ASPIRE
Adaptive Strategy Prediction In a RTS Environment

By: Ruben Emil Oen

Department of Information Science and Media Studies

June 1, 2012

Abstract

When playing a Real Time Strategy(RTS) game against the non-human player(bot) it is important that the bot can do different strategies to create a challenging experience over time. In this thesis we aim to improve the way the bot can predict what strategies the player is doing by analyzing the replays of the given players games. This way the bot can change its strategy based upon the known knowledge of the game state and what strategies the player have used before. We constructed a Bayesian Network to handle the predictions of the opponent's strategy and inserted that into a preexisting bot. Based on the results from our experiments we can state that the Bayesian Network adapted to the strategies our bot was exposed to. In addition we can see that the Bayesian Network only predicted the possible strategies given the obtained information about the game state.

Acknowledgements

First and foremost I want to thank my supervisor, Weiqin Chen, for her support and guidance, it has been a pleasure working with you. I also want to thank my fellow students that always were there when I needed some input, and to keep me sane during the last two years.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Artificial intelligence in games has come a long way the last decade or so. But some game genres have been lacking behind when it comes to how good the AI for bot that have been implemented is. One of these genres is RTS, there are multiple reasons why this has happened. One of the reasons can be that these games have been lacking behind is that the campaign of these games doesn't really need a very sophisticated AI. A different reason can be that the game is made for a multiplayer experience, as well as it can be that it is really hard to make a bot that can compete with an experienced human opponent.

Every developer prioritizes differently and they all have their different reasons. Though it looks like there is a trend that the developers think that the AI for the bots in RTS games are not that important for the game experience and are not willing to commit much resources to this field. This is a negative trend because this means that more of the game experience is hinging on a good community and a good multiplayer solution more than is necessary. Not that having a good community and a good multiplayer solution is a bad thing, but there are times when one can't connect to the Internet or simply don't want to play against other people.

The result of the lack of investment in adaptive bots in RTS games is that there is no real challenge when playing the bot over time. It can be hard to start with, but when the player finds a way to beat the bot it will not adapt in future games and will still be vulnerable for that strategy. This means that a game against the bot will become trivial and few will play against it if they are not really new to the game, don't have Internet access or want to improve their skill in some way that does not need an opponent. One example of improving their skill is practicing new strategies or game mechanics like hot keys.

As stated above the current implementation of bots in RTS games is not that adaptive when it comes to counter different strategies. It may adapt to the players unit composition to be able to create some kind of challenge. For example if the player invests heavily in air units the bot may create units to combat that. But the bot will always adapt to the here and now. It will not recognize the players strategy and adapt to that. This is the core of the problem where the bot have only a few strategies and will only change the unit composition in these strategies, it will for instance attack after 5 minutes and after that it will attack every time it has x amount if units. Instead of figuring out what strategy the player is doing and adapt to it. Since the bot always has a very limited set of strategies it becomes trivial to play against the bot over multiple games.

Another problem with the bots in many RTS games is that to make the game more challenging they will let the bot cheat. The bot will for example often have access to the entire map and therefor know what the opponent is doing at any time as well as having a higher gathering rate pr. gathering unit. This is a problem because it can be demotivating to know that the opponent is actually cheating and does not have the same restrictions as the player has.

We offer a solution that focuses more on what the opponent is doing. This lets us ignore all the advanced techniques to build the bot, but rather how to supply the bot with better information so that it can make better decisions. Our solution is based upon a offline learning algorithm that learns between games and tries to predict what strategies the opponent might use. This will let the bot know what strategies it might be up against so that it can prepare for them.

1.2 Research questions

The guiding research questions for this thesis are:

1. How can we build software that will adapt to specific players preferences in strategy choices in a RTS environment over multiple games?
2. How can we build software that will adapt its strategy predictions over time to what strategies the player has used over multiple games?

1.3 Organization of the thesis

This thesis is organized in 6 chapters. First we start with the introduction, in this chapter we give some background information about our motivation for this work, what this work offer and finally we present our research questions.

In the second chapter we start off presenting an overview of the RTS genre. We continue to present a brief history of AI in games and selected techniques used in games. We finish of the second chapter presenting related AI research.

In the third chapter we present the game that is our platform for the work. We are also presenting the reasons for choosing this game. We are finally presenting the game related tools that we used in this work.

In the fourth chapter we present the design and development process. We start off with analyzing the problem, then we list the requirements for our software. We show the system architecture and present the different phases in the development process. Finally we talk about what problems we encountered during the development process.

In fifth chapter we present the training and testing of the Bayesian Network. We start off with listing our goals for the training and testing of the Bayesian Network. Then we present the settings for our training and testing, then we go through the procedure of the experiments, the analysis, the findings and finally we discuss our findings.

In sixth chapter we summarize the thesis, reflect on the work and suggest future work.

2 Literature Review

2.1 Overview of the RTS genre

RTS is a sub-genre of strategy video games that progresses in real time and is a type of war game. Normally these games are played from a bird's-eye perspective with some sort of action bars to show the essential information. From this bird's-eye perspective the player will be able to control all the units and buildings from his/her team. Since this genre progresses in real time it's important to be able to handle much information and react according to the information one has at a specific time and what ones strategy is.

RTS incorporate some form of strategy in a real time environment.In most cases the player is placed in an environment where there are limited resources, and multiple players has to fight over these. These games can be viewed as simplified military simulations, where the player has to balance its investments in economy, army and tactical placement of its army in real time. In simple terms the most common scenario in RTS games is that one have several players in a limited environment trying to secure resources by setting up an economy, building armies and leading them in to battle (Buro, 2003).

The more resources the player has the more army units the player can produce. This is one of the big problems players have to solve in RTS games. How much should the player invest in acquiring a better economy and how much should the player invest in acquiring a stronger army? If the player invests too much in a better economy, its army will be too weak to be able to defend a possible attack from one of its enemies. If it invests too much in a stronger army, and the player is not able to deal enough damage to the enemy, the enemy might have a better economy that will make the enemy able to acquire a stronger army in the future. There are multiple ways to invest in a stronger army, one can invest in a stronger army by creating already available units, build more infrastructure, research upgrades or develop more technologies that will unlock further upgrades and/or units.

2.2 A brief history of AI in games

Arthur Samuel had a desire to use computers for applications that would attract public attention. So he started his 25-year quest to build a strong checkers-playing program. The First working checkers program appeared in 1952 and where followed shortly after by a multiple chess programs (Schaeffer & Van den Herik, 2002).

Early efforts of Shannon, Samuel, Turing, Allan Newell, Herbert Simon and others generated interest in researching computer performance at games. Developing game playing programs were a major part in the early research field of Artificial Intelligence (Schaeffer & Van den Herik, 2002). One of the major challenges were to develop a world-championship-caliber chess program. At the time few realized the difficulty of creating such a program that should exhibit human-level "intelligence", and the early days of AI were plagued by over-optimistic predictions (Simon & Newell, 1958). In the 1970's to late 1980's the research field was concentrated on chess and the so-called brute force approach. It was shown that there was a strong correlation between search speed and the chess-program's performance. The consequence of this was that for a prolonged period of time the research field was largely devoted to building faster search engines, and the importance of developing a high-performance chess program seemed to fade (Schaeffer & Van den Herik, 2002).

First in the 1990's there was a series of dramatic successes for computer programs against top human players. The first game top human players were beaten by computer programs were checkers(8x8 draughts). This was in 1994 at the World Man-Machine Championship. The Program was called Chinook and was developed at the department of Computing Science at the University of Alberta (Schaeffer, 1997). This was shortly after followed by Deep Blue that got its victory over the world chess champion Garry Kasparov, and Logistello that that took home the victory against the world Othello champion Takeshi Murakami both in 1997 (Hsu, 2002) (Buro, 1997).

The introduction of low cost, high performance home computers has led to new forms of games, namely interactive computer games (also referred to as video games or digital games). These games introduced completely new forms of game-play and new genres. These new forms of games represented a departure from traditional games in game AI research , as these new games made use of increasingly complex and detailed virtual environments. This new form of computer games often incorporated human controlled characters and many computer controlled opponents. These environments requires an even more advanced and believable AI than traditional games. Examples of genres of this new kind of games are: First-Person shooter Games, Role-Playing Games and Strategy Games (Galway *et al.*, 2008).

2.3 AI techniques in games

2.3.1 Pathfinding

The most common decision in computer games is probably pathfinding. Pathfinding is to search for a good route for the object from one location to the next. This problem is not a trivial problem, but there are some well-established and solid algorithms that one can use to solve this problem. The best established algorithm for searching for optimal paths is A* (Stout, 1996). The A* algorithm works by minimizing the total estimated cost, the total estimated cost is the cost to arrive at the node plus the estimated cost to go from the note to the goal node. It then compares all the estimated cost from all the nodes that it can reach and chooses the lowest value. This will be repeated until it runs out of nodes to visit or it reach the goal node. This algorithm will find the optimal path as long as the heuristic is admissible. That the heuristic is admissible means that the heuristic never over estimates (Russell & Norvig, 2003).

An example of a game that uses A* is Age of Empires. They use A* to find the optimal path from one location to another. This means that when the player tells the unit to go from its location to another location, the unit should take the best possible path to its goal. The implementation of A* in games is a non-trivial task and if not implemented in a good way it can result in bad pathfinding. For example in Age of Empires when the player tells a group of units to go to a location that has a forest between the units and their goal. Some of the units will go around the forest but some of them can get stuck in the forest (Cui & Shi, 2011).

2.3.2 Scripting

In the early and mid- 1990's, most of the decisions in games where hard coded using custom written code. This method is fast and works well in small development teams, when the programmer also is likely to be designing the behaviors for game characters. As the development became more complex there was a need to separate the behavior design from the engine. Some developers moved to use other AI techniques and others continued to write their behaviors in a scripting language separate from the main game code. If the scripting language is simple enough level-designers or technical artists can create the behaviors.

A side effect of having scripting language support is that it gives the users the ability of creating their own version of the AI. If a game has a map editor that lets the users create own maps and script special AI for that specific map, the community can create modifications(mod) of the game. This lets the community create new game modes that the creators of the original game did not think of or did not want to use resources to develop. An example of a successful mod is Defense Of The Ancients which is a Warcraft 3 mod. This is a mod that created an entire new genre of games and shows how important it is to let the community loose to mod the original game. The possibility to create mods of the original game can extend their full-price shelf life beyond the 8 weeks typical of other titles (Millington & Funge, 2009).

2.3.3 Dynamic Scripting

Spronck *et al.* (2006) described Dynamic scripting to be an online competitive machine-learning technique for game AI, that can be characterized as stochastic optimization. Dynamic scripting is based on reinforcement learning and its architecture is quite similar to an actor-critic architecture (Spronck *et al.*, 2006). Dynamic scripting has one rulebase for each agent class in the game. This rulebase is used every time a new agent is generated to make a new script for that agent. This script contains the rules that control the agent. What rules that are used from the rulebase is defined by a probability that is attached to each rule. The probabilities of the rules in the rulebase are adjusted according to if the used rules have a high or low success rate and the weight-update function. The goal of this is to adapt the weights in the rulebase so that the expected fitness of the defined behavior of the generated scripts increases rapidly, even in changing environments (Spronck *et al.*, 2006).

In dynamic scripting when it generates its first script, it will generate the script by randomly selecting a number of rules from the associated rulebase. After this it will run the agent in the specified environment to check how it worked. This will again adjust the weights of the used rules. The rules that lead to success will have their probability weight increased and the rules that lead to failure will have their probability weight decreased. The next time it generates a script from the rulebase it will increase the expected fitness value (Spronck *et al.*, 2006).

2.3.4 Finite State Machines

The finite state machine is a set of states that can occur in the game connected by transitions. Each transitions leads from one state to another, these transitions are associated with conditions. If these conditions are met the transition is said to trigger, then the game will change it's current state to the target state. If no transition were triggered the game will continue in the same state (Millington & Funge, 2009). For each iteration, which is normally every frame, the state machine's update function is called. This method checks if any transitions from the current state are triggered. The first transitions that are triggered will then be put in use. It will then execute all the actions to get to the target state (Millington & Funge, 2009).

2.3.5 Bayesian Networks

A Bayesian Network is an acyclic graph where each node is annotated with quantitative probability information. The structure of the graph specifies the conditional relationships that is in the domain, when there is an edge from node A to node B it means that node A directly influence node B (Russell & Norvig, 2003).

An example is that we are developing a fighting game and we want to predict the next strike the player will throw. This way we can make a computer controlled opponent that can anticipate the strikes and defend or counter accordingly. Assuming the player can throw 3 kinds of strikes: punch, low kick and high kick. For every strike that is thrown we are going to calculate the probability for that strike given the two previous strikes. This will let us capture three strike combinations (Bourg & Seemann, 2004).

In figure 1 we call the first strike A, the second B, and the third C. We see from figure 1 that the second strike thrown is dependent of the first strike thrown, the third strike thrown is dependent of both the first and the second strikes. The combinations can be anything, an example of a combination is low kick - high kick - punch. This lets us give probabilities of what strikes comes based upon what strikes have been thrown before.

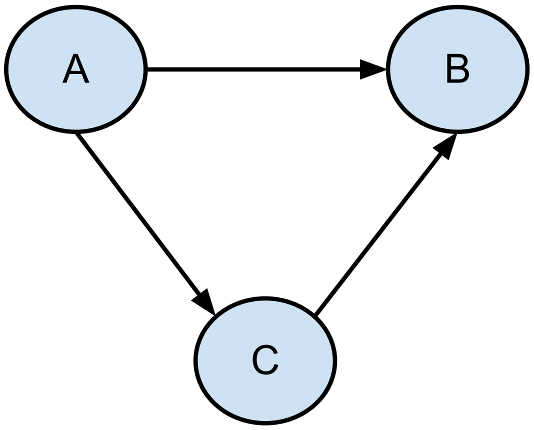


Figure 1: Strike network

2.3.6 Artificial Neural Networks

The inspiration for artificial neural networks came from the observation of biological learning systems that are very complex webs of interconnected neurons. ANN(Artificial neural network) is in analog built up from densely interconnected set of simple units. Each of these units takes in a number of real-valued inputs and produces one real-valued output. The inputs might be outputs from other units and the output might become inputs to other units (Mitchell, 1997). These units are normally called neurons by researchers. There have been attempts to use artificial neural networks in digital games for a number of years and the reason is that ANN is about learning.

An example of how artificial neural networks have been implemented is the program TD-gammon. This is a program that teaches itself how to play backgammon by playing against itself. TD-gammon uses temporal difference learning methodology. The combination of artificial neural network and temporal difference learning turned out to work very well and resulted in a game playing capability that is beyond current human capabilities (Tesauro, 2002).

2.3.7 Reinforcement Learning

Reinforcement learning is a learning algorithm that learns what to do by trial and error. For example we know that we can learn a computer to play chess by applying a supervised learning algorithm. But what if there are no teacher providing examples? How can the agent learn? By trying random moves the agent can learn a basic understanding of its environment. But without having some feedback on what moves is good and what moves is bad the agent will have no grounds for deciding which move to make. The agent therefor need to know if something good has happened or if something bad have happened. This type of feedback is called reward, or reinforcement. In some cases like chess the reinforcement will be received at the end of the game, since one does not know if a move is good or bad until the game concludes. In other cases like Ping-Pong the reinforcement will be received during the game, since each point is considered as a good behavior (Russell & Norvig, 2003).

2.3.8 Genetic Algorithms

Genetic algorithms(GA) are modeled after biological evolution. Instead of searching from general to specific hypotheses or simple to complex, GA' s are recombining and mutating parts of the of the best known hypotheses. Every step, called generations, is a set of hypotheses that are called population (Mitchell, 1997).

The implementation of genetic algorithms varies in their detail, but they generally share the same structure. The genetic algorithms generally work as follows: The algorithm iteratively updating the population(all the current hypotheses). For every iteration every individuals in the population are evaluated according to a defined fitness function. This fitness function can be for instance how many kills the individual had in a round of an FPS game. A new population is generated by selecting the best individuals from the current population. Some of the selected individuals are copied to the next population, while others are used as a basis for creating new hypotheses by applying genetic operations like crossover and mutation. This creates the next generation of hypotheses (Mitchell, 1997). The algorithm will repeat this behavior until a set amount of generations has past or that a hypothesis is valued to be good enough.

2.3.9 Online and Offline Learning

There are two different modes of learning. The first is online learning and the second one is offline learning. The difference between them in simple terms is that online learning learns while the player is playing the game, and offline learning is learning either between games or before the product is shipped. While online learning can adapt dynamically to the player's style it produces problems with predictability and testing. When the AI is changing all the time it can be difficult to replicate bugs, an example of this is that the AI decides that the best way to counter the players tactic is to run into a wall, it can be a nightmare to replicate the behavior (Millington & Funge, 2009).

The most of machine-learning in games is done offline and the most common way to use offline learning is to do the learning before the game is released. One of the reasons for this is that learning between games has many of the same disadvantages as online learning. Doing the entire learning process before the game is released means that the developers can use more unpredictable learning algorithms and test their results extensively (Millington & Funge, 2009).

2.4 Related AI research in RTS games

In recent years research has shifted from focusing on board games like chess to real time strategy games (Metoyer *et al.*, 2010). Typically commercial real time strategy games provides user with a set of opponent strategies, each encoded as a script. Aha *et al.* (2005) states that the main reasons why AI performance in RTS games is lagging behind compared to classic board games is that the RTS game world feature many objects, imperfect information, micro actions, fast-paced action and that the market dictated AI resource limitations and the lack of AI competition.

RTS games offer a large variety of AI research problems , unlike other games genres that are studied by the AI community so far. Buro & Furtak (2004) present 7 research problems that RTS games bring to the table. These are Adversarial real-time planning, decisions making under uncertainty, opponent modeling(learning), spatial and temporal reasoning, resource management, collaboration and pathfinding. All these problems make RTS games a very complex environment to make AI for. There have been several different approaches to make better AI in RTS in the later years. Some examples that will be presented in the following sections are strategy prediction by data mining

(Weber & Mateas, 2009), goal-driven autonomy (Weber *et al.*, 2010) , case based planning (Aha *et al.*, 2005), Evolutionary Learning (Ponsen *et al.*, 2005) and Monte Carlo Planning (Chung *et al.*, 2005).

2.4.1 A Data Mining approach to strategy prediction

Weber & Mateas (2009) presents a data mining approach to opponent modeling in strategy games. For their research they use the game StarCraft. One of the reasons for choosing StarCraft was that they could mine the data from a large number of available replays of games played by expert players. The StarCraft replays are stored in a proprietary, binary format. They used LordMartin Replay Browser to convert the replay files to game logs (Weber & Mateas, 2009). Weber and Mateas used several machine learning algorithms to strategy prediction and timing prediction. They represented strategy prediction as a multi classification problem, and the following algorithms were applied to classification: j48- C4.5 decision tree, k-NNNearest neighbor, Nnge- Non-nested generalized exemplars and LogitBoost- additive logistic regression. The different algorithms were evaluated at different time steps throughout the game. The results showed that different algorithms are better at different stages of the game (Weber & Mateas, 2009). Weber & Mateas (2009) have demonstrated that the machine learning algorithms they used outperformed the rule set classifiers in the initial stages of a game. One of the algorithms that they used was able to predict the opponent's tier 2 strategies with a 70 % precision five minutes into a game (Weber & Mateas, 2009).

2.4.2 Applying Goal Driven Autonomy to StarCraft

Weber *et al.* (2010) applied goal-driven autonomy(GDA) to StarCraft. During their work they developed a bot called EISBot. This bot plays complete games of StarCraft. They applied GDA to StarCraft to determine when new goals should be selected and to decide which goal should be pursued at each level. They used a reactive planning language called ABL(A Behavior Language) to implement the components specified in the GDA conceptual model. The first experiment evaluated EISBot versus the built-in bot of StarCraft, in this experiment the EISBot had a 73 % win rate. To ensure that the built-in bot used a variety of its scripts they ran 20 games on each map for each opponent race. The second experiment evaluated EISBot against human opponents. The Games were hosed on the international Cyber Cup(ICCup), a competitive ladder for StarCraft players (Weber et al. 2010). The ICCup have a point system similar to Elo rating system in chess. A player starts with 1000 point and gains points if he/she wins and loses points if he/she loses. The EISBot achieved a win present of 37 % against competitive humans, which gave the EISBot a score of 1182 after 100 games. The EISBot achieved a ranking of 33, 639 out of 65,646, that means that the EISBot outperformed 48% of competitive players (Weber *et al.*, 2010).

2.4.3 Learning to Win: Case-based Plan Selection in a Real-Time Strategy Game

Aha *et al.* (2005) present a case-based approach to developing a more advanced AI. They implemented their bot in Wargus, which is a clone of the commercial game Warcraft 2. They used 8 opponent scripts, some were publicly available, and others were manually developed. For each opponent script they used Ponsen & Spronck (2004)'s genetic algorithm to evolve counter strategy scripts that they used as a source of tactics. Their case-based approach to select what tactic the bot is going to use in each state uses an abstraction of the state space and state-specific tactics. By doing so they made the decision space small and possible to handle. They compared their bot versus its competitor on how many Wargus games it won. They used a fixed initial scenario, on a 128 x128 tile map, the games was set up as 1 versus 1 games. The results show that after 100 games their AI had a winning percentage of 82.4%. The best performance among the counter strategies had a 72.5% win rate, showing that their bot clearly outperformed the best individual counter strategy (Aha *et al.*, 2005).

2.4.4 Automatically Acquiring Domain Knowledge For Adaptive Game AI Using Evolutionary Learning

Ponsen *et al.* (2005) tried to automatically generate strong adaptive AI opponents for RTS games. They discussed the dynamic scripting technique and its application to Wargus. The core of what they did rotate around that domain knowledge is a crucial factor for dynamic scripting's performance. This is where they proposed a methodology that can generate automatically high-quality domain knowledge for use by dynamic scripting. The methodology they proposed was called AKADS(Automatic Knowledge Acquisition for Dynamic Scripting. This methodology uses an evolutionary algorithm to automatically generate tactics used by dynamic scripting. From there experiments they concluded with that evolved knowledge bases improved the performance of dynamic scripting against the four static opponents that were used in previous experiments (Ponsen & Spronck, 2004). They also tested AKADS against four strong new scripts, the results were very promising, which shows that dynamic scripting can adapt to many different static strategies. From their experiments they conclude with that it is possible to automatically generate high-quality domain knowledge that can be used to generate strong adaptive AI opponents in RTS games (Ponsen *et al.*, 2005).

2.4.5 Monte Carlo Planning in RTS Games

Chung *et al.* (2005) presents a framework called MCPlan. The contributions in this work are the following.

- Design of a Monte Carlo search engine for planning (MCPlan) in domains with imperfect information, stochasticity and simultaneous moves (Chung *et al.*, 2005).
- Implementation of the MCPlan algorithm for decision making in a real-time capture-the-flag game (Chung *et al.*, 2005).
- Characterization of MCPlan performance parameters (Chung *et al.*, 2005).

The result from their experiments shows that the program's play improves when the number of plans that is evaluated get higher. This has as expected diminishing returns as the number of plans get higher. They also discover that when they increase the number of units on the map the program have a higher win percentage. The reason for this is that when one increase the number of units on the map there are more decisions to be made and there are more opportunities to make "smarter" moves. When they change the map they discover that the more complex the map is the better the random player does, however the MCPlan player is a consistent winner. When the map is more complex there are more uncertainty to where the enemy units are located, this reduces the MCPlan player's simulations accuracy (Chung *et al.*, 2005).

When the MCPlan player is faced with less units that the opponent(random player) they show that given enough plans to consider the MCPlan player can overcome the random player with more men but a poorer AI. When optimizing the max distance look ahead in the plans they found out that a distance of 6 was the best, any longer and it would introduce more noise and errors. When the MCPlan player was faced up against an opponent that is rushing the flag it resulted in that on an empty map the rush-the-flag player won nearly all the games, when there is some obstacles on the map the MCPlan won more but when the map was even more complex the rush-the-flag player becomes a more challenging opponent. However with enough plans to consider the MCPlan won 55% of the games.

3 Platform

3.1 The Game: StarCraft Brood War

StarCraft Brood War is an expansion set to the original StarCraft developed by Blizzard Entertainment (Blizzard, nd*b*). StarCraft Brood War was released in the United States on 31 March 1998 (Blizzard, nd*a*). The game have received many patches up through the years both fixing bugs and balancing the game, making it one of the most successful and well known RTS game in the world. It also have a reputation of being easy to learn and hard to master. StarCraft Brood War have had one of the biggest professional gaming scenes when it comes to RTS games, does also had much television coverage in South Korea for many years.

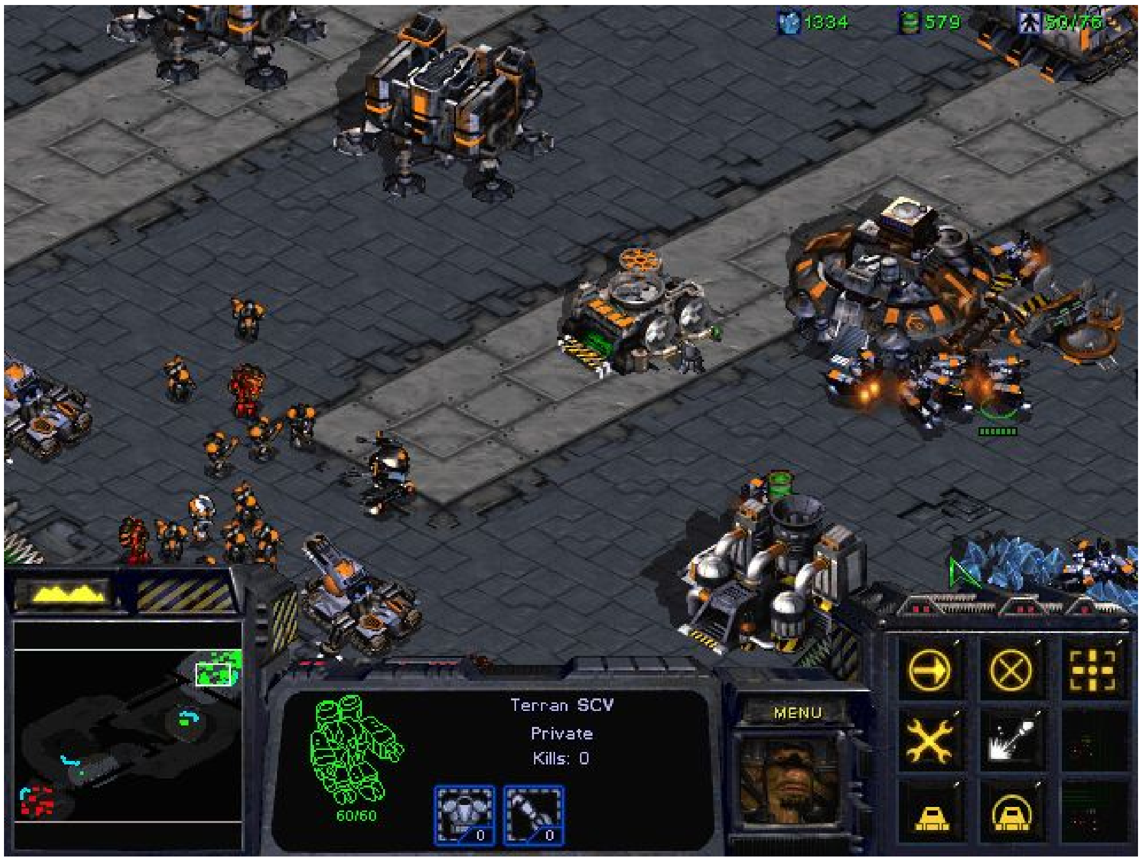


Figure 2: StarCraft Brood War - Race:Terran

The game has 3 different playable races, this is Zerg, Protoss and Terran all with their own history. These three races work quite differently from each other. This difference facilitated many different play styles and room for experimentation with different strategies. Figure 2 shows screenshot of a Terran base.

The Zerg are a bug like race that is focused around biological units and buildings and does not use any technology. Zerg units are typically less expensive than their counterparts that the two other races have. Zerg is considered to be the most mobile race. One thing that is special with the Zerg is that it produces all its units from one building and produces buildings by morphing workers. Zerg are because of this an extremely flexible race because they have only one kind of production building (Liquipedia, 2009*c*).

The Protoss are a race of humanoids that are considered to be the most technologically advanced race in the StarCraft universe. Protoss have typically more expensive units than the two other races. Therefor play styles that focus around having fewer but more powerful units are good for this race (Liquipedia, 2009*b*).

The Terran is a future version of the human race, this race has typically more expensive units than Zerg but less expensive than Protoss (Liquipedia, 2008). Therefor this race is some sort of middle way between the Zerg and Protoss when it comes to gameplay.

3.2 Why did we choose StarCraft?

The reasons for choosing this game as a platform for this project many and are as follows:

- StarCraft Brood War is a very balanced game. This means that the units that each race has to their disposal are not doing too high damage or have too much health. This facilitates games that focus on having good strategies and not just racing for that one very good unit.
- StarCraft Brood War is a well-played game and has had a big user base including a professional scene, this means that there is many strategies and well thought out play styles in the community. This will make it easier to test and develop the AI.
- StarCraft Brood War is well known and that there is an annual competition on AIIDE which started in 2010, and therefore an even better platform to use since it is a game that other

people and researchers are also working on.

- There is already developed a third party API to the basic game functions. This is the biggest reasons why this game was chosen as a platform for this project. The third party API that will be used is an open source project that is called BWAPI and is developed under the GNU Lesser General Public License. This API is well developed and is still currently under development adding on new functionality and fixing bugs.
- There is many replays of expert level of play that one can download from different community sites and convert to game logs and use in a training setting. This makes it much easier to create big training sets by just downloading the replays and convert them instead of actually have to make the training sets from the ground up.
- StarCraft Brood War contains many real world problems such as reasoning about economics, strategic and tactical goals. This makes the results of this research applicable not only to video games.

3.3 Concepts in StarCraft Brood War

In StarCraft Brood War there are many of different concepts that one should know to understand the depth of the game. Many of these concepts are true for other RTS games as well but we will explain them in the setting of StarCraft Brood War. These concepts will be from all aspects of the game, from what that is the goal of the game is to how different mechanics affect the game. Some of these concepts have many different definitions since they are community based and operate in somewhat unclear area when it comes to the specifics. Team Liquid is one of the biggest international communities when it comes to StarCraft. It's this community's wiki we have used to define most of the concepts.

- Fog of war in StarCraft Brood War is a mechanic that makes ones units and buildings only have a certain radius where they can see and report back about what they see. The implementation of fog of war is that the player cant see what happens outside of its units and buildings visual radius. This mechanic makes the player do decisions with incomplete information.
- The goal of most RTS games is to acquire undisputed supremacy of the resources on the map one are placed. This is done in most games by killing all the units and buildings and hereby

destroying the opponent.

- Team play in StarCraft Brood War is a very important aspect of the general gameplay if one is playing with other people on one's team. If the team the player is on does not play as a team or have a good team strategy there is a big probability that they will lose.
- Multiplayer in RTS games mostly revolve around 2 or more people that tries to acquire their team undisputed supremacy of the map.
- Single player in RTS games can be a bit different from multiplayer. In most RTS games there is a campaign mode where one can play the game and go through a history. It is also possible to play the same kind of games that one can play in multiplayer, just that one plays against bots instead of other players.
- Technology is an important concept in StarCraft Brood War. This concept is about what buildings and units that are available to the player in a given state. When a game starts the player has a given level of technology, some of the buildings that the player can build will unlock other buildings and units. This is the mechanic makes the player choose what buildings it will build and what tech he/she will unlock so that he/she can acquire more advanced and more powerful units to combat the enemy.
- Units are what the army consists of that is not building. An example of a unit can be a marine that is trained from a barracks.
- Buildings is what that will build ones units, upgrade ones units, getting a certain technology or be a building that can attack buildings and units.
- There are three races in StarCraft Brood War. These are Terran, Protoss and Zerg. The different races have different set of buildings and units, this makes it possible for 6 different match-ups if one doesn't count what the player is playing. If one takes in to account what the player is playing there is 9 match-ups (Liquipedia, 2010) . The different races have a different mechanics to them. An example of this is that Zerg are morphing all the units they can make from larva that spawns from their hatcheries, while Terran and Protoss builds their units from different buildings. Zerg have to morph one worker into a given building, in this case Terran have to designate a worker to build the building and will be busy for the time it takes to build the building. Protoss designates a worker to start warp in(constructing) in the building and is free to leave to do other tasks after the warp in is started.

- Meta game in StarCraft Brood War is any planning, preparation or maneuvering that a player does outside of the actual gameplay to gain an advantage. There is three major branches, first there is preparation done before a match to exploit current trends in StarCraft. Second is preparation to exploit an opponent's or map's style of play. Third is strategic decisions designed to exploit a players reaction or weakened mental state in the future. The last one is also called mind games (Liquipedia, 2011*a*). When one refers to how the metagame evolves its namely how these 3 branches develop. What new strategies are discovered and how other players react to them and how strategies that before was hard to survive now are well understood and easy to defend.

- The game is played in a specified environment, this is called a map. There many of different maps that the players are able to play against each other. Blizzard even has supplied a powerful map editor so that the players are able to make the maps they want to play in addition to the maps supplied with the game.

- A spawn location is where the player starts the game on a map. On some maps there are multiple spawn locations so that the player does not know where the opponent is at the start of the game.

- The resources in StarCraft Brood War that one use to build buildings and units are minerals and vespene gas. Minerals are the main resource and are the only resource needed to produce low tech units and buildings. Vespene gas is mainly used for higher tech units, buildings and upgrades in addition to minerals. One mineral patch contains a specified number of minerals and can be mined directly by a worker, when this patch is completely mined out the mineral patch will vanish. To be able to gather gas one has to place a special building on the vespene gas patch. When this patch is emptied it becomes depleted. One can still gather vespene gas from this patch but at a lower efficiency (Liquipedia, 2011*b*).

- Normally one talk about 3 main time periods in a game, these are early, mid and late game. The definitions of when one ends and the other starts can be hard and there are many different versions. But generally one can say that the early game starts at the start of the game and ends when the players natural expansion is up and running (Liquipedia, 2009*d*). The mid game starts where the early game ended and when the player is making use of their tier two tech (Liquipedia, 2009*a*). The late game is the rest of the game after the mid game.

- There are 3 different tier units. The tier one units are the units one can build without having to build any buildings on the "advanced" page (or Lair for Zerg) (Liquipedia, 2012*b*). Tier two units are units that only require one building from the advanced tab . Tier three is the units at the top of the tech tree (Liquipedia, 2012*c*).

- Expanding is the act of building a new base close to new resources to be able to gather these resources.
- A strategy is what you intend to do to win the game.
- A Worker is the most basic unit for each race, its main functions are to create new buildings and to collect resources.(Liquipedia, 2009*e*)
- Scouting is the act of revealing information about areas of the map to gain information about what your opponent is doing and intend to do (Liquipedia, 2011*c*). Since StarCraft Brood War is a game where one has incomplete information, the act of scouting is very important

3.4 Tools for further development of StarCraft Brood War

The tools and libraries that we are going to use are BWAPI that supplies an API to StarCraft Brood War, BWTA for terrain analysis, BTHAI that is a multi-agent open source bot, Chaoslauncher that injects the BWAPI into the game process and BWAI that is a collection of different scripts for the built in bot.

BWAPI: The library is an open source project that aims to serve an complete and stable framework to create bots for StarCraft Brood war in C++ (BWAPI, 2012). In a nutshell one can:

- Write competitive bots for StarCraft: Brood War by controlling individual units (BWAPI, 2012).
- Read all relevant aspects of the game state (BWAPI, 2012).
- Analyze replays frame-by-frame, and extract trends, build orders and common strategies (BWAPI, 2012).
- Get comprehensive information on the unit types, upgrades , technologies, weapons, and more (BWAPI, 2012).

- Study and research real-time AI algorithms in a robust commercial RTS environment (BWAPI, 2012).

BWTA: The name stands for Brood War Terrain Analyzer and is an add-on for BWAPI that analyzes the current StarCraft map (BWTA, nd).

BTHAI: This is an open source bot that use BWAPI and BWTA. It is based on a flexible and expandable multi-agent architecture. BTHAI is very modular so it is easy to change specific behavior without having to think of everything else. It is mainly designed and written by Johan Hagelbäck at the Belkinge Institute of Technology, Sweeden (BTHAI, 2012).

Chaoslauncher: Chaoslauncher is a program that serves a gui for a number of StarCraft plugins (Liquipedia, 2012*a*). We used it to inject the BWAPI into the StarCraft process.

BWAI Launcher: BWAI Launcher is a program that allows you to play against many different community made scripts (poiuy qwert, 2009).

4 Design and Development

4.1 Analysis of the Problem

What is the problem with building software that adapts to specific player preferences in strategy choices in a RTS environment over multiple games? And why is it hard to adapt the strategy predictions over time to what strategies the player has done before?

To be able to adapt to specific players preferences in strategy choices we have to know something about the player and how he/she plays. Since we have incomplete information when playing a game, we need to be able to analyze how the opponent played in some other way. Since StarCraft Brood War can save replays, these replays are a good source for information about how the game were played with complete information. To be able to classify how the opponent is playing we need to define what is a strategy. Defining when a strategy begins and when it ends can be very hard since there are often an over reaching theme during the entire game. When knowing what kind of strategies the opponent has used before, how can we with incomplete information decide what strategy we believe the opponent is doing in a game? Since the opponent can do a number of different strategies only knowing what strategies the opponent have used before only helps us to a certain extent. We need a way to exclude strategies that the opponent can't do and find the strategies that the opponent is more likely doing based on the information we have. When we know what strategies the opponent is more likely to use we only have to supply that information to a bot that can act on that information.

When formalizing the problems we can state the following:

- We should be able to analyze how the opponent is playing post game.
- We should define and classify the opponents strategies.
- We should be able to use the known knowledge to better predict what the opponent is doing.
- We should be able to put the predictions in use.

In these four problems there are two of them that require any form of AI. The two problems that require AI is to be able to use our know knowledge to better predict what the opponent is doing and to put those predictions in use. The two other problems can be solved by simply generating a log of what the opponent is doing based on the replays and manually defining the rules that will create the strategies based upon the logs and putting them into a database.

Our main focus is to be able to use our knowledge about the opponent to better predict what he/she is doing. Putting This information in use can be done in multiple ways and will not be our main focus in this work.

So how can we predict what strategy the opponent is doing during a game? We have incomplete information about what he/she is doing in the current game, but we know what he/she has done before. The information we have in a game will vary quite a bit, we might have gotten much information from our scouting or it might be very little. Since we do not know what information we have access to and when we get access to it we have to be able to make good predictions given the information we are given.

There are many different AI techniques out there, but what is most suited for our needs? Earlier in the literature chapter we presented different AI techniques that are used in game AI. We will discuss if any of these will be able to help us solve our problems and what that would solve it in the best way.

- Search algorithms are good at tasks like path-finding where one have good heuristics. Can we use an algorithm like A* to solve our problem with predicting the opponent's strategy? A* is well suited for environments where we have heuristics that never under estimates the cost of getting from one state to another. Since we can't know what strategy the opponent is going for we can never say with certainty that we have found the opponents strategy. This makes it hard for us to use a search algorithm to solve our problem. However it would be central when implementing the actual path-finding in the game.
- Scripting is a well-known technique that gives the bot a certain set of things to do. It is however not adaptive in any way and is predefined by the creator. This makes it not well suited for our needs.
- Dynamic Scripting is based around telling the bot what to do and is based on reinforcement learning and its architecture is quite similar to an actor-critic architecture. Dynamic scripting got a rulebase for each agent class in the game and will generate a script based upon the success of the rules in previous encounters.

- Finite State Machines are a set of states that can occur in the game connected by transitions. This is suited where we can easily define that one game state transition to another. This however does not help us when we need to predict the opponent's strategy.
- Bayesian Networks is able to give us a framework where we can tell state what variables are dependent of other variables. This could make us able to state what that is dependent of what, and what that is not. It also lets us state evidence that a certain node is in a certain state, but we can also let it stay blank. It will then calculate the probability of the different states depending on the training set.
- Artificial Neural Networks is based upon observations of biological learning systems. We can create a network that would take in all the different variables of the game state and train it to give a certain strategy as output but it will be hard to for instance getting out a ranked list of all the possible strategies.
- Reinforcement Learning is a learning algorithm that learns by trial and error. We could in theory implement reinforcement learning to learn how to play StarCraft but since we are looking at how to predict the opponent's strategy it would to be hard to implement this feature with this technique.
- Genetic Algorithms is based around evolution. It could evolve with the right weight to predict a strategy based upon the information at hand. But it would be problematic to adapt between games since there is a long training time.

So what of these techniques will help us the most? Search algorithms, Scripting, Dynamic Scripting and Finite State Machines are all techniques that could be used in making a bot for RTS but when it comes to making strategy predicting software they are not the most interesting. If we look away from these techniques we have Bayesian Network, Artificial Neural Network, Reinforcement Learning and Genetic Algorithms left. Genetic Algorithms would first of all take too long time to retrain between the games to get a decent result. As for Artificial Neural Network, Reinforcement Learning and Genetic Algorithms we could make a solution that would try to predict the opponents strategy but since there are so many different variables that we do not have any information about in the game and that the problem is generally about dependent probabilities we think that using Bayesian Network for our strategy predicting software would be the smartest thing. The Bayesian Network handles unknown information very well and gives us a list with probable strategies without much work. The only thing we have to do is to model the dependencies.

4.2 Requirements

- The bot needs to adapt to what the player is doing in a specific game.
- The bot needs to adapt to what the player is doing over multiple games.
- The bot recognizes different strategies.
- The bot can use different strategies.
- The bot remembers what the player has done before.

4.3 Design

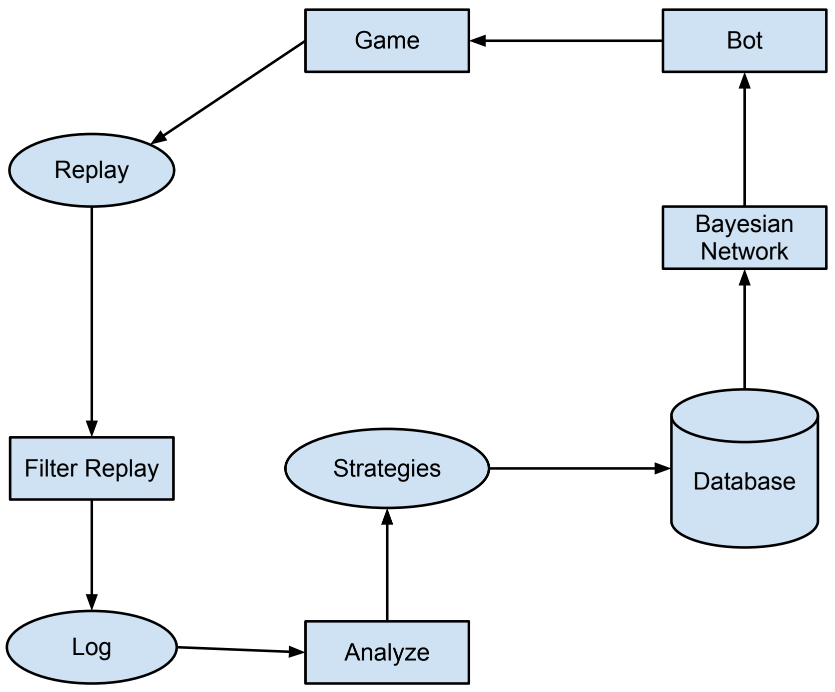


Figure 3: System Architecture

Figure 3 is a representation of the system architecture. The process is circular but it starts at the Game box, this box represents the actual StarCraft Brood War game. The output from the game box is the replay of the played game, this is a log of all the recorded actions the players have made during the game (we can collect multiple replays before starting on the next step). We take the replay and filter out the data that we need and put them into a log. Then we analyze the log to find strategies in the logs and put them in a database. We then train the Bayesian Network based upon the strategies we have in the database. This trained Bayesian Network is inserted into the bot that uses it for strategy prediction in a played game and the process starts over.

4.4 Phases in Development

4.4.1 Creating the replay logs

First task in the development phase was to generate logs of the StarCraft Brood War replays from the TSL(Teamliquid StarLeague) ladder that we had collected from the Team Liquid Forum (Hotbid, 2008). For the sake of speed we tried to read the replay files directly without having to load them into StarCraft Brood War. For this we used a library called bwrep (bwchart, nd*a*), but after some trial and error, we found out that the StarCraft Brood War replays record all the players actions. This includes actions that are invalid (bwchart, nd*b*). An example of an invalid action is that the player tries to build a unit that he/she does not have the resources to build. This is the reason why we decided to load all the replays into StarCraft Brood War and use BWAPI to get access to the information we needed, instead of filtering what is valid actions and what is not. The drawback with this solution is that the replays has to be ran in StarCraft Brood war. This puts certain constraints on us. Most noteworthy is that the maximum speed that a replay can be ran in StarCraft Brood War is 16x which is much slower than what one could read directly from the replay file.

BWAPI throws events when certain things happen in the game, an example of this is when a unit is created and destroyed. These events can be caught if one is interested in them. In this case we was interested in when different units and buildings was created, morphed and killed so we gathered that information with a time stamp. In addition we collected the name of the map, starting positions, the matchup and what player won. We wrote the information to two files one for each player. The only difference between the files is that the unit creating and morphing of units and buildings for the opposing player is represented only by that a unit/building was created, it is not recorded what kind.

4.4.2 Analyzing the replay logs and generating strategies

Since we already have done some work with the replay logs we got all the information we need in a raw format that needs to be analyzed and put in a form that we can use. We started with making text files that represents the different strategies that the players use in what matchup, in what game and if the player won or lost. The strategies are represented by what the given player have of buildings and units, time stamp, matchup, map size and what game period the attack was in. we recorded the number of the buildings but only if the player had a unit or not.

We defined that there are 3 game periods early game, mid game and late game. This is a widespread way of talking about different game periods in the StarCraft community, however people also often use early/late in front of each period. But we defined it as 3 game periods for simplicity reasons. We defined that the game starts with the early game. After the first attack it goes in to the mid game. After the next two attacks it goes in to the late game. This was done with the definition in mind and was adapted to be more easily implemented.

We defined an attack as when 3 units died in a row independent of the player in the replay logs without anything in between. The attack continues until there have been 15 other actions in a row that is not a unit getting killed or a building getting destroyed. If there have been 14 lines of other actions and then a unit is killed there have to come 15 new actions in the log before the attack is over.

We record the strategies for every attack we record. So when the first attack comes we record how the strategy for both the players as the early game. The two next attacks are mid game strategies, after this all attacks are late game strategies.

The way we represent a strategy is a string with two parts separated by "|", both parts of the string have its values comma separated. The first part is 15 values for Protoss, these values are the number of each different structure built by the player. The two other races have different numbers of values in the strategies, this is because of the number of buildings and units the race can build/train. The second part is 13 values, these values is one for each unit Protoss can build. These values can only be 1 or 0, these values means if there is that kind of unit on the map or not. The reason for this is to not over specify the strategies. In addition to the strategy string we also recorded what game period and matchup the strategy is viable in. What game period the strategy is in is represented by a number between 0-2, 0 for early game, 1 for mid game and 2 for late game. The matchup is represented by a string, for example PT for Protoss vs. Terran.

An example of a strategy string is 8,0,1,0,1,1,0,0,1,1,3,4,1,1,0|1,1,1,0,1,0,0,0,0,0,0,1,0. Where 8,0,1,0,1,1,0,0,1,1,3,4,1,1,0 is the different buildings that the player has. The positions for the first part of the string mean Gateway, Stargate, Robotics Facility, Robotics Support Bay, Cybernetics Core, Observatory, Fleetbeacon, Arbiter Tribunal, Citadel Of Adun, Templar Archives, Assimilator, Nexus, Forge, Photon Cannon and Shield Battery. This means that the example string has 8 Gateways, 1 Robotics Facility, 1 Cybernetics Core, 1 Observatories, 1 Citadel Of Adun, 1 Templar Archives, 3 Assimilators, 4 Nexus', 1 Forge and 1 Photon Cannon. The positions for the last part means Zealot, Dragoon, Dark Archon, High Templar, Dark Templar, Archon, Arbiter, Scout, Corsair, Carrier, Reaver, Observer and Shuttle. The last part of the example string is 1,1,1,0,1,0,0,0,0,0,0,1,0, This represents that there has been spotted at least one of the following units on the map: Zealot, Dragoon, Dark Archon, Dark Templar and Observer. The strategy is a late game strategy that is used in the Protoss vs. Terran Matchup.

In addition to making the strategy files we also put the information into a database. We also put what strategies are used against other strategies in what game period as well as counting the how many times the different strategies are used in the different matchups. When analyzing a replay that is played by the bot against a player all the strategies are counted 10 times for every time they occur. This is done to be able to speed up the adapting process to a given player. The strategies that occur less than three times in the database after all the replays are analyzed will be deleted to remove strategies that might be faults or not used strategies.

The last thing we did was to generate the different scripts that the BTHAI bot needs as input. The way we did it with the build orders was to simply detect every time there was a building built and then write it to the correct build order file. With the squads we had to count the units before an attack and then write it to the correct squad file. There is also a script with upgrades, we decided that it was out of scope and just used the standard one with a few manual changes. Finally we connected the squad and build order files to the appropriate strategy in the database.

The software that we use for our database is SQLITE. SQLite is an in-process library that implements a self-contained, server-less, zero-configuration, transactional SQL database engine. This means that unlike most other SQL databases, SQLite does not have a separate server process and it reads and writes directly to ordinary disk files. The whole database is contained in a single disk file. The database file is also cross-platform. SQLite is developed by Hwaci and its source code is decided to be in the public domain by its developers(Hwaci, nd).

4.4.3 Designing the Bayesian Network

We designed our Bayesian network(see figure 4) using GeNie. Genie is a graphical interface to SMILE (Decision-Systems-Labaratory, nd*b*) that allows us to design a Bayesian Network in an easy way and load it into ones software. SMILE stands for 'Structural Modeling, Inference, and Learning Engine' and implements graphical decision-theoretic methods, such as Bayesian Networks. it's made by a research group at the Decision systems laboratory at the Department of Information Science and Telecommunications and the Intelligent Systems program at the university of Pittsburg (Decision-Systems-Labaratory, nd*a*).

In the designing process we decided to design the Bayesian Network around what information is the most easy to scout in any given game. This means that since information about some of the units the opponent have is relative easy to scout, it's just to move a unit to the opponent's base and one will see at least some of the units that the opponent got at the front and if one is lucky some buildings, we started with letting the presence of a unit affect the likelihood of the tech structure that is required to build it. The tech structure is then again affecting the other tech buildings that are required to build it. In addition all structures and units are affected by what game period the game is in. All the different tech structures are then affecting one of three different tech nodes, one for each tech tree, which again affect the strategy node. The assimilator affects the nexus node because of that it's a resource gathering building that is normally placed besides a nexus. The nexus node affects the strategy node directly because of its economic role.

In simple terms the Bayesian Network in figure 4 is the tech tree in StarCraft Brood War and what game period the game is in at a given point in time. The Bayesian Network has two more nodes, namely strategy and counterStrategy, they serve as output nodes. The strategy node is the node that outputs how likely the different opponent strategies is. The counterStrategy is how likely it is good to use the different counter strategies given the state of the network. The network is a simplified version of the problem.

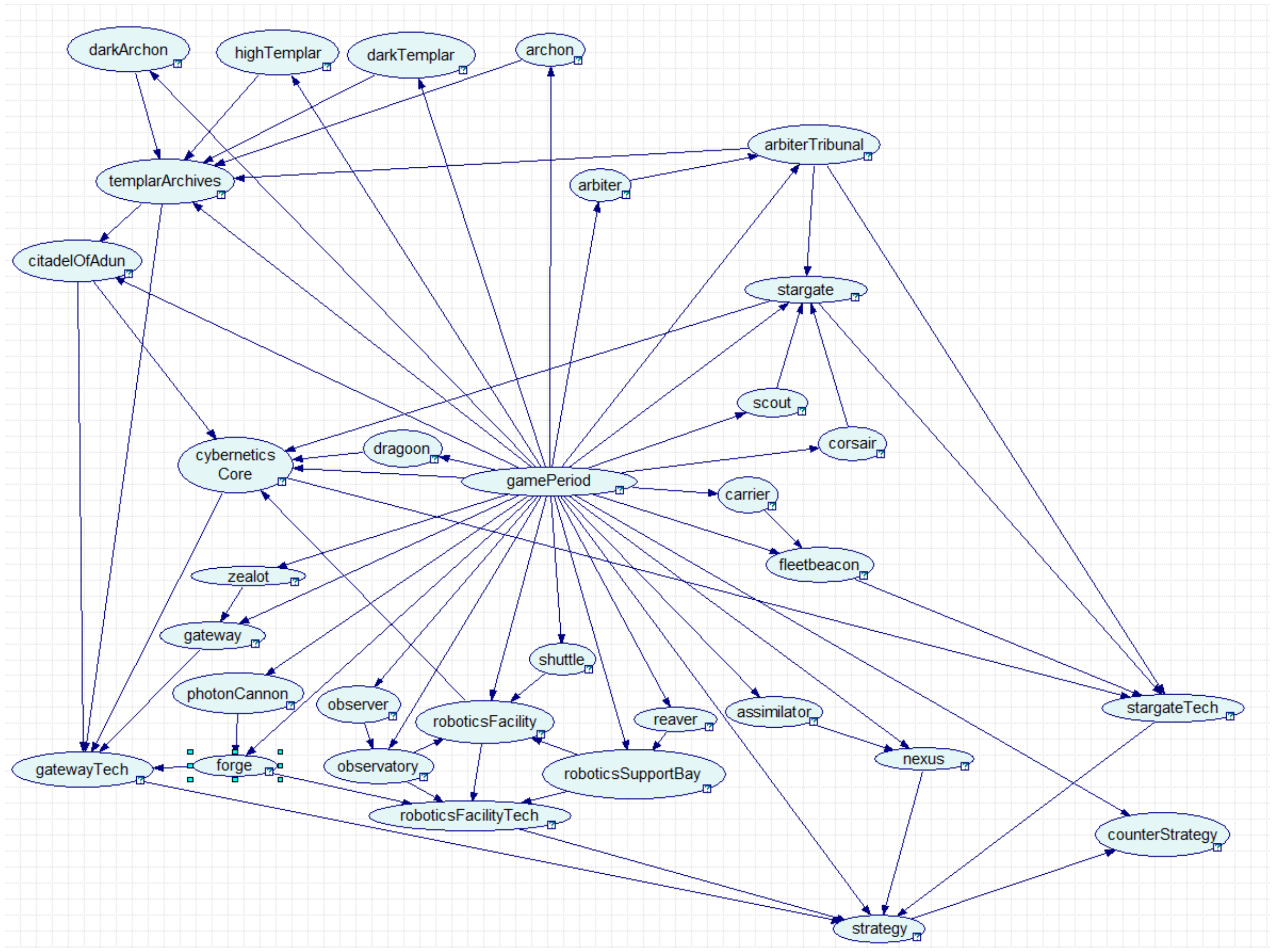


Figure 4: Final Bayesian network

All of the unit nodes have true or false states, not the exact number of units, this is also correct for most of the buildings. This is done because one very rarely know exactly how many of a certain unit the opponent have, so it's much easier to state that there is a kind of unit on the map if one see it. The nodes that have true/false states function as the input nodes in the network in addition to the gamePeriod node. This lets us take the scouted information about the opponents units and buildings, and get probabilities for what strategies that the opponent can do.

4.4.4 Generating training data

Generating the training data for training the Bayesian Network is rather simple since most of the work is already done when we analyzed the replay logs. All the information is already in the database, so this just has to be written to a file in the right format. The format for the training set is defined by SMILE, since this is the library that will handle the training of the Bayesian Network.

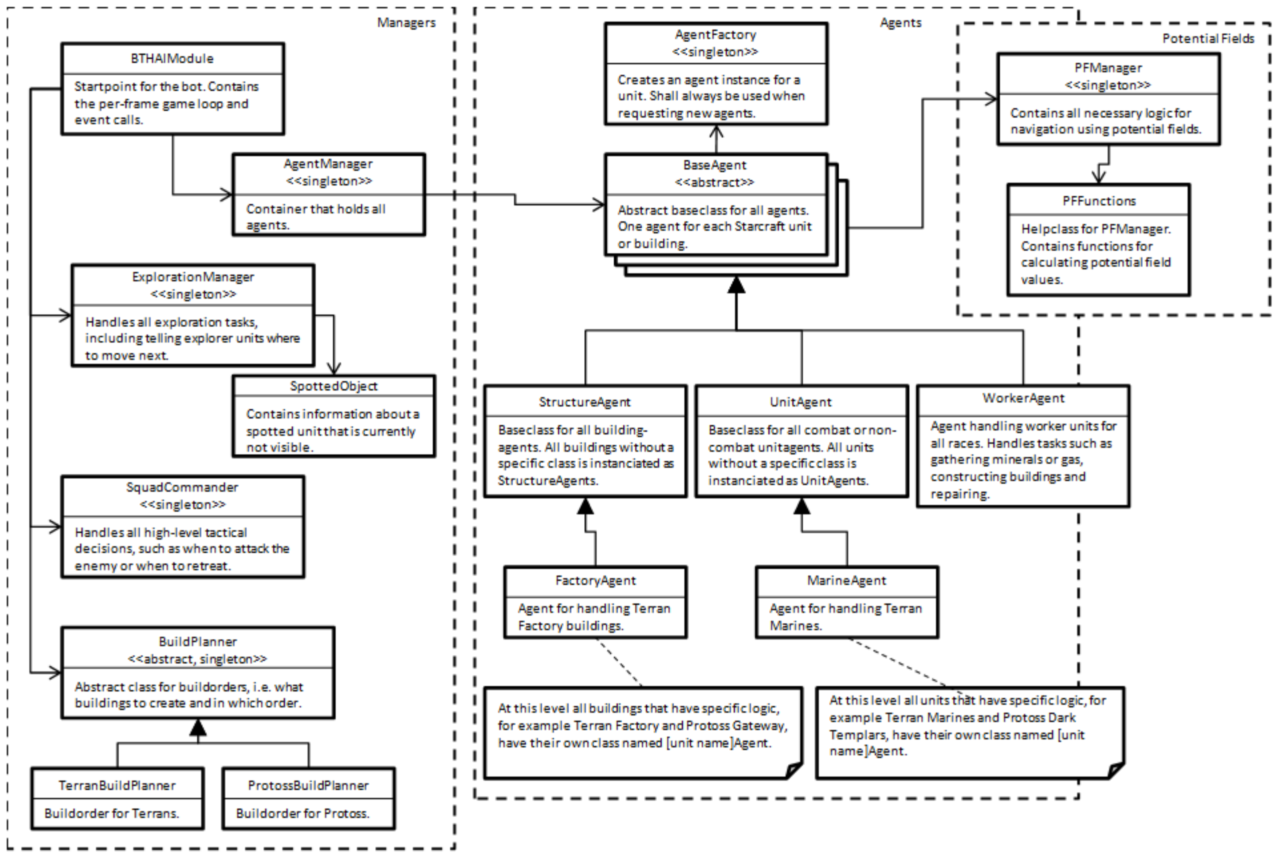
The format of the training data file is the following. The first line will include the names of the different nodes in the network. The following lines are the occurrences of the different strategies, one per line. This means that if a strategy is used 10 times it will be written to 10 lines. The strategies are represented by their values for the different nodes in the Bayesian Network separated by white spaces. So what we had to do was to find all the different strategies that were recorded in replays and how many times they occurred from the database and write them to the training data file.

4.4.5 Adapting BTHAI bot

BTHAI is a bot that have a multi-agent architecture (see figure 5) and that is based around being modular so it is easy to expand and change specific behaviors (BTHAI, 2012). As standard the bot uses 3 scripts to tell the bot what to do and when to do it. These scripts are build order, squads, and upgrades. The build order script is simply what buildings it should build and in what order.

The squad script is what units it should build, what role the different groups have and what priority they have. The upgrades script is what upgrades it should research and what priority they have. The BTHAI is highly modular so we can do changes in one part of the bot and not having to think how that would affect other parts of the bot as long as we do not break the interface. The modularity and that BTHAI uses simple scripts as inputs is the reasons for choosing this bot.

To be able to use the bot with the Bayesian Network we had to do some modifications to the bot itself. As standard the BTHAI bot requires one of each script at the start of the game, we needed to be able to change the scripts during the game to be able to adapt to what the player was doing. We also had to load the Bayesian Network, collect information about enemy units and buildings and put it into the Bayesian Network. We had some problems with implementing these changes but more on that later.



Managers
BTHAIModule
Startpoint for the bot. Contains the per-frame game loop and event calls.
AgentManager <<singleton>>
Container that holds all agents.
ExplorationManager <<singleton>>
Handles all exploration tasks, including telling explorer units where to move next.
SpottedObject
Contains information about a spotted unit that is currently not visible.
SquadCommander <<singleton>>
Handles all high-level tactical decisions, such as when to attack the enemy or when to retreat.
BuildPlanner <<abstract, singleton>>
Abstract class for buildorders, i.e. what buildings to create and in which order.
TerranBuildPlanner
Buildorder for Terrans.
ProtossBuildPlanner
Buildorder for Protoss.
Agents
AgentFactory <<singleton>>
Creates an agent instance for a unit. Shall always be used when requesting new agents.
BaseAgent <<abstract>>
Abstract baseclass for all agents. One agent for each Starcraft unit or building.
StructureAgent
Baseclass for all building-agents. All buildings without a specific class is instanciated as StructureAgents.
UnitAgent
Baseclass for all combat or non-combat unitagents. All units without a specific class is instanciated as UnitAgents.
WorkerAgent
Agent handling worker units for all races. Handles tasks such as gathering minerals or gas, constructing buildings and repairing.
FactoryAgent
Agent for handling Terran Factory buildings.
MarineAgent
Agent for handling Terran Marines.
At this level all buildings that have specific logic, for example Terran Factory and Protoss Gateway, have their own class named [unit name]Agent.
At this level all units that have specific logic, for example Terran Marines and Protoss Dark Templars, have their own class named [unit name]Agent.
Potential Fields
PFManager <<singleton>>
Contains all necessary logic for navigation using potential fields.
PFFunctions
Helpclass for PFManager. Contains functions for calculating potential field values.

Figure 5: BTHAI architecture (BTHAI, 2010)

The events that BWAPI throws when a unit or building is discovered was already caught by the BTHAI bot so the only thing we had to do here was to include the code that sent that information about the unit or building to the Bayesian Network. When new information about the players units or buildings is entered in the Bayesian Network we have to update it to see if the bot have to change scripts. If this is the case it has to generate new scripts based upon the new suggested script from the Bayesian Network and what buildings and units it already got and then load it into the bot.

The code below shows one crucial method in our adaptation of the BTHAI bot, it shows the implementation of how we check if it is needed to change scripts based on the strategy prediction from the Bayesian Network, if it is needed it will change them. This code is running every time there is discovered a new unit or building. It also runs whenever the game period is changed.

```
void BTHAIModule::calculateStrategy(){
    network.UpdateBeliefs();
    DSL_node *counterNode = network.GetNode(network.FindNode("counterStrategy"));
    DSL_Dmatrix *matrix = counterNode->Value()->GetMatrix();
    int strategyID = 0;
    double highestStrategyProb=0;
    int matrixSize = matrix->GetSizeOfDimension(0);
    for(int i =0; i <matrixSize; i++){
        if(highestStrategyProb<matrix[0][i]){
            strategyID=i;
            highestStrategyProb= matrix[0][i];
            }
    }
```

The code block above is first updating the beliefs in the Bayesian Network, then it's getting the counterStrategy node. This node contains all the different counter strategies that the Network can suggest. It then looks for the strategy with the highest probability and assigns the value of that strategy id to a variable for later use.

```
if(currentStrategyID!=strategyID){
    currentStrategyID=strategyID+0;
    string racesStrategyID = "counterPStrategyIds";
    string racesStrategyFileID = "strategyToFileID";
    FileReaderUtils utils = FileReaderUtils();
    std::string databasePath ="\\requiredFiles\\db.sqlite";;
    char databasePathChar[200];
    strcpy(databasePathChar, databasePath.c_str());
    Database* db = new Database(databasePathChar);
    std::string dnaQuery = "SELECT counterStrategy,gamePeriod FROM "
                          + racesStrategyID
                          + " WHERE id= "
                          + boost::lexical_cast<string>(strategyID);
    char dnaQueryChar[200];
    strcpy(dnaQueryChar,dnaQuery.c_str());
    vector<vector<string>> dnaResult= db->query(dnaQueryChar);
```

The next part checks if the new strategy ID is different than the strategy ID that is currently in use, If this is not the case the method returns. If the new strategy Id is different from the one that is currently in use the method continues. It then open the database and runs a query that finds the counter strategy and the game period based on the new strategy ID.

```
vector<string> row = dnaResult[0];
std::string fileNameQuery = "SELECT id FROM "+racesStrategyFileID
                          + " WHERE dna='"+row.at(0)
                          + "' AND gamePeriod= " + row.at(1);

char fileNameQueryChar[200];
strcpy(fileNameQueryChar,fileNameQuery.c_str());
vector<vector<string>> fileNameQueryResult= db->query(fileNameQueryChar);
int randomFileNumber;
if(fileNameQueryResult.size()>1){
        int maxNumber = fileNameQueryResult.size()-1;
randomFileNumber = rand() %  maxNumber + 0;
}else{
        randomFileNumber=0;
}
```

The next part takes the counter strategy and game period that the last block found and runs a query that finds the ID for the files that contains the new scripts. Since there can be multiple files that is connected with the same strategy it selects a random file id from the query result, if there is only one file it selects that one.

```
        setScripts(fileNameQueryResult[randomFileNumber].at(0));
        db->close();
        Commander::getInstance()->reassignUnitsToSquads();
        BuildPlanner::getInstance()->computeActions();

    }
}
```

The last part sets the new scripts, closes the database and reassigns the different units on the field to new squads and compute the new actions the BuildPlanner have to do.

4.5 Encountered problems in the development process

As in any development process we also hit obstacles. Here we will go through some of the obstacles we encountered.

The way actions are saved in the replay files made it hard to read them directly. The reason for this is that all actions like build a certain unit or building is logged even if it did not result it that unit or building being built. This resulted in that if we should read the files directly we had to filter what actions actually resulted in that unit or building being built and what did not. This would require us to write much of the game logic, instead of doing this we decided to acquire the information in the replay files by loading the replay files into the game and use BWAPI to receive the events and information that we needed. This solution is not the best since it takes much longer to load the replay into StarCraft Brood War and run it on max speed which is 16x. But this should not represent a problem for game developers since they either could make replay files that only contain valid actions or apply the game logic on the replay files. This should greatly increase the speed.

BWAPI have to be compiled on the v90 compiler because of external dependencies. This made finding compatible libraries that we needed harder.

We had some problems with stability with the StarCraft Brood War related libraries. The game crashed seemingly at random points when the bot was running. Often this was before the 10 minute mark. Normally people trust the libraries they are using to a certain degree so it's normal to start debugging their own code. After a while we started to take a closer look at the different libraries that we are using. Since the StarCraft Brood War related libraries are open source it was possible to go into the code and read line by line. This took a while locating the bug since it could be in one of three libraries but we finally found some bugs and managed to make the bot more stable.

There was some problems making BWAPI and SMILE working together since BWAPI is compiled with checked iterators and SMILE requires that one disable the checked iterators. After recompiling BWAPI with unchecked iterators we started to have problems with getting BWAPI to work properly. Normally one can make AIs by compiling to both dll and exe. When we recompiled with unchecked iterators dlls stopped working. After a while we decided to adapt BTHAI to be able to run as an external exe file instead of a dll file. We had some problems with resetting the BTHAI bot between games. This was not a problem when it was running as a dll file, since it was reloaded between the games.

Changing scripts in the middle of the game was more challenging than we hoped. We discovered that because of the multi agent structure of BTHAI it was not easy to make all the agents reset and use the new scripts. In addition to this it was challenging to get the bot to make use of the existing structures and units on the map when we changed the scripts. We got this however to work at a sufficient level to be able to perform our experiments.

5 Training and testing of the Bayesian Network

In the training and testing, we have the following goals:

- Find out if the Bayesian Network learned to predict the different strategies given incomplete information.
- Find out if there is some difference between how the different experiments adapted to the players.
- Find out if the Bayesian Network can exclude some strategies based on what information it got.
- Find out if the Bayesian Network can be used in a full game environment with success.
- Find out if the bot is adapting to what the player is doing in a single game.

5.1 Setting

The process is designed to test the adaptation in the Bayesian Network and to show that the Bayesian Network's precision of strategy prediction increase. The bot outside of this is not in scope and we will not be looking at the general performance of the executing of the strategies.

There are two different setups, both of them starts with the same Bayesian Network that is trained from data mined replays from the TSL ladder collected from team liquids forum.

- First setup will have the structure of playing 5 games against one script, retrain and play 5 more game against another script. This is done until it has run against 4 different scripts.
- Second setup will have the structure of playing 5 games against the original built in AI, retrain and play 5 more games against the original built in AI. This is done 4 times.

5.2 Procedure

The data procedure of the experiments is as follows. We started with analyzing all the replays of expert players that we downloaded from team liquid. From this we trained what would be our default Bayesian Network, database and scripts that would be the basis for our two experiments. The first iteration in the experiments we ran the bot for 5 games against a script called Ashara P v1.2. We made logs of the replays, analyzed logs to find what strategies was used, generated a dataset for training the Bayesian Network and retrained the network while saving the old one into another directory. This was done 3 more times against different scripts. These are called Rocky 2 Protoss v2.6 non-cheating, TaranokAI P v 1.14d and Raynor p v2.1. These are all community created scripts that is included with a program called BWAI Launcher (poiuy qwert, 2009). In this experiment there were played a total of 20 games.

The second experiment was done in the exact same manner but against the original scripts for the built in bot instead of different scripts every iteration.

The collection of data that we used in our analysis was done by saving the different trained Bayesian Networks, the databases and the replays of the played games. We saved the database at the end of each experiment to be able to find out what the different states on the strategy node is. This was our raw data, we have used these networks and databases to gather data about how the different networks developed and performed during the experiments. We have gathered the top 4 likely states in the strategy node based on 4 different sets of known information. This has given us a good overview about how the different parts of the experiments affected the Bayesian Networks.

We have also manually watched random replays to check if the bot behaved as intended. The reason behind this was to check if the bot actually used the output from the Bayesian Network or not. This way we can with a reasonable certainty state if the bot used the Bayesian network by putting in the information it acquired during the experiments and using the output it got or not.

5.2.1 Analysis

The data that we acquired from the data collection, that we will be analyzing is the 9 different versions of the Bayesian Network, 2 different databases and 40 replays. We have analyzed how the different Bayesian Networks differ from each other and how they evolved during the course of the experiments. This is where the main data source is, we have included the databases and replays to ensure that there have been no errors during the experiments.

To illustrate how the probabilities in the Bayesian Network evolve after each iteration in the experiments we included two sets of pie charts, namely figure 6 and 7, one figure for each experiment. In these figures one can see how the probabilities for the different strategies were in the default Bayesian Network and how they changed after each iteration in the different experiments. In this case the only information that is given to the Bayesian Network is what game period the game is in, namely early game. The resulting pie charts are a representation of the different probabilities of all the different strategies.

These pie charts show that there has occurred adaptation during the experiments. Figure 6 and 7 are too general to be able to say something about how they have adapted when more information is available, it can only give us a bird's-eye view of how the probabilities of the different strategies changed over the course of the experiments. Figure 6 and 7 does not say anything about how different input to the Bayesian Network will affect the output. So to be able to get better information about how the Bayesian Network changed we gave the different versions of the Bayesian Network 4 sets of different information about a fictional game state to get more information about how the Bayesian Network adapted and what it adapted to. One thing that can be said when looking at figure 6 and 7 is that there are some strategies that get a higher probability over the course of the experiments. This is because that in every game there is an early game period and there will be used an early game strategy in this period by definition. This is reflected in the different strategies that comes up in figure 6 and 7 and how the probabilities change.

To acquire more concrete data about what strategies is probable and what that is not we decided to make tables to show the 4 most probable strategies given certain information. This was done to all the different versions of the Bayesian Network to get a complete image of how they evolved when exposed to different strategies. This information will be presented in tables that show the different strategies, their ranking in the different experiments and what iteration in the given experiment it was in. We limited the tables to the four most probable states, the reason for this is that we are able to show how the Bayesian Network changes behavior after each of the iteration, while not showing too big tables.

We have transformed the figure 6 and 7 to table 1 so that one can see how that data looks like in the table form. The actual percentage of each possible state is not that interesting. This is because

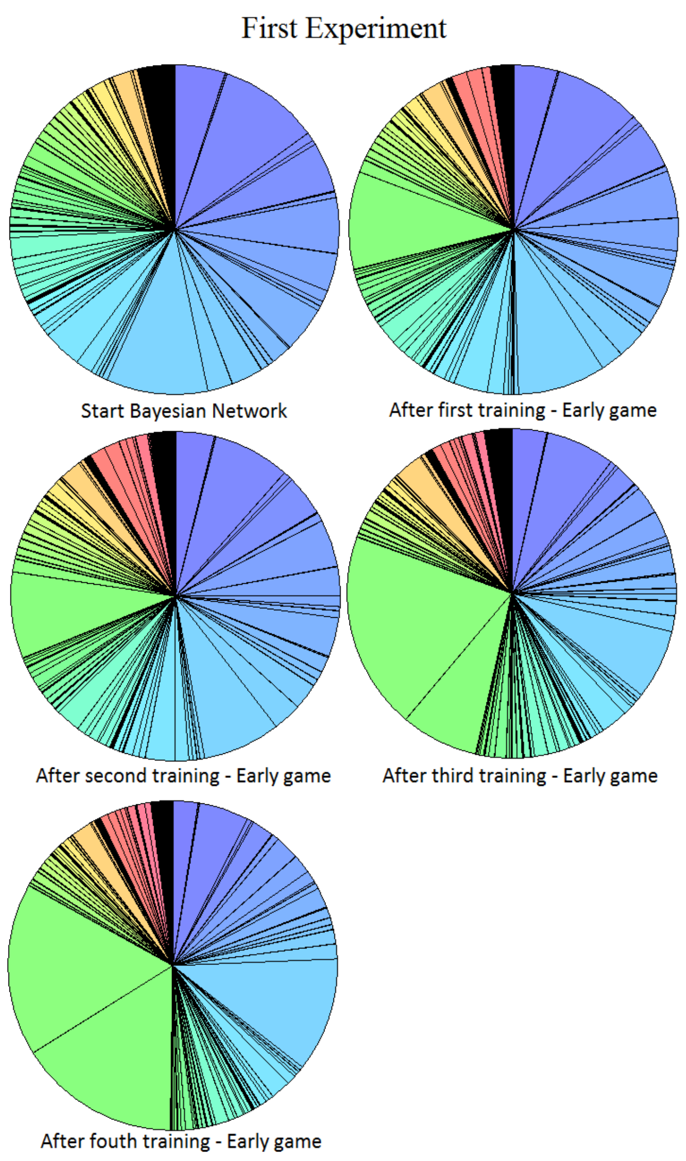


Figure 6: First experiment results with early game as only know information

the bot only needs to know what strategies it needs to take into account when selecting strategy.

In table 2-4 we have supplied more information than in table 1. This information is about what units have been spotted on the map. in table 2 and 3 we have stated that Dark Templars have been spotted. This unit is a unit that has high melee damage and that is invisible if one does not have detection. In table 4 we stated that High Templars have been spotted. This is a unit that has no

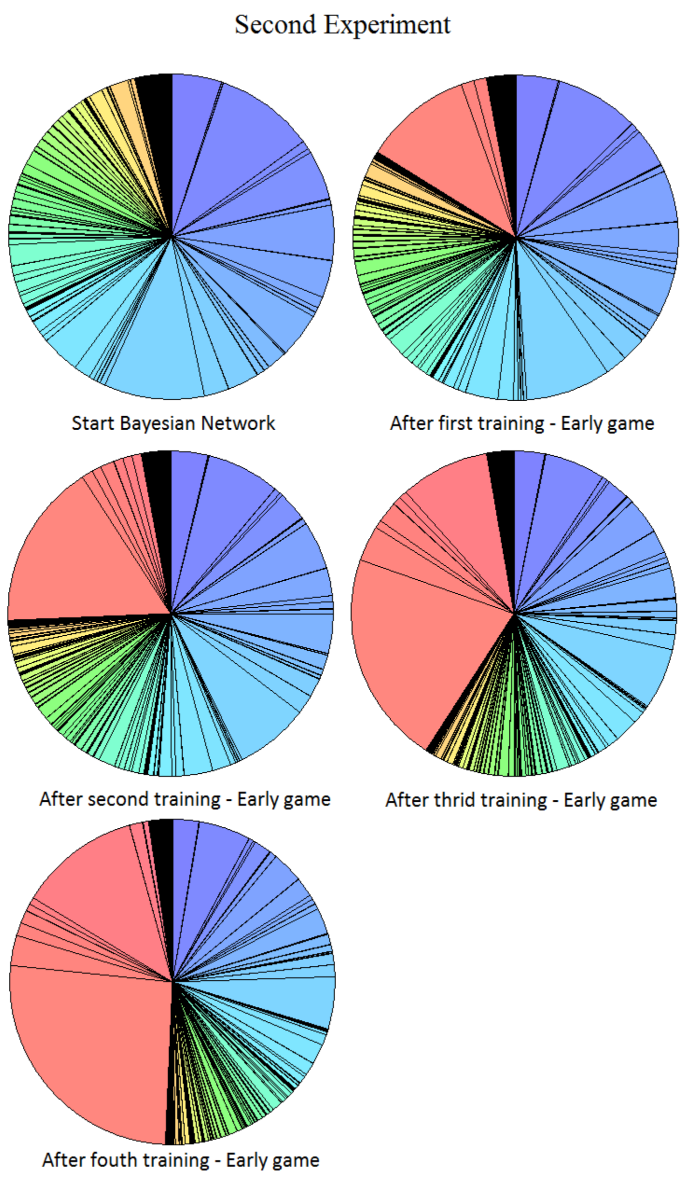


Figure 7: Second experiment results with early game as only know information

melee attack but has powerful spells.

Table 2 is the four most probable strategies given early game and that Dark Templars have been confirmed. We can see that there is a difference between table 1 and 2. First of all there are different strategies that are in the top 4. One interesting part is that in the case that we have confirmed that the opponent is building Dark Templars and we are in the early game, we can see that there has

Table 1: Four most probable states given early game.

Experiment 1					
Rating	Start values	1. training	2. training	3. training	4. training
First	State 45	State 139	State 139	State 141	State 141
Second	State 6	State 45	State 45	State 139	State 139
Third	State 19	State 6	State 6	State 45	State 45
Fourth	State 2	State 19	State 19	State 6	State 6
Experiment 2					
First	State 45	State 263	State 263	State 263	State 263
Second	State 6	State 45	State 45	State 271	State 271
Third	State 19	State 6	State 6	State 45	State 45
Fourth	State 2	State 19	State 19	State 6	State 6

Table 2: Four most probable states given early game and Dark Templars.

Experiment 1					
Rating	Start values	1. training	2. training	3. training	4. training
First	State 23	State 23	State 23	State 23	State 23
Second	State 155	State 155	State 155	State 98	State 155
Third	State 171	State 171	State 171	State 83	State 171
Fourth	State 98	State 98	State 98	State 155	State 98
Experiment 2					
First	State 23	State 265	State 270	State 270	State 265
Second	State 155	State 23	State 269	State 269	State 270
Third	State 171	State 155	State 265	State 265	State 269
Fourth	State 98	State 98	State 23	State 23	State 56

been no change to the order of the first four strategies after two iterations in experiment 1. This either means that there have not been used any strategy that uses Dark Templars in the early game or that the strategies that have been in use do not change the internal order of top four strategies. In experiment two we can see that there is more change in the top four strategies. Here we can see that the Bayesian Network have adapted more to the opponent than in experiment 1. This is because there have been used strategies that include Dark Templars in the early game or that internal ranking did not change based on what strategies that was used.

When we look at what strategies that were used when we state that the game period is mid game instead of early game and maintain that Dark Templars is confirmed there are a completely different set of strategies that is predicted by the Bayesian Network (see table 3). This is expected behavior since mid-game strategies are a different subset of strategies than early game strategies. These

Table 3: Four most probable states given mid game and Dark Templars.

Experiment 1					
Rating	Start values	1. training	2. training	3. training	4. training
First	State 77	State 77	State 272	State 272	State 272
Second	State 125	State 125	State 125	State 125	State 125
Third	State 99	State 99	State 77	State 77	State 77
Fourth	State 67++	State 67++	State 99	State 99	State 99
Experiment 2					
First	State 77	State 77	State 77	State 77	State 77
Second	State 125	State 125	State 125	State 125	State 125
Third	State 99	State 99	State 99	State 99	State 99
Fourth	State 67++	State 67++	State 67++	State 67++	State 67++

Table 4: Four most probable states given late game and High Templars.

Experiment 1					
Rating	Start values	1. training	2. training	3. training	4. training
First	State 258	State 258	State 275	State 275	State 275
Second	State 169	State 169	State 169	State 258	State 258
Third	State 232	State 232	State 258	State 169	State 169
Fourth	State 128	State 128	State 232	State 232	State 232
Experiment 2					
First	State 258	State 258	State 258	State 258	State 169
Second	State 169	State 169	State 169	State 169	State 258
Third	State 232	State 232	State 232	State 232	State 232
Fourth	State 128	State 128	State 128	State 128	State 250

two sets are disjoint, meaning that there are no strategies that are in both sets. In table 3 there is little activity, the only change between the iteration is that state 272 moved to first place and that state 125 stayed in second place pushing down state 77 and 99 after the second iteration in the first experiment.

In table 4, the information given is that it is in the late game and that High Templars have been spotted. This case behaves like the other cases there no change if there have not been used any strategy that fits the given parameters and a change if there has been used one. These data points supports that the software is working as intended on a functional level.

Since we exposed the Bayesian Network to different strategies during the experiments we can see that the probabilities have adapted to what strategies it have been exposed to. In table 1 we can see that there has been a change in the top 4 strategies during the experiments. The results tells us that after the 2 first iterations in experiment 1 there is a new strategy that has taken first place as the most probable strategy. This strategy is state 139, the representation of the strategy is 3,0,0,0,1,0,0,0,0,0,1,1,0,0,0|1,1,0,0,0,0,0,0,0,0,0,0,0. This means that there is following buildings on the field: 3 Gateways, 1 Cybernetics core, 1 Assimilator and 1 Nexus, the units that are in use is Zealots and Dragoons. After the two next iterations there are a new strategy at first place, this is state 141. The difference between state 139 and state 141 is that in state 141 there is one less Gateway, one more Nexus and that there are not any Dragoons made.

We can see there is a similar behavior in the tables 2-4, however there are some parts that there has been no or little change. One place that there has been no change is in the second experiment in table 3. The reason for this is that there have been no games played that used Dark Templars in the mid game or the internal ranking is not changed. In table 4 we can see that there have been changes in the probability of the different strategies in the late game as well.

When we compare table 1 and table 2 we can see that there are different strategies in the top 4. This is because that using Dark Templars is not used that often and when the Bayesian Network is supplied with only information about the game period and does not know if there is Dark Templars made the probability that they are made is low, so other more used strategies are in the top 4. When we supply the information that there are Dark Templars made the Bayesian Network can exclude all the strategies that do not include the tech required to build Dark Templars.

When looking at the different tables 1-4 we can see that there have been adaptation to what strategies the bot have been exposed to and that it handles incomplete information quite well. When there is little information available it will show the most probable strategies given that information, it cannot exclude a many of the strategies but when there is more information available it will exclude the strategies that does not fit the information that is given.

The two experiments differ by what scripts they were playing against, this was done to see if there was any difference when the bot is exposed to different set of scripts for every iteration and when it is exposed to the same set of scripts every iteration. When comparing the two experiments we can't find any significant difference between the two experiments at any point of the experiments. Figure 6 and 7 and tables 1-4 show no evidence that there are any difference between how the Bayesian Network learn. The only difference we can see is the strategies that are used at what time.

When it comes to excluding different strategies when given certain information about the game state, the data in table 1-4 shows that there are different strategies that are in the top 4 when given different information. This means that the strategies that fit the given information will have a higher probability than the ones that does not, the strategies that does not fit with this information will have a much lower probability and will therefore be easy to exclude from the possible strategies. In figure 6 and 7 we can see that some parts of the pie charts are black or have very small probability, the black parts of the pie charts are multiple states that have very low probability are besides each other. These low probability strategies could easily be excluded from what strategies that should be taken into account. Excluding different strategies does not only come from what information that is known about the game state but also what the opponent have done in the past.

The Bayesian Network did not produce any false positives during the experiments. This means that there was not produced any strategies with high probability that did not contain the tech required to produce all the units and buildings in the given information. All the top strategies were strategies that were possible given the information that was supplied. All of the strategies that were in the Bayesian Network were also in the given replays, either from the downloaded replays or from the played games in the experiments.

We discovered that when the Bayesian Network was exposed to a certain building and unit mix that it had not come across before it did not have any strategies that had high probability, but just distributed the probabilities between all the different strategies since it did not have any strategy that fit the given information. This however was fixed when it was trained next time, since the replay of the given game was included in the training set and therefor was learned by the Bayesian Network. When it comes to the practical function of the Bayesian Network into the bot it worked flawless, it was able to take in new information from the bot and update its values and output them to the bot.

The implementation of the bot itself did work. We used the BTHAI bot as a basis of our work, we had to do some changes to the bot on a functional level to make it work for our purpose. This meant that we were using it in another way than intended and that produced some problems. The problems resulted in problems like that the bot did not adapt as good as we would like. When going through the replays we did notice that the bot did never change scripts when it discovered a new unit, it only changed when the game period changed. This is however a problem associated with the bot and not our strategy predicting software and is therefore out of scope. The only way this affected our strategy predicting software was that the bot often lost at the first attack and therefor did not get that much experience about what the player would do in the mid and late game.

5.2.2 Findings

- In our experiments we found that the predictions in the Bayesian Network changed over time and that it adapted to the strategies that it were exposed to. It handled incomplete information better over time as seen in figure 6 and 7. These figures shows that different strategies are expected when it have been playing against an opponent.
- The two different experiments did not show any notable different behavior other than that there were used other strategies in the different experiments.
- From the data at hand we can say that the Bayesian Network lowered the probabilities of the strategies that do not fit the given information about the game state to next to zero. This however will not give a list of excluded strategies, it will instead give a list of strategies that the bot can take into account.
- When looking only on how the Bayesian Network performed in the experiments, we can conclude with that it worked on a functional level. The Bayesian Network did what was intended. It received and delivered information, it also did not slow down the game at all. Actually we performed the experiments at the fastest game speed that BWAPI allowed us.
- We found out that the bot itself did not use the given information from the Bayesian Network as well as we would like. This however was not the fault the Bayesian Network but how we implemented the Bayesian Network into the bot. We discovered that the only actual changing of scripts did only happen when the game period changed and not whenever there was a change in the probabilities of the top strategy.

5.3 Discussion

How do our findings fit into our goals with the training and testing, and how did they help answer the research questions? We found out that the Bayesian Network changed and adapted to the strategies that it were exposed to and that it handled incomplete information better over time. We also found that the Bayesian Network lowered the probabilities to the strategies that did not fit the known information about the game state to such a low probability that they would easily excluded from the probable strategies. From this we can state that the Bayesian Network learned to predict the possible strategies used by the player. This however is not an exact prediction, the strategy would only be in a list of probable strategies with the most probable on top. We found that there was no notable difference between the two experiments. It is positive that there was not any notable difference because that shows that the system is robust and will handle a situation with a player that plays many different strategies and play styles and is not limited to a small set of strategies.

The bot that used our Bayesian Network worked but not as well as we would like, but worked good enough to show the potential of our Bayesian Network. We would like to see a better bot implemented with a Bayesian Network for strategy prediction.

When it comes to our research questions, did our findings answer them?

The first and most general research question was as following: How can we build software that will adapt to specific players preferences in strategy choices in a RTS environment over multiple games? Based on our findings we can say that our bot did adapt to the strategies that was used against it and changed its strategy to counter the opponent.

The second research question was as following: How can we build software that will adapt its strategy predictions over time to what strategies the player has done over multiple games? Based upon findings we can say that our solution learned what strategies the opponent has used in the past and how often he/she used those strategies.

Based on this we can conclude with that one way to make the kind of software talked about in the research questions is with our approach.

6 Conclusion and Future Work

6.1 Summary

When the work with this thesis began we started off with a problem in RTS games that we wanted to help solve. This problem was that the bots in RTS games did not adapt to a player between games. This made it trivial to win against the bot when the player found a strategy that worked, it was just to repeat it and he/she would win. We wanted to make a bot that would adapt to what strategies the player was using. This would make the player change his/her strategy to be able to win consistently over multiple games.

This resulted in two research questions:

- How can we build software that will adapt to specific players preferences in strategy choices in a RTS environment over multiple games?
- How can we build software that will adapt its strategy predictions over time to what strategies the player has done over multiple games?

6.1.1 Background

We started out looking at a brief history of AI in games in general and how AI in games have developed from its beginning in 1952 up to present day, presenting a number of different AI techniques and how they have been used in game, and presented a number of related AI research in RTS games.

The game that was chosen as a platform for our work is StarCraft Brood War. The reasons for choosing the game was the following

- The game is very balanced.
- The game is had a big professional scene with many replays that we could data mine.

- There is a third party API to the basic game functions.
- There is an AI competition in StarCraft Brood War.
- The game contains real world problems like, economy management, strategic and tactical goals.

6.1.2 Problem Analysis

When analyzing the problem we found that to be able to adapt to a player's strategy preferences we had to know something about the player and how he/she plays. We came up if the following formalizing of the problem:

- We need to be able to analyze how the opponent is playing post game.
- We need to define and classify the opponent's strategies.
- We need to be able to use the known knowledge to better predict what the opponent is doing.
- We need to be able to put the predictions in use.

For these problems we decided to use a Bayesian Network. Bayesian Network was mainly chosen because it gives us a framework for stating dependent probabilities and that it handles unknown information well.

Having our formalization of our problem in mind we stated the following requirements for our software:

- The bot needs to adapt to what the player is doing in a specific game.
- The bot needs to adapt to what the player is doing over multiple games.
- The bot recognizes different strategies.
- The bot can use different strategies.
- The bot remembers what the player has done before.

The design of the information flow in our software is as following. We take replays from played games, create logs of the information we need from these replays. We analyze these logs to find strategies, then print these strategies to the training set and train the Bayesian Network with this training set. The trained Bayesian Network is used with the bot playing new games. The replays of these games are analyzed and the Bayesian Network is retrained based upon this data. A model of this is shown in figure 3.

6.1.3 Development Process

There was 5 phases in the development process. The first thing we did was to generate logs of StarCraft Brood War replays. We tried to read the files directly but discovered that the replays included all the action that the players did during the game, this resulting in many invalid action being recorded, for instance because missing resources. Then we decided to use BWAPI to dump all the valid commands that we were interested in, such as unit discover and when units die, into two files one for each player.

When we had the logs with the information we needed, we started the second phase. This phase was to analyze the logs and to generate the strategies that should be recognized and used by the software. We started to analyze the log files that were created in the last phase, looking for the strategies that were used. When looking for strategies, we looked for when there were an attack. When we recorded that an attack was in progress we gathered the information about what units and buildings the players had before the attack started and wrote it to two files, one for each player. The strategies are represented by the buildings, units, time stamp, matchup and map size.

In addition to writing the strategies to the files we also put the information in a database, we also recorded what strategies were used against other strategies and how many times the strategies were used. When analyzing games that are played against our bot every strategy used are counted 10 times every time they occur. The strategies that have a lower count than 3 when all the replays are analyzed are deleted from the database to remove not used strategies.

When we analyzed the replay logs we also generated the different scripts that the BTHAI bot that we use needs. These scripts are simple recordings of what order the buildings was built and how many units of the different units the player had before an attack.

In the third phase we designed the Bayesian Network. For this task we used a program called GeNie, this program is a graphical interface to SMILE that allows us to design the Bayesian Network.

In the designing process we decided to design the Bayesian Network around what information is easy to obtain in any given game. The presence of the different units affect the likelihood of the different tech structure that is required to build it, that structure is again affecting the structures that are required to build it and so on. The different tech structures are affecting one of three different tech nodes, which again affect the strategy node.

The assimilator node is affecting the nexus node because it is a building that one can gather gas from and is only able to be put down on special vespene gas nodes that normally only are besides expansion sites. The nexus node is affecting the strategy node directly because it is a very central building that is focused around economy and not obtaining a more advanced army.

In simple terms the Bayesian Network is modeled around the tech tree in StarCraft Brood War plus the gameperiod node and the different strategy nodes.

In the fourth phase we were generating the training data for the Bayesian Network. This task required only to gather all the used strategies in a given matchup and write it to a file using the correct syntax. The first line of the file is the names of the nodes in the network separated by white spaces. The following lines are one strategy occurrence per line. The strategies are represented by the value of the all the nodes in the network separated by white spaces.

In the fifth phase we had to adapt the BTHAI bot to our use. First we had to load the Bayesian network in to the bot, collect the information about the enemy units and buildings, put that information into the Bayesian Network, run the new information on the network and collect the output information from the network and change the scripts if they needed to be changed.

This meant to change how the bot took in its scripts. As standard it requires one of each script at the start of the game, we had to change this so that we were able to change the script mid game. Changing scripts mid game we had to check what buildings and units the bot had made so we could adapt the scripts to that specific situation.

6.1.4 Training and Testing

When we tested our Bayesian Network we had the following goals:

- Find out if the Bayesian Network learned to predict the different strategies given incomplete information.
- Find out if there is some difference between how the different experiments adapted to the players.
- Find out if the Bayesian Network can exclude some strategies based on what information it got.
- Find out if the Bayesian Network can be used in a full game environment with success.
- Find out if the bot is adapting to what the player is doing in a single game.

We had two different setups, both starting with the same trained Bayesian Network.

- First setup will have the structure of play 5 games against one script, retrain and play 5 more game against another script. This is done until it has run against 4 different scripts.
- Second setup will have the structure of playing 5 games against the original built in AI, retrain and play 5 more games against the original built in AI. This is done 4 times.

6.1.5 Conclusion

After the two experiments was done we had 40 replays, 2 databases and 9 different trained Bayesian Networks. We focused on analyzing the different Bayesian Networks. Looking at how the Bayesian Network performed in the experiments, we can conclude with that it worked on a functional level, it received and delivered the information as intended. Further analysis shows how the different networks, given the same information, predicted different strategies. It was shown that when the Bayesian Network was exposed for a given strategy the probability for that strategy increased given that the information that was available fit that strategy. If the strategy did not fit the available information the probability was lowered so much that they can easily be excluded. The results show that the two different experiments did not have any notable different behavior other than that there was used other strategies in the experiments.

When analyzing how the bot itself worked we found that it did not work as well as intended. This was not the fault of the Bayesian Network but how we implemented it into the bot. This resulted in that the changing of scripts did not work as intended, the scripts only changed when the game period changed and not whenever there was a new strategy output from the Bayesian Network.

From the experiments we found the following findings:

- It was shown that the Bayesian Network adapted to the strategies that it was exposed to and was able to predict the strategies that was used by elevating their probabilities this was also true when given incomplete information.
- We did not see any difference between how the different experiments behaved.
- The Bayesian Network was able to exclude the strategies that did not fit the given information.
- The experiments show that it is possible to use a Bayesian Network for strategy prediction in a full game environment.
- The bot adapted to the player during the game but not as well as we hoped.

Based upon the findings we can state the following about the requirements for the software.

- The bot adapts to what the player is doing in a single game, not as well as we hoped but well enough for our purpose.
- The bot adapted to what the player is doing over multiple games by adapting its values in the Bayesian Network.
- The bot recognizes different strategies by putting the information available into the Bayesian Network and getting the most probable strategies as output.
- The bot can use different strategies by having analyzed replays of good players and recorded different build orders and army compositions.

- The bot remembers what the player has done before by analyzing the replays played against it and retraining the Bayesian Network based upon the result.

From this we can conclude with that our implementation of a Bayesian Network to predict the opponents strategy based on incomplete information were successful. Our implementation was also successful to adapt to the opponents preferences in strategy choices both inside a single game and over multiple games.

6.2 Reflection

6.2.1 The design/development process

The bot worked in a satisfying way as a proof of concept. The way we implemented the Bayesian Network into the bot could have been done better. The way we used the Bayesian Network within the bot was to use the strategy with the highest probability in the counterStrategy node. This is a very simplistic way of using the Information the Bayesian supplies. By changing how the bot used the information that the Bayesian Network gave as output we could increase the effectiveness of the bot by a significant margin. If this was done we could have gotten more long and a more realistic games, this would give better training data and would again improve the strategy prediction in the later parts of the games.

The Bayesian Network worked in a good way in our experiments but it was modeled as a very simplified version of the real problem and therefore there are relations that are not modeled in our model. Creating a more sophisticated model could give better strategy prediction.

6.2.2 The method for evaluation

In evaluation of the bot was only played against preprogrammed scripts, It would have been preferable to have the bot play against human players to get more realistic data. interviewing the players in retrospect would give us a better understanding on what the players thought about the bot.

6.3 Future Research

The strategy prediction have been a central focus point for this thesis, the strategy prediction we suggest is far from perfect and we would like to suggest the following as possible future research to further improve upon the adaptive strategy prediction in RTS games.

- Improve the way the strategies are modeled. The way we modeled the strategies is very specific. One could make a more general model to group strategies that is very similar together. This would make it more easy to find good counter strategies to a group of strategies. Also modeling what strategies transition well with each other would be positive.
- Improve the way time is modeled and how they transition between each other. The way we modeled time is a inaccurate, having more time classifications and focusing around key times in the game in addition to the attacks could improve the accuracy of the game time.
- Implementing the Bayesian Network strategy prediction with another bot that is not based upon scripts, like a goal-driven approach could improve the effectiveness of the bot by a significant margin.
- Improve the modeling of our Bayesian Network to make the prediction more precise. This could be done by having the resource collection rate modeled in to the Bayyesian Network. This would affect that the opponent could not build certain buildings/units in addition to what we had observed given the known resource restrictions.
- Having a qualitative study of what human players think and feel when playing against an adaptive bot over multiple games. Since games is all about the players having fun, it is very interesting to see if the players enjoy playing against an adaptive bot like ours or not.
- Testing strategy prediction based on Bayesian Network on human players. Our experiments are done against bots, these bots are more easy to predict than human players, so it is interesting to see if our results transfer to experiments done with human players.
- Testing other AI techniques for strategy prediction in RTS games. Trying other techniques to solve this problem gives us a better idea of what works best.

7 References

Aha, D, Molineaux, M & Ponsen, M. 2005. Learning to win: case-based plan selection in a real-time strategy game. *Case-Based Reasoning Research and Development,* , 5–20.

Blizzard. n.d*a*. Blizzard entertainment: starcraft. `http://eu.blizzard.com/en-gb/games/sc/`. [Online; accessed 01-June-2012].

Blizzard. n.d*b*. Blizzard entertainment. `http://eu.blizzard.com/en-gb/`. [Online; accessed 01-June-2012].

Bourg, D & Seemann, G. 2004. *AI for game developers*. O'Reilly Media.

BTHAI. 2010. Techstuff. `http://code.google.com/p/bthai/wiki/TechStuff`. [Online; accessed 19-April-2012].

BTHAI. 2012. bthai starcraft bot using bwapi. `http://code.google.com/p/bthai/wiki/Home`. [Online; accessed 16-April-2012].

Buro, M. 1997. Takeshi murakami vs. logistello. *International Computer-Chess Association Journal,* 20 (3), 189–193.

Buro, M. 2003. Real-time strategy gaines: a new ai research challenge. In *Proceedings of the 18th international joint conference on Artificial intelligence* pp. 1534–1535, Morgan Kaufmann Publishers Inc.

Buro, M & Furtak, T. 2004. Rts games and real-time ai research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)* vol. 6370,.

BWAPI. 2012. Bwapi an api for interacting with starcraft: broodwar (1.16.1). `http://code.google.com/p/bwapi`. [Online; accessed 16-April-2012].

bwchart. n.d*a*. Bwchart. `http://bwchart.teamliquid.net/us/bwlib.php`. [Online; accessed 18-April-2012].

bwchart. n.d*b*. Bwchart faq. `http://bwchart.teamliquid.net/us/bwfaq.php`. [Online; accessed 18-April-2012].

BWTA. n.d. Bwta a terrain analyzer for bwapi. `http://code.google.com/p/bwta/`. [Online; accessed 17-April-2012].

Chung, M, Buro, M & Schaeffer, J. 2005. *Monte Carlo planning in RTS games*. PhD thesis, University of Alberta.

Cui, X & Shi, H. 2011. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security,* 11 (1), 125–130.

Decision-Systems-Labaratory. n.d*a*. Welcome. `http://dsl.sis.pitt.edu/`. [Online; accessed 17-April-2012].

Decision-Systems-Labaratory. n.d*b*. About genie smile. `http://genie.sis.pitt.edu/about.html`. [Online; accessed 17-April-2012].

Galway, L, Charles, D & Black, M. 2008. Machine learning in digital games: a survey. *Artificial Intelligence Review,* 29 (2), 123–161.

Hot-bid. 2008. 9500+ tsl ladder replays. `http://www.teamliquid.net/forum/viewmessage.php?topic_id=72552`. [Online; accessed 16-April-2012].

Hsu, F. 2002. *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton Univ Pr.

Hwaci. n.d. About sqlite. `http://www.hwaci.com/sw/sqlite/about.html`. [Online; accessed 17-April-2012].

Liquipedia. 2008. Terran. `http://wiki.teamliquid.net/starcraft/Terran`. [Online; accessed 15-May-2012].

Liquipedia. 2009*a*. Mid game. `http://wiki.teamliquid.net/starcraft/Mid_game`. [Online; accessed 16-April-2012].

Liquipedia. 2009*b*. Protoss. `http://wiki.teamliquid.net/starcraft/Protoss`. [Online; accessed 15-May-2012].

Liquipedia. 2009*c*. Zerg. `http://wiki.teamliquid.net/starcraft/Zerg`. [Online; accessed 15-May-2012].

Liquipedia. 2009*d*. Early game. `http://wiki.teamliquid.net/starcraft/Early_game`. [Online; accessed 16-April-2012].

Liquipedia. 2009*e*. Worker. http://wiki.teamliquid.net/starcraft/Worker. [Online; accessed 16-April-2012].

Liquipedia. 2010. Starcraft beginners' reference. http://wiki.teamliquid.net/starcraft/Starcraft_Beginners'_Reference. [Online; accessed 16-April-2012].

Liquipedia. 2011*a*. Metagame. http://wiki.teamliquid.net/starcraft/Metagame. [Online; accessed 16-April-2012].

Liquipedia. 2011*b*. Resources. http://wiki.teamliquid.net/starcraft/Resources. [Online; accessed 16-April-2012].

Liquipedia. 2011*c*. Scouting. http://wiki.teamliquid.net/starcraft/Scouting. [Online; accessed 16-April-2012].

Liquipedia. 2012*a*. Chaoslauncher. http://wiki.teamliquid.net/starcraft/Chaoslauncher. [Online; accessed 16-April-2012].

Liquipedia. 2012*b*. Tier one tech. http://wiki.teamliquid.net/starcraft/Tier_One_Tech. [Online; accessed 16-April-2012].

Liquipedia. 2012*c*. Tier three tech. http://wiki.teamliquid.net/starcraft/Tier_Three_Tech. [Online; accessed 16-April-2012].

Metoyer, R, Stumpf, S, Neumann, C, Dodge, J, Cao, J & Schnabel, A. 2010. Explaining how to play real-time strategy games. *Knowledge-Based Systems,* 23 (4), 295–301.

Millington, I & Funge, J. 2009. *Artificial intelligence for games*. Morgan Kaufmann.

Mitchell, T. 1997. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill,* .

poiuy qwert. 2009. Bwai. http://www.broodwarai.com. [Online; accessed 18-April-2012].

Ponsen, M, Muñoz-Avila, H, Spronck, P & Aha, D. 2005. Automatically acquiring domain knowledge for adaptive game ai using evolutionary learning. In *Proceedings Of The National Conference On Artificial Intelligence* vol. 20, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Ponsen, M & Spronck, I. 2004. Improving adaptive game ai with evolutionary learning. In *University of Wolverhampton* Citeseer.

Russell, S & Norvig, P. 2003. Artificial intelligence: a modern approach. 2003. *Section15,* .

Schaeffer, J. 1997. *One jump ahead: challenging human supremacy in checkers*. Springer-Verlag New York Inc.

Schaeffer, J & Van den Herik, H. 2002. Games, computers, and artificial intelligence. *Artificial Intelligence,* 134 (1-2), 1–8.

Simon, H & Newell, A. 1958. Heuristic problem solving: the next advance in operations research. *Operations research,* 6 (1), 1–10.

Spronck, P, Ponsen, M, Sprinkhuizen-Kuyper, I & Postma, E. 2006. Adaptive game ai with dynamic scripting. *Machine Learning,* 63 (3), 217–248.

Stout, B. 1996. Smart moves: intelligent path-finding. *Game Developer Magazine,* , 28–35.

Tesauro, G. 2002. Programming backgammon using self-teaching neural nets. *Artificial Intelligence,* 134 (1), 181–199.

Weber, B & Mateas, M. 2009. A data mining approach to strategy prediction. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on* pp. 140–147, IEEE.

Weber, B, Mateas, M & Jhala, A. 2010. Applying goal-driven autonomy to starcraft. In *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*.