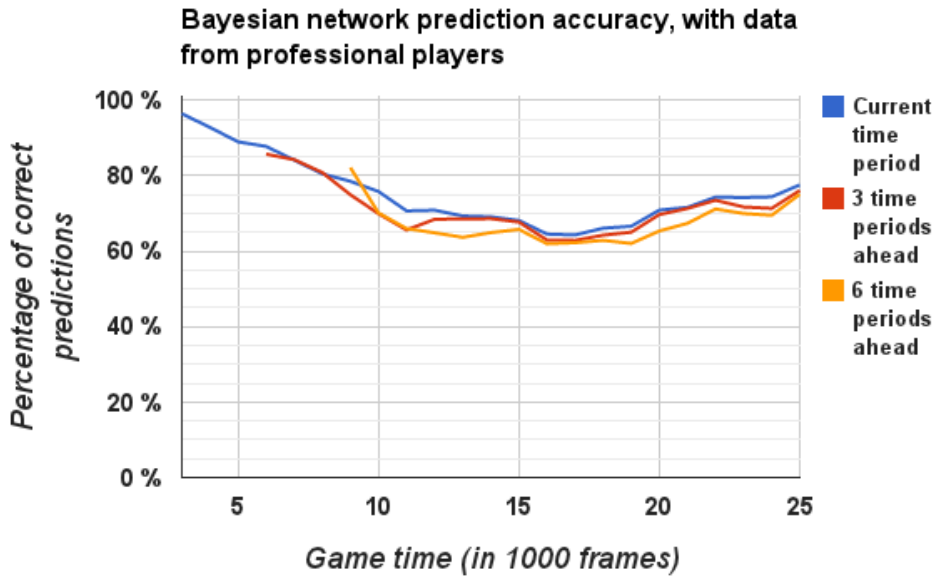

Figure 10. Percentage of correct predictions for the current time period, 3 time periods into the future, and 6 time periods into the future. The data in the Bayesian networks is gathered from replays of professional StarCraft players, but the predictions were tried out on the native StarCraft AI.

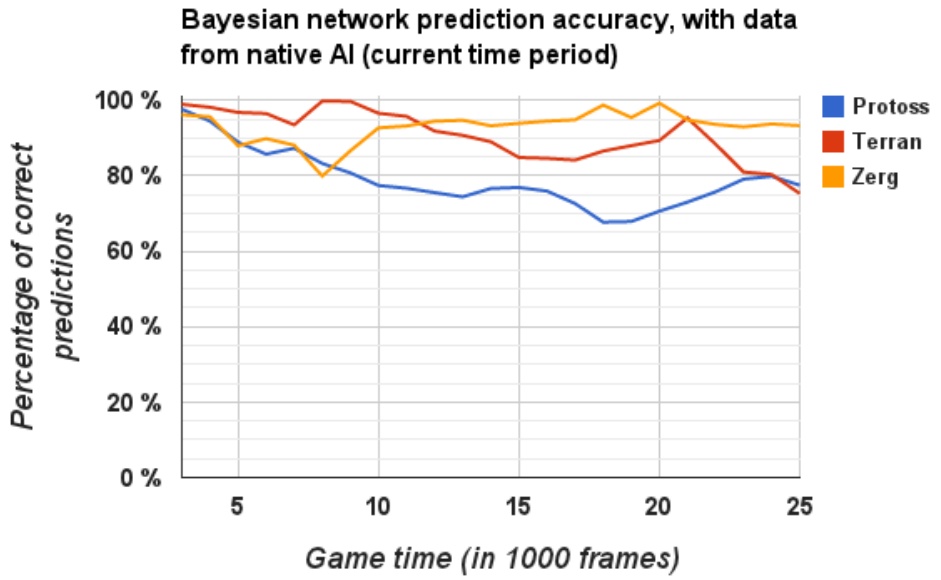

Figure 11. Percentage of correct predictions for the three different races in StarCraft. The data in the Bayesian networks is gathered from replays of the native StarCraft AI, and the predictions were tried out on the same AI.

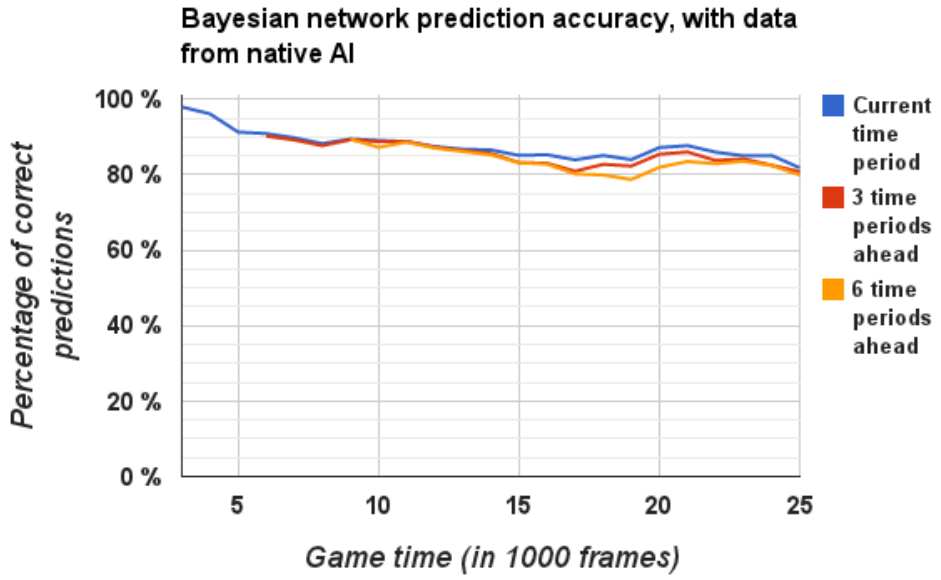

Figure 12. Percentage of correct predictions for the current time period, 3 time periods into the future, and 6 time periods into the future. The data in the Bayesian networks is gathered from replays of the native StarCraft AI, and the predictions were tried out on the same AI.

5.2 Opening strategies

The testing of opening adaptation was done by letting Odin play against the different races and noting the results. For all three races, the native AI was used as an opponent. In addition, another set of tests was done against BOB. However, due to BOB's inability to play as Zerg, the tests against BOB only included Protoss and Terran. Descriptions of the openings that were used can be found in section 4.1.

The tests were divided into two parts for each opponent, where games were played and the results were noted for each part. The reason for this setup was to first let Odin try out the different openings in the first part, and then observe which openings Odin continues to choose in the second part. This way, it is possible to determine if Odin manages to maximise his win rate by using successful openings more than unsuccessful ones. 15 games were played in each part, giving a total of 30 games against each opponent.

Table 1. The number of wins and losses after different openings while playing against the native Zerg AI in StarCraft.

	Part 1		**Part 2**	
Openings vs Zerg	**Wins**	**Losses**	**Wins**	**Losses**
Zealot Rush	3	0	2	0
Gateway Fast Expand	3	0	2	0
Nexus First	3	0	2	0
Forge Fast Expand	2	0	3	0
Two Gate Observer	2	0	3	0
Arbiter	2	0	3	0

As can be seen in Table 1, Odin won all the games against the native Zerg AI. Since the performance of an opening is measured in the number of won and lost games, Odin used all openings equally much.

When Odin played against the native Terran AI (see Table 2), the results looked quite different compared to the results against Zerg. During the first part of the test, only two openings managed to win all the games in which they were used; Zealot Rush and Gate Core Expand. The Zealot Rush opening seems to have lost a game quite early in the second part of the test, and Gate Core Expand was the only opening to not lose any game at all. Therefore, Gate Core Expand was used the most. Some of the other openings were also tried during the second part, since the algorithm for choosing opening takes into account how many times an opening has been used.

In the test against the native Protoss AI, as shown in Table 3, the two openings Arbiter and Gate Core Expand did not work very well, and so resulted in one loss each during the first part of the test. As the other openings were more successful, these two were never used during the second part of the test.

Table 2. The number of wins and losses after different openings while playing against the native Terran AI in StarCraft

	Part 1		Part 2	
Openings vs Terran	**Wins**	**Losses**	**Wins**	**Losses**
Gateway Fast Expand	0	1	0	0
Nexus First	1	1	1	1
Zealot Rush	4	0	1	1
Gate Core Expand	3	0	10	0
Arbiter	2	1	0	1
Two Gate Observer	0	1	0	0
Two Base Carrier	0	1	0	0

Table 3. The number of wins and losses after different openings while playing against the native Protoss AI in StarCraft. The number in parentheses denotes whether the results come from the first or second part of the test.

	Part 1		Part 2	
Openings vs Protoss	**Wins**	**Losses**	**Wins**	**Losses**
Zealot Rush	2	1	3	0
Two Gate DT	2	1	3	0
Two Gate Observer	2	1	2	0
Arbiter	0	1	0	0
Gateway Fast Expand	2	0	3	1
Nexus First	1	1	3	0
Gate Core Expand	0	1	0	0

Odin had an easier time against BOB playing as Protoss (see Table 4). In the first part of the test, only two openings, Arbiter and Nexus First, resulted in losses. These two were never used in the second part of the test. Only one opening resulted in a loss in the second part. The build order that BOB used for these tests was Dragoon With Range, which is a build order that focuses on producing many Dragoons (see Appendix D for the detailed build order).

However, the tests against BOB playing as Terran provided very different results, as can be seen in Table 5. In the first part of the tests, only two openings, Nexus First and Two Gate Observer, managed to result in victory. Later in the second part, the first use of Nexus First resulted in a loss, causing Two Gate Observer to be played for the majority of the remaining games. During these tests, BOB was using the Marine Medic Push build order, which is optimised to quickly produce large amounts of Marines and Medics (see Appendix D for the detailed build order).

As can be seen in Table 6 the results from the random native AI gave results very similar to the

Table 4. The number of wins and losses after different openings while playing against BOB (as Protoss). The number in parentheses denotes whether the results come from the first or second part of the test.

	Part 1		Part 2	
Openings vs BOB (Protoss)	**Wins**	**Losses**	**Wins**	**Losses**
Zealot Rush	3	0	3	0
Two Gate DT	3	0	2	1
Two Gate Observer	2	0	3	0
Arbiter	0	1	0	0
Gateway Fast Expand	2	0	3	0
Nexus First	1	1	0	0
Gate Core Expand	2	0	3	0

Table 5. The number of wins and losses after different openings while playing against BOB (as Terran). The number in parentheses denotes whether the results come from the first or second part of the test.

	Part 1		Part 2	
Openings vs BOB (Terran)	**Wins**	**Losses**	**Wins**	**Losses**
Gateway Fast Expand	0	2	0	0
Nexus First	1	2	0	1
Zealot Rush	0	2	0	0
Gate Core Expand	0	2	0	0
Arbiter	0	2	0	0
Two Gate Observer	2	1	8	5
Two Base Carrier	0	1	0	1

Table 6. The number of wins and losses after different openings while playing against the native AI using random races. The number in parentheses denotes whether the results come from the first or second part of the test.

	Part 1		Part 2	
Openings vs Random	**Wins**	**Losses**	**Wins**	**Losses**
Zealot Rush	3	0	4	0
Gateway Fast Expand	2	1	0	0
Nexus First	3	0	4	0
Gate Core Expand	3	0	4	0
Forge Fast Expand	3	0	3	0

three other native AI tests. The only opening that lost was Gateway Fast Expand. Even though it had a good win rate, it was never used in the second part of the test as the rest never resulted in losses.

5.3 Army composition generator

The purpose of testing the army composition generator is to understand its effect on Odin's overall performance, and especially how the predictions might change the outcome of games. The ACG has two constants which are tweaked in order to adapt its behaviour. The two constant are called unitThreshold and unitStartValue which can be read about in Section 4.3. These constants are varied in order to understand the importance of predictions when using the ACG. We also let Odin use Bayesian networks that predicted what the opponent would have two time periods (1 minute and 36 seconds) into the future. This was done in order to observe if Odin would win more matches by adapting to the opponent's future unit composition.

Table 7. Results from using different sets of parameters in the army composition generator (ACG). The parameters are listed in the following order: (predictionThreshold, unitStartValue).

ACG	**Parameters**	**Wins**	**Losses**	**Win rate**
Default values	(0.6, 0.1)	29	1	96.7 %
No predictions	(0.99, 0.01)	24	6	80.0 %
Overuse predictions	(0.3, 0.3)	26	4	86.7 %
Network 2 time periods ahead	(0.6, 0.1)	28	2	93.3 %
Total		107	13	89.2 %

The test was conducted by letting Odin play against the random native AI, the results can be seen in Table 7. Overall, Odin had an 89.2 % win rate against StarCraft's native AI. With the default values Odin accomplished to win 29 out of 30 games and without predictions Odin was able to win 24 out of 30 games.

5.4 The attack-defend state uncertainty factor

As explained in section 4.3, Odin compares the two players' army potentials in order to determine whether the attack or defend state should be active. This comparison also involves an uncertainty factor and by varying the value of this factor, it is possible to optimise the win/loss ratio of the AI agent.

This test was performed by letting Odin play against the random native AI. For each of the tested values of the uncertainty factor, 30 games were played. Before each value was tested the knowledge about which opening resulted in most victories was reset, since the choice of opening is optimised over time. The results can be seen in Table 8, and the plotted data can be seen in Figure 13.

Table 8. The win rates obtained from tests where the uncertainty factor was varied. A lower value of the uncertainty factor results in the Odin playing more aggressively, while a greater value results in the agent playing more defensively. Each value was tested by letting Odin play 30 games against the random native AI.

Uncertainty factor	Wins	Losses	Win rate
1.0	28	2	93.3 %
1.25	28	2	93.3 %
1.5	27	3	90.0 %
1.75	25	5	83.3 %
2.0	22	8	73.3 %

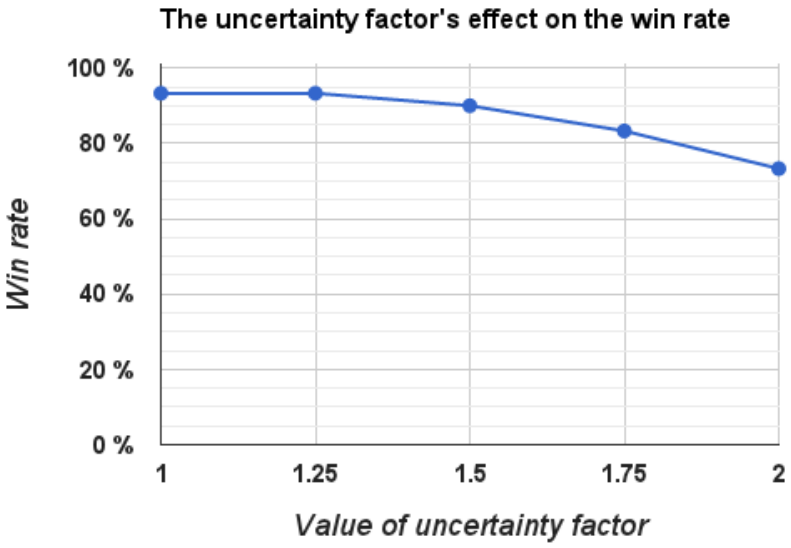

Figure 13. The win rate data from Table 8 plotted against the uncertainty factor.

6 Results and discussion

In this section we evaluate and discuss the results, as well as the possible error sources for the test data. We also present some of the problems that we encountered during the implementation. Lastly, we discuss the project's place in the field of research, and present possible future improvements of the project.

6.1 Predictions

The initial plan was to use BWAPI to examine replays of professional StarCraft players' games. Predictions would be made with the fog of war enabled. Then disabling the fog of war would allow us to evaluate the results of the predictions. However, this was not possible as BWAPI is not able handle the individual players' fog of war during replays, and it was impossible to determine which player triggered each event. Thus it was decided to test our predictions against the native AI.

As is shown in Figures 9, 10, 11, and 12; once the Bayesian networks were trained with games from the native AI, their predictions against it became more accurate. This is hardly surprising as the networks were both trained and validated against the native AI. If they instead had been validated against professional players, the results would probably not have been as accurate. We can observe that the native AI only uses a very limited set of strategies, which also might explain the high prediction accuracies. If we had trained and validated the Bayesian networks against professional players, the prediction accuracy would probably have been slightly worse. This is mainly because professional players tend to use a larger variety of strategies. If there is a larger number of strategies, but they are similar to each other, it becomes harder to accurately predict which strategy is used.

It is also possible to observe that the accuracy of the predictions barely went down at all when we tried to predict the game state of some time periods ahead. While this is more surprising, it can be explained with the fact that most players' usage of the technology tree follows almost the same order and only deviates slightly. This further strengthens the assumption that the technology tree can be used to accurately predict the enemy's overall strategy.

Overall we could see a decline in the accuracy of our predictions over time, which increased again in the later game stages. This seems only natural, as predictions rely heavily on the scouting efforts. As the game progresses, it becomes harder to gather information about the enemy since the enemy army often guards its base. Without information it is harder to predict the state of the opponent's technology tree, and the accuracy drops. The further the game progresses, the more likely it is for battles to occur, in which more information about the opponent might be gained. The further the game goes, the more likely it is for the technology tree of the opponent to expand to its fullest. So despite using different openings, the tech tree will most likely start to resemble a generic one, the longer the game progresses.

6.2 Opening strategies

There are several factors that affect whether Odin wins a game with a certain opening or not. For example, Odin controls some unit types better than others, so the types of units that are used by the opening is a factor. There is also some randomness involved, since Odin does not know what opening strategy the enemy will use. The most determinant factor is probably how

well the opening prepares for the rest of the game. This could be done, either by making a big army to get the advantage of an early attack, or by building up a strong economy. It is also important to note that the opening only specifies what Odin should do in the beginning of the game, and the result may be determined by an event in a later stage of the game, unrelated to the choice of opening.

The opening Gate Core Expand outperformed all the other openings against the native Terran AI. It was the only opening that won all its games, and was therefore used much more than any other opening. The reason that this opening performed best could be due to it being the shortest opening. This gives Odin an early opportunity to use the army composition generator to train units that can counter the enemy units, instead of being forced to follow a static opening build order. The opening also builds up a good economy, which gives Odin the resources to tech to the units it needs as fast as possible.

The results against BOB as Protoss and native AI as Zerg, Protoss and Random were quite similar. There was no opening that was significantly better than any other since Odin won almost all matches. This lead to a little or no change in which openings were used.

When testing Odin against BOB as Terran, the result was not very good since Odin had a win rate of only 37 %. The main reason for this was that BOB, as explained in section 5.2, uses the Marine Medic Push build order. Medics are support units, which rather than directly attacking enemies, heals friendly units. As Odin micromanages each unit to target what is most dangerous, the Medics are seldom targeted first, and they are thus able to heal the nearby Marines as soon as they are damaged. This makes it hard for Odin to defeat an enemy army which includes Medics.

The Two Gate Observer opening resulted in victory in 10 out of the 16 times it was used. This was, however, not because the Observer unit is a good counter for the Marine-Medic army. It was rather because BOB never built any unit capable of detecting invisible units such as Observers. In most of the games where this opening was used, BOB did manage to destroy Odin's starting base. Despite that, in most of the cases Odin had some buildings located far from the main base, keeping Odin from losing the game. Additionally, BOB's army chased the invisible Observer without being able to attack it due to the lack of detection. All this resulted with Odin winning the games by simple having a higher in-game score when the time limit defined by the tournament rules was reached.

6.3 Army composition generator

The most notable results in Figure 7 are when the predictions are either overused, or not used at all. These two cases both resulted in lower win rates. The other two test cases, 'Default values' and 'Network 2 time periods ahead', generated slightly different results. This difference may, however, have been caused by pure chance.

Evaluating the army composition by only looking at the win rate might not be the best way to evaluate. Even so, it is possible to observe that the ACG affects the outcome of games. This is especially evident in the test cases where no predictions are used at all, since Odin then depends on directly observing enemy units in order to know what the enemy has. When an opening is finished, Odin is usually able to predict what units the opponent has even though no direct observations have been made. However, when the predictions are not used, it is not as easy for

Odin to adapt, as indicated by the results.

Even though using predictions is a significant part of achieving a high win rate, it is also important to not overuse them. This can be seen in the test case 'Overuse predictions', where the results are worse than for the default ACG parameters. The reason for these poor results is likely that Odin tries to build counters for everything at once, and so does not manage to concentrate on what the enemy actually has. Odin often tries to counter units that do not exist, probably because of the low threshold value.

Both the tests where the predictions were changed gave significantly worse results than the default ACG parameters. When using a Bayesian network that predicted two time periods into the future, the results were not significantly worse. Instead, the difference was so small that it is not possible to safely say whether 'Default values' or 'Network 2 time periods ahead' is better or not.

All in all, the ACG is important and changing the parameters has a significant effect on the test results. Predictions are an important part of the ACG, though they should not be overused.

6.4 The attack-defend state uncertainty factor

In order to understand the following discussion, one must first recall from section 4.3 that a low value of the uncertainty factor causes Odin to enter the attack state more frequently, while a higher value causes the agent to spend more time in the defend state. It should also be stated that an uncertainty factor below 1 was not used since it would indicate that the opponent has less units than observed.

As can be seen in Table 8 and Figure 13, Odin won fewer matches as the uncertainty factor increased in value. This was likely because the attack state more or less is the aggressive behaviuor inherited from UAlbertaBot, while the defend state was implemented in the later parts of our project. This has caused the defend state to not be optimised to the same extent as the attack state, a fact that might have contributed to the declining win rate as the uncertainty factor increased. Another possible explanation is that a StarCraft player rarely wins a match only by defending. In order to win, a player must attack the opponent, and when Odin attacks the opponent more seldom it affects the win rate.

6.5 Possible sources of errors

It is important to note that no test can be perfect, and there are some possible sources of errors that might have affected the test results. These sources of errors are either so small that they were not taken into consideration, or they were simply not thought of when the tests were constructed. The possible sources of errors are presented and explained below. Of course, there is a possibility that there are other sources of errors as well that we have not thought of.

The different tests were performed on different computers, and each had a different architecture and clock rate. As the tournament rules limit the amount of time each frame can take, a higher clock rate allows for more calculations per time frame and may gain an advantage over computers with lower clock rates. However, Odin seldom reaches this limit, and the clock rates were therefore not taken into consideration.

When the data mining was done on StarCraft's native AI, the games were played against randomly chosen races. One might imagine that this would result in the three races being chosen approximately the same number of times, but this was not the case. Instead, the result was very close to 50 % Protoss, 30 % Terran, and 20 % Zerg games. This led us to believe that StarCraft did not choose randomly between the three races, but instead between ten different strategies, where Protoss used five strategies, Terran used three and Zerg used two. This would mean that the results against the random native AI are biased towards Protoss. Unfortunately, we have not gathered any hard data to support this hypothesis.

Another possible source of error could be that relatively few games, about 30 per test, were played during the testing. If more games had been played, the results would likely be more accurate. Letting the agent play games during the tests is however a very time-consuming task, and there was simply not enough time to play a larger number of games.

6.6 Problems during implementation

While developing the usage of predictions, we discussed what we wanted Odin to predict. At one point, the goal was that the agent should be able to predict the number of units the opponent has, as well as the probability of a unit or building being available. However, using the same Bayesian networks for predicting all this would have caused the networks to become too complex to be efficiently used. Because of this, we decided that the networks should not be able to determine the number of units the opponent has.

During the project we also had some problems regarding the BuildingPlacer included in UAlbertaBot. It often had problems finding valid positions for new buildings, the search algorithm needed so much time that it slowed the game down. The results of previous searches are not remembered, and this means that an invalid position is examined every time the algorithm is run. These problems often causes the game to slow down in the later parts of long matches.

Sometimes Odin's army refuses to search for more enemies after the enemy main base is destroyed. This is caused by a bug in the InformationManager, which fails to update the information about a Terran building type called *add-ons*. An add-on is an addition to a conventional Terran building. In the StarCraft engine, an add-on belongs to a player if it is attached to a building belonging to the player. However, if the add-on is not attached to any building, it becomes neutral and no longer belongs to any player. The InformationManager seems to only update the knowledge of enemy buildings if they belong to the enemy. This means that the InformationManager believes that a hostile add-on is still alive, if the add-on is neutral when it is destroyed. Sometimes, Odin therefore gets stuck in the enemy base, trying to destroy the already destroyed building. This behaviour often delays the victory by quite a lot of time.

The module in UAlbertaBot that takes the most time to run is the BuildOrderSearch. When more unit types than UAlbertaBot's default types were added and used, the module did not work very well any more. Firstly, the program started to crash when new unit types were added, because they were not defined in a list used by the BuildOrderSearch. Secondly, the module did not scale very well. It needed too much time in order to find a build order for the goal provided by the ACG. Sometimes, it did not even manage to reach any goal at all, with the result that the agent spent a lot of time doing nothing. Therefore, the way in which the BuildOrderSearch was used had to be changed. In section 4.1, some of the changes are described. The changes improved the performance of Odin greatly. Another problem that also occurred from time to

time was that the BuildOrderSearch simply did not run at all. We never understood why this happened, and so have not been able to come up with a solution for this. If we could redo the project with the knowledge we now have, we would have replaced the BuildOrderSearch module.

We would also plan the project in another, more detailed, way. Unfortunately, we realised too late that our plan to use case-based reasoning would be hard to implement using the program structure from UAlbertaBot, with rule-based behaviour and an update-loop design. If we had done more detailed research before starting the implementation, it might have been possible to realise the difficulties of making the agent perform a sequence of actions, as we had first planned. Of course, it is easy to be afterwise, but this also highlights the importance of doing research and planning before starting to implement the agent. That said, it would have been hard to anticipate the problems and bugs we came across during the implementation.

6.7 Contributions

This project touches on a subject where not much research has been done. As stated in 'A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft' [12], high-level decision-making is an open field. Decisions are based on information, and just as humans understand information, it is important that computers also understand the information before making decisions. This project aims to give the agent as good information as possible as a base for its decisions, and an approach to decision-making is shown.

6.8 Future work

There are numerous ways in which it is possible to improve Odin in the future. One example is to make more advanced predictions, not only of the enemy tech tree, but also of his army size and attack timings. One can assume that this would in turn incite for more sophisticated scouting, since predictions rely heavily on what is observed. Also, the correlation between scouting and prediction accuracy was not examined in this project, but would probably be very interesting to investigate.

The ACG could be improved to learn which units are cost-effective against certain units, instead of relying on the predefined counters used by Odin. This would lead to a more dynamic behaviour where Odin would adapt even more over time. There are several machine-learning techniques that could be applied to do this, one of them being case-based reasoning.

Some future work might also include fixing the problems we found in the behaviours Odin inherited from UAlbertaBot. One obvious example is the micromanagement of units. The only units whose micromanagement is currently optimised are Zealots and Dragoons, since these are the two units primarily used by UAlbertaBot. The management of units such as High Templars, Reavers, Carriers, Observers, and many more can likely be significantly improved.

The combat simulator used by Odin in order to predict the outcomes of small battles can also be refined. The original version used in UAlbertaBot does not take spell-casters into account, resulting in inaccurate predictions of battles that involve units such as Medics, High Templars, or Reavers. The version used by Odin is only slightly changed, in that it treats High Templars, Reavers and Carriers in the same way as Dragoons. Modifying the combat simulator to correctly represent spell-casters should lead to better combat predictions, and therefore also improved behaviour by the AI agent.

For some, it might be hard to see exactly how other areas might benefit from game AI research, but the explored techniques can be applied to many everyday real-world situations. They can be used in a variety of ways to highlight correlation and causes in large data sets, and can thus be applied to most fields. One area where it could work very well is risk analysis. Insurance companies must weigh the risk against the profit when insuring people, in order to negotiate a reasonable fee. Additionally, companies in the finance sector, that handle considerable amounts of money, need to be aware of all possible risks to minimise losses. One might also argue that Bayesian networks are usable while examining the large data sets that are generated on the Internet. In particular in social media, there is a growing need to analyse user data.

7 Conclusion

The results suggest that Odin, after observing the opponent during gameplay, is able to accurately predict the state of the enemy's tech tree. It is also shown that these predictions can be used for the decision-making process by using the army composition generator, and that they have a positive effect on the outcome of games. Odin is better at adapting its unit composition when predictions are done, which is shown by the higher win rate against the native AI.

Predictions give the AI agent the intuitive feeling that most human players have about their opponents. Without actually observing what technology the opponent has acquired, Odin is able to estimate how far the enemy has progressed in the tech tree and what units or buildings the opponent might be expected to have. Achieving this in an environment as complex as StarCraft indicates that it is possible to perform similar tasks in other, complex, environments as well. It would also be applicable to real-world problems where an AI agent has to make decisions based on incomplete information.

References

[1] IBM, "Deep Blue," http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/, 2012, [Online; accessed 2014-04-26].

[2] D. Churchill and M. Buro, "Incorporating Search Algorithms into RTS Game Agents," 2012.

[3] M. Buro and T. M. Furtak, "RTS Games and Real–Time AI Research," 2004.

[4] P. Spronck, M. Ponsen, and E. Postma, "Adaptive game ai with dynamic scripting," in *Machine Learning*. Kluwer, 2006, pp. 217–248.

[5] ai-junkie, "State-Driven Game Agent Design," http://www.ai-junkie.com/architecture/state_driven/tut_state1.html, 2004, [Online; accessed 2014-05-08].

[6] A. J. Champandard, "10 Reasons the Age of Finite State Machines is Over," http://aigamedev.com/open/article/fsm-age-is-over/, 2007, [Online; accessed 2014-05-10].

[7] J. Eriksson and D. Ø. Tornes, "Learning to play Starcraft with Case-based Reasoning - Investigating issues in large-scale case-based planning," Master thesis, Norwegian University of Science and Technology, June 2012.

[8] B. G. Weber and S. Ontañón, "Using automated replay annotation for case-based planning in games," http://users.soe.ucsc.edu/~bweber/pubs/iccbr-cg.pdf, 2010, [Online; accessed 2014-05-09].

[9] I. Martinez, "Villagers Finite State Machine ," http://martidore.blogspot.se/2013/02/villagers-finite-state-machine.html, 2013, [Online; accessed 2014-06-03].

[10] "StarCraft - Does the AI cheat?" http://gaming.stackexchange.com/questions/5544/does-the-ai-cheat, 2010, [Online; accessed 2014-05-09].

[11] B. Schwab, *AI game engine programming*. Cengage Learning, 2009.

[12] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," 2013.

[13] M. Buro, "Real-time strategy games: A new ai research challenge," 2003.

[14] Edge-online, "Blizzard Confirms One "Frontline Release" for '09," http://www.edge-online.com/news/blizzard-confirms-one-frontline-release-09/#null, 2008, [Online; accessed 2014-05-08].

[15] T. Chick, "Starcraft," http://www.ign.com/articles/2000/06/03/starcraft-2, 2000, [Online; accessed 2014-04-28].

[16] e-Sports Earnings, "StarCraft: Brood War," http://www.esportsearnings.com/games/152-starcraft-brood-war/list_events, 2014, [Online; accessed 2014-05-08].

[17] Blue Screen of Awesome, "The Hunt — Best RTS: StarCraft," http://bluescreenofawesome.com/2012/05/18/the-hunt-best-rts-starcraft/, 2012, [Online; accessed 2014-05-12].

[18] B. Das, "Representing uncertainties using bayesian networks," DTIC Document, Tech. Rep., 1999.

[19] V. Pavlovic, J. M. Rehg, T.-J. Cham, and K. P. Murphy, "A dynamic bayesian network approach to figure tracking using learned dynamic models," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 1. IEEE, 1999, pp. 94–101.

[20] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," *AI communications*, vol. 7, no. 1, pp. 39–59, 1994.

[21] G. Synnaeve and P. Bessière, "A Bayesian model for opening prediction in RTS games with application to StarCraft," 2011.

[22] R. E. Oen, "ASPIRE Adaptive strategy prediction in a RTS environment," Master thesis, The University of Bergen, June 2012.

[23] M. Certicky and M. Certicky, "Case-Based Reasoning for Army Compositions in Real-Time Strategy Games," http://agents.fel.cvut.cz/~certicky//files/publications/scyr2013.pdf, 2013, [Online; accessed 2014-05-08].

[24] K. van Mens, "Strategic reasoning in complex domains - A comparative survey on scientific AI techniques to improve real-time strategy game AI," Bachelor thesis, Universiteit Utrecht, May 2012.

[25] A. Smith, "Skynet," https://code.google.com/p/skynetbot/, 2014, [Online; accessed 2014-05-09].

[26] D. Churchill, "2013 AIIDE StarCraft AI Competition Report," http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/report2013.shtml, 2013, [Online; accessed 2014-05-08].

[27] F. Richoux, "aiurproject," https://code.google.com/p/aiurproject/, 2014, [Online; accessed 2014-05-09].

[28] D. Klein, "The Berkeley Overmind Project," http://overmind.cs.berkeley.edu/, 2014, [Online; accessed 2014-05-09].

[29] D. Churchill, "UAlbertaBot," https://code.google.com/p/ualbertabot/, 2014, [Online; accessed 2014-05-09].

[30] M. Buro, "CMPUT 350 Advanced Game Programming," https://skatgame.net/mburo/courses/350/, 2013, [Online; accessed 2014-05-08].

[31] D. Churchill, "Artificial Intelligence - UAlbertaBot," https://code.google.com/p/ualbertabot/wiki/ArtificialIntelligence, 2014, [Online; accessed 2014-05-09].

[32] D. Churchill and M. Buro, "Build Order Optimization in StarCraft," 2011.

[33] ——, "Portfolio greedy search and simulation for large-scale combat in starcraft," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[34] teamliquid.net, "Team Liquid," http://www.teamliquid.net/, 2014, [Online; accessed 2014-05-12].

[35] dlib.net, "Dlib C++ Library," http://dlib.net/, 2014, [Online; accessed 2014-05-09].

[36] 2014 StarCraft AI Competition, "Rules," http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/rules.shtml, 2014, [Online; accessed 2014-05-18].

[37] langpop.com, "Programming Language Popularity," http://langpop.com/, 2013, [Online; accessed 2014-05-09].

[38] K. Schwaber and J. Sutherland, *The Scrum Guide*, Scrum.org, 2013.

Appendices

A Tools

During the project, several tools and libraries were used in order to get a structured way of working with the project as well as to be able to create an AI agent. This chapter gives an insight into those tools and why they were chosen.

BWAPI

BWAPI is an acronym for the Brood War API. It was used since it is, by far, the easiest way to implement an AI agent for StarCraft: Brood War. BWAPI is written in C++ and interacts with StarCraft by injecting code into the game process. When compiling an AI agent with BWAPI, the end result is a dll file, and the game is started via a program called ChaosLauncher that injects the created dll into the StarCraft process.

C++

C++ was used as a programming language for the project because it is recommended to use with BWAPI. C++ is widely used around the world and has influenced other common languages such as Java and C# [37]. C++ is very common in game development because of its both low level and high level features meaning it is both a powerful, simple and clear language.

Visual Studio

BWAPI is built using the integrated development environment Visual Studio. Since Visual Studio is the recommended programming environment for developing AI agents using BWAPI, this project was also developed and compiled within Visual Studio.

GeNIe

In order to be able to work with a graphical representation of the Bayesian network, a development environment called GeNIe (an acronym for Graphical Network Interface) was used. However, for the agent to be able to use the network during runtime, the network created from GeNIe was parsed and then loaded into the general purpose library Dlib.

Dlib

Dlib is a cross platform open source software library written in C++. It is a collection of independent software components, which can deal with a lot of different tasks. Features include networking, threading, different data structures, Bayesian networks and more. Dlib was used for the Bayesian network functionality.

Scrum

The project was managed using a modified version of the agile software development framework called Scrum. In Scrum, the team is known as the Scrum Team and consists of a Product Owner, the Development Team and a Scrum Master [38]. The Product Owner is responsible for managing the Product Backlog, a list of tasks that need to be done. The Development

Team does the work to get the product closer to release at the end of each sprint. The Scrum Master is responsible for ensuring that the Scrum process is understood and enacted by the team.

The Scrum process is mainly composed of Sprints. A Sprint consists of Sprint Planning, Daily Scrums, the development work, the Sprint Review and the Sprint Retrospective. The Sprint Planning is where the work to be performed during the Sprint is planned. The plan is created by the whole Scrum Team. Daily Scrums are short daily meetings which are limited to a maximum of 15 minutes. They are used to synchronise the team and create a plan for the next 24 hours. A Sprint Review is held at the end of a Sprint in order to inspect the work that has been done. The Product Backlog might also be updated if needed. During the Sprint Review, the team members discuss what should be done next. After the Sprint Review, but before the next Sprint Planning, an event called Sprint Retrospective is held. The purpose of this event is to give the team an opportunity to review itself and create a plan for how to improve during the next Sprint.

Weekly Scrum meetings were held in order to decide what should be done during the following week. However, unlike the standard Scrum version, no Daily Scrum meetings were held since the group members would mostly work from home by themselves, likely at different times of the day. As an aid for the Scrum documentation and administrative tasks, the online service Pivotal Tracker was used.

Pivotal Tracker

Pivotal Tracker is a software for agile project management and collaboration. It is a web-based service where stories can be created that defines what needs to be done. Everyone in the team can see these stories and anyone can mark a story as 'started' to show the rest of the team that the story is being worked on. Pivotal Tracker was used to keep track on who was working on what and to be able to see what was needed to be done.

Git

The version control system Git was used for the project. Git is a widely used source code management system initially designed by Linus Torvalds for the development of the Linux kernel. The source code of the project is hosted on Github, a free web-based code hosting service for open-source projects. The source code of our project can be found at `https://github.com/Plankton555/Odin`.

B StarCraft maps used for testing

The following table contains the StarCraft maps that were used for testing. Map size denotes the maximum number of players that can play on the map.

Map name	**Map size**
Benzene	2
Neo Moon Glaive	3
Tau Cross	3
Circuit Breaker	4
Empire of the Sun	4
Fighting Spirit	4
Icarus	4
Jade	4
La Mancha 1.1	4
Python	4
Roadrunner	4
Total nr of maps	11

C Unit counters

The following three tables contain the units used by Odin in order to counter the opponent's units.

Protoss units	**Cheap counter**	**Expensive counter**
Zealot	Dragoon	High Templar
Dragoon	Dragoon	High Templar
Dark Templar	Photon Cannon	Observer
High Templar	Reaver	Dark Templar
Archon	Reaver	Carrier
Dark Archon	Dragoon	Reaver
Reaver	Dark Templar	Carrier
Scout	Dragoon	Corsair
Corsair	Dragoon	-
Arbiter	Observer	High Templar
Carrier	Dragoon	Corsair
Observer	Photon Cannon	Observer
Shuttle	Photon Cannon	Dragoon

Terran units	**Cheap counter**	**Expensive counter**
Marine	Zealot	Dragoon
Firebat	Dragoon	High Templar
Medic	Zealot	Reaver
Ghost	Photon Cannon	Observer
Vulture	Dragoon	Carrier
Spider Mine	Zealot	Observer
Siege Tank Tank Mode	Zealot	Dark Templar
Siege Tank Siege Mode	Zealot	Carrier
Goliath	Zealot	Reaver
Wraith	Dragoon	Observer
Dropship	Photon Cannon	Corsair
Science Vessel	Dragoon	Scout
Battlecruiser	Dragoon	Scout
Valkyrie	Dragoon	Scout

Zerg units	**Cheap counter**	**Expensive counter**
Zergling	Zealot	Reaver
Hydralisk	Zealot	Dragoon
Lurker	Observer	High Templar
Ultralisk	Dragoon	Carrier
Defiler	Photon Cannon	Reaver
Overlord	Dragoon	-
Mutalisk	Corsair	High Templar
Scourge	Dragoon	Scout
Queen	Dragoon	Reaver
Guardian	Corsair	High Templar
Devourer	Dragoon	High Templar

D Opening build orders

The opening build orders are stored as sequences of numbers, where each number represent a specific unit, building, or upgrade. Below is a dictionary of which units the different numbers represent.

Id	Protoss Name	Terran Name
0	Probe	SCV
1	Pylon	Supply Depot
2	Nexus	Command Center
3	Gateway	Barracks
4	Zealot	Refinery
5	Cybernetics Core	Marine
6	Dragoon	Academy
7	Assimilator	Stim Packs
8	Singularity Charge	Medic
9	Forge	-
10	Photon Cannon	-
11	High Templar	-
12	Citadel of Adun	-
13	Templar Archives	-
14	Robotics Facility	-
15	Robotics Support Bay	-
16	Observatory	-
17	Stargate	-
18	Scout	-
19	Arbiter Tribunal	-
20	Arbiter	-
21	Shield Battery	-
22	Dark Templar	-
23	Shuttle	-
24	Reaver	-
25	Observer	-
26	Corsair	-
27	Fleet Beacon	-
28	Carrier	-
29	Leg Enhancements	-
30	Psionic Storm	-

The table below contains the different opening build orders that are used by Odin.

Opening name	**Build order**
Zealot Rush	0 0 0 0 1 0 3 3 0 0 4 1 4 4 0 4 4 0 1 4 3 0 1 0 4 0 4 4 4 4 1 0 4 4 4
Gateway Fast Expand	0 0 0 0 1 0 0 3 0 0 1 0 4 0 0 0 0 2 0 7 0 0 1 5
Nexus First	0 0 0 0 1 0 0 0 0 0 2 0 3 0 7 0 0 5 3 4 0 0 1 6 6 8 0 0 1
Arbiter	0 0 0 0 1 0 3 3 0 0 4 1 4 4 7 0 5 1 0 12 0 13 1 0 17 0 4 0 19 1 0 0 20 17 3 3 3 1 4 4 4 4 1 0 4 0 20 1 0 4 0 20 4 4 4 4 0 1 4 1 20 4 1 4 4 4 0 4 20 0 4
Forge Fast Expand	0 0 0 0 1 0 0 0 0 2 0 9 0 0 1 0 0 3 0 10 0 0 7 0 0 0 5
Two Gate Observer	0 0 0 0 1 0 0 3 0 0 7 0 0 5 0 0 1 0 0 8 6 0 0 1 6 0 14 6 3 6 1 16 25
Two Gate DT	0 0 0 0 1 0 0 3 0 0 7 0 4 0 0 1 0 0 5 0 4 0 0 1 0 6 0 12 0 6 3 1 13 0 4 4 0 1 22 22 0 9 0 1 2
Gate Core Expand	0 0 0 0 1 0 0 3 0 0 7 0 0 5 0 1 0 0 6 2
Two Base Carrier	0 0 0 0 1 0 0 0 0 0 2 0 3 0 7 0 0 5 4 0 7 0 3 1 6 6 8 0 0 1 0 0 14 0 0 17 0 0 27 0 0 17 28 28 3 3 3

The table below contains the opening build orders that are used by BOB.

Opening name	**Race**	**Build order**
Dragoon With Range	Protoss	0 0 0 0 1 0 0 3 0 0 7 0 0 5 0 1 0 0 0 8 6 0 0 1 6 0 6 3 3 3 6 1 6 6 6 6 1
Marine Medic Push	Terran	0 0 0 0 0 1 0 0 3 0 0 3 0 5 1 0 5 4 5 0 0 5 0 6 1 5 5 0 3 3 5 5 8 1 5 5 0 8 5 5 0 8 1 0 5 5 0 8 5 0 1 5 0 5 8 5 0 1 5 5 8 0 5 8 5 1 5 0 5 8 5 0 5 5 1 8 5 0 5 8 5 0 5 1 8 1 5 5 5 5 5 5 5

E Prediction results

The table below shows the prediction accuracy of the Bayesian network for the different races. The columns that are marked as ”Pro” used a network that was trained against professional players replays, while the other columns used a network that was trained against the native AI of StarCraft. Since the games were played against Random, the networks could not be loaded until the first encounter and thus some of the entries in the table are empty.

Time period	**Protoss Pro**	**Protoss**	**Terran Pro**	**Terran**	**Zerg Pro**	**Zerg**
1						
2	100,00 %				96,15 %	
3	95,24 %	97,62 %	95,34 %	98,92 %	95,05 %	96,15 %
4	91,33 %	94,47 %	96,77 %	98,14 %	86,71 %	95,60 %
5	85,71 %	88,87 %	88,10 %	96,77 %	87,28 %	87,93 %
6	86,79 %	85,71 %	82,66 %	96,47 %	89,05 %	89,83 %
7	87,14 %	87,27 %	76,41 %	93,47 %	85,50 %	88,09 %
8	87,86 %	83,23 %	70,16 %	99,77 %	90,53 %	79,90 %
9	83,57 %	80,75 %	70,56 %	99,70 %	84,94 %	86,60 %
10	76,79 %	77,41 %	69,96 %	96,55 %	83,22 %	92,68 %
11	67,50 %	76,67 %	68,55 %	95,72 %	83,92 %	93,18 %
12	66,27 %	75,51 %	68,55 %	91,82 %	83,22 %	94,43 %
13	64,29 %	74,40 %	71,17 %	90,70 %	77,27 %	94,69 %
14	66,33 %	76,60 %	71,71 %	89,00 %	78,08 %	93,24 %
15	66,84 %	76,88 %	72,90 %	84,83 %	78,85 %	93,87 %
16	70,24 %	75,92 %	68,10 %	84,60 %	70,51 %	94,46 %
17	66,67 %	72,62 %	69,35 %	84,14 %	70,19 %	94,81 %
18	69,64 %	67,69 %	71,89 %	86,53 %	70,19 %	98,72 %
19	73,81 %	67,86 %	78,23 %	87,96 %	69,23 %	95,43 %
20	75,60 %	70,59 %	74,19 %	89,33 %	69,23 %	99,23 %
21	75,60 %	72,99 %	77,42 %	95,31 %	65,38 %	94,78 %
22	76,79 %	75,71 %	79,57 %	88,39 %	66,67 %	93,59 %
23	74,40 %	79,08 %	79,57 %	80,97 %	62,82 %	92,95 %
24	79,29 %	79,85 %	79,57 %	80,32 %	62,82 %	93,71 %
25	72,14 %	77,47 %	84,95 %	75,27 %	59,62 %	93,27 %

The table below shows the prediction accuracy of a network trained with professional players replays and validated against the native AI. The validation was done in 49 games (25 against Protoss, 12 against Terran and 12 against Zerg), where the prediction accuracies against the three races were weighted equally.

Time period	**Current time period**	**3 time periods ahead**	**6 time periods ahead**
1			
2			
3	96,43 %		
4	92,81 %		
5	88,98 %		
6	87,76 %	85,71 %	
7	84,15 %	84,27 %	
8	80,49 %	80,83 %	
9	78,51 %	74,96 %	82,14 %
10	75,81 %	69,94 %	70,13 %
11	70,68 %	65,62 %	65,99 %
12	70,86 %	68,45 %	64,86 %
13	69,31 %	68,56 %	63,66 %
14	69,10 %	68,62 %	64,91 %
15	68,13 %	67,72 %	65,75 %
16	64,53 %	62,93 %	61,95 %
17	64,29 %	62,84 %	62,24 %
18	66,08 %	64,24 %	62,86 %
19	66,61 %	65,02 %	62,03 %
20	70,88 %	69,70 %	65,35 %
21	71,60 %	71,31 %	67,37 %
22	74,32 %	73,49 %	71,17 %
23	74,21 %	71,71 %	69,98 %
24	74,39 %	71,34 %	69,50 %
25	77,56 %	76,07 %	75,09 %

The table below shows the prediction accuracy when using a network trained against the native AI. 120 games (46 against Protoss, 43 against Terran and 31 against Zerg) of Odin playing against the native AI were analysed. Prediction accuracies against the three races are weighted equally.

Time period	**Current time period**	**3 time periods ahead**	**6 time periods ahead**
1			
2			
3	97,95 %		
4	96,13 %		
5	91,31 %		
6	90,90 %	90,23 %	
7	89,78 %	89,21 %	
8	88,25 %	87,70 %	
9	89,45 %	89,26 %	89,42 %
10	89,06 %	88,81 %	87,26 %
11	88,75 %	88,86 %	88,63 %
12	87,45 %	87,26 %	87,07 %
13	86,75 %	86,41 %	86,17 %
14	86,57 %	85,63 %	85,26 %
15	85,12 %	83,22 %	83,20 %
16	85,29 %	83,02 %	82,79 %
17	83,93 %	80,93 %	80,22 %
18	85,11 %	82,74 %	79,93 %
19	83,99 %	82,30 %	78,74 %
20	87,18 %	85,42 %	81,95 %
21	87,71 %	86,03 %	83,50 %
22	85,99 %	83,77 %	82,90 %
23	85,02 %	84,17 %	83,57 %
24	85,11 %	82,50 %	82,40 %
25	81,82 %	80,76 %	79,91 %