Poznan University of Technology
Faculty of Computing
Institute of Computing Science

On Design of an Effective AI Agent for StarCraft

Filip Cyrus Bober

Poznań, 2014

Supervisor: Wojciech Jaśkowski, Ph.D.

Abstract

This thesis focuses on the field of real-time strategy games artificial intelligence. We introduce this area of research by comparing it to the widely known traditional games like chess to show the unique challenges and problems related to this field. To prove that the field is relevant for research, we show the resemblance between the real-life problems and those encountered in the real-time strategy games.

Our research focuses on the game StarCraft by Blizzard Entertainment, a representative of the real-time strategy games and one of the best selling computer titles in history. We chose this game primarily because it has recently become a test-bed for testing AI methods and many researchers evaluate their solutions in this game's environment.

We implement a fully functional artificial intelligence for StarCraft. We base our solution on the UAlbertaBot, which we extend and improve to play Terrans. The two main strategies used by our solution are "Vultures and Tanks" and "Wraith Rush". To evaluate our work, we participate in two largest real-time strategy games artificial intelligence tournaments organized in 2014: "AIIDE StarCraft AI Competition" and "CIG StarCraft RTS AI Competition" where we place 11th out of 18 and 10th out of 13, respectively. We also test our program against the default StarCraft AI, achieving 90% win rate. Implemented proved to be effective. Nevertheless it has many weaknesses and there is much to improve.

Finally, we share the final thoughts about the subject in the summary. Our conclusion is that even though the there is more and more work in the field of real-time strategy games artificial intelligence, there is still much to be done before the human level of effectiveness will be achieved.

Acknowledgements

I wish to express my gratitude to my supervisor, Wojciech Jaśkowski. His constant support, inspiring guidance and involvement from the beginning till the very end of this research motivated me greatly. I am very grateful for the time he spent on constructive criticism, remarks and corrections to this research. His ideas and suggestions were invaluable to me. Without his help and guidance this research would have never be completed.

I would also like to thank Bartosz Walter and Wojciech Complak for the diploma seminar classes they conducted, which provided be with a solid foundation for my work.

At last but not least, I wish to thank the whole Poznan University of Technology Faculty of Computing staff, with special honors to Mirosław Ochodek, Jerzy Nawrocki and Maciej Zakrzewicz, who I will always remember as the exemplar tutors, always full of understanding, helpful and willing to share their vast knowledge with others.

Contents

1 Introduction

1.1 Scope of Research

This thesis concerns problems from the field of real-time strategy (RTS) games artificial intelligence (AI). Most games of this type focus on building a base, gathering resources, training units and finally, attacking and destroying the enemy's base. To achieve victory, players need to plan their strategy ahead, manage the economy, control units and groups of units, predict and counter enemy's strategy, use abilities, reseach technologies and many other things that depend on the particular title. RTS games are very popular among players worldwide and many of the most widely known and top selling titles, such as WarCraft, StarCraft, Command and Conquer, Supreme Commander, Age of Empires, and Settlers, belong to this genre.

Almost all RTS games have some sort of AI implemented. Artificial intelligence in a RTS game is a computer program that plays the game just as the human player would do, or hopefully better. The AI should follow the same rules and victory conditions as a human player. Sadly, many game developers allow AIs to cheat to compensate for their weaknesses. Because of that players usually treat matches with computer opponents as a training or warmup before games with other human players. The interest in multiplayer games from the players side led finally to appearance of a competetive scene of RTS games, where StarCraft [sta08] (and currently StarCraft II) is most renown title.

The competetive side of computer games began to be called esports, where professional players compete against each other to win fame and prizes, similar to real sports. StarCraft, developed by Blizzard Entertainment is most popular competetive RTS game worldwide. The game is considered very well balanced and has recently become a test-bed for AI methods.

In this paper we focus on development of an artificial intelligence (commonly called bot) for StarCraft game. We describe what has been already achieved in this field of research and what is yet to be done. We also describe our approach and evaluate it against the default StarCraft AI, as well as the other existing bots.

1.2 Motivation

The area of classical games AI has been researched since mid 50's XX century. The problems concerning classical games AI are well described in a countless papers and the combined effort of researchers provided efficient solutions. In the most popular strategy board games such as chess or Go computer programs either pose a significant challenge for the human players or surpass them greatly. The best AIs are unbeatable to amateur players and even the professional players and world champions must do their best to win, and even then they sometimes lose.

This situation is just the opposite of what we observe in the RTS games area. The best bots struggle to beat an amateur players and are no match for even the worst professional ones. The games are predictable and not fun for the players, because there is no challenge in beating such a poor AI. To increase the difficulty of the AI, it is a common practice, seen in many RTS games, to allow the computer player cheating. Once again, from the human player perspective it is not fun, because losing to a cheating opponent does not sound fair.

Unfortunately, there is no satisfactory solution for the problem yet. The field itself is becoming more and more popular. In 2013 AIIDE competition there were eight participants while in this year's tournament, AIIDE 2014, there were eighteen [aii14]. Nevertheless, the advances are are slow as there has been no bot that could beat at least a moderate StarCraft player developed yet. Most of the algorithms that work for the traditional games cannot be applied to RTS games or are just computationally inefficient.

The field itself is interesting from the academic point of view, because there are many subproblems that are related to the real life problems. Solving those problems in RTS games will provide solutions to the real problems as well. Managing production in RTS games strongly resembles problems of the industrial scale production in real world. Algorithms invented for the purpose of units movement in RTS games might be later extended to the traffic control in real cities. Advances in RTS map terrain analysis could also be easily applied to real terrain, potentially improving the development of road networks or infrastructure planning. Adaptation to the changes in environment and decision making under uncertainty so common in RTS games is also inherent attribute of the real world. Advances in this area will improve stock market prediction [PG03], management efficiency [MTTM09], wildlife preservation [Hol13], risk management [Pow08], disease and epidemic control, health diagnosis [PKJHA], just to name a few.

Even though RTS games are complex, it is much easier to address these problems and evaluate the results in a controlled environment rather than in real life.

1.3 Goals

Our main goal is to provide a fully functional bot for the StarCraft game. We want our bot to be able to play the game through all the stages from the beginning till the end. The bot should be good enough to compete against other bots in one of the StarCraft AI tournaments that have been organized annually since 2010 [fir09].

Apart from the our main goal there are also several secondary objectives that we wanted to achieve:

1. Evaluate the existing bots and choose the best one for the base of our solution.

2. Make the bot to be an interesting opponent, fun to play against for the human players or at least make the bot play differently than the default StarCraft AIs.
3. Defeat the default StarCraft AIs.
4. Participate in the AIIDE and CIG StarCraft tournaments.

2
Real-Time Strategy Games

2.1 Problem Description

Real-Time Strategy games (RTS games) derived from turn based strategy games while the latter derived from board strategy games such as Chess or Go and tabletop wargames. Although there were some RTS games before, the Dune II is considered the one that popularized and defined the whole genre. Because of the several differences between mentioned genres, which we describe in this chapter, there are a number of problems associated with RTS games that are not encountered in the other two game types. Those problems require entirely different approach and prove many common algorithms used in traditional games useless. By traditional games we mean competetive, turn-based boardgames, invented mostly before the computers existed. The most popular traditional games are chess, Go, Draughts or Reversi.

Below we enumerate and describe characteristics shared by most RTS games, as was shown by David Churchill and others [SO13]. Each one of them has a huge impact on the algorithms used by the AI.

1. **Simultaneous:** Players issue actions at the same time. There is no such thing like a particular player's turn. Each player can issue any number of actions independently of the others or no actions at all. What is more, unlike the traditional games, some actions are not executed immediately but require some time to complete.
2. **Real-time:** The trait shared among all RTS games, defining the whole genre. There are no turns in RTS games. At least there are no turns in a way there are in traditional games . The game runs in a loop which processes player input but does not wait for it. There is always some minimal quant of time between the two game updates, though. In StarCraft this time is equal to 42 ms and it is small enough, compared to the turn based games, that the game can be considered to be played in real-time. The simulation in newer games is considerably faster and ofter reaches a dozen of thousands updates per second in modern games like Dota 2 or Counter-Strike.
3. **Partially observable:** The traditional games are mostly fully observable. Each players has full view on the state of the game. There are no secret moves, hidden from the other player. The RTS games are just the opposite. Every unit has some

line of sight. Everything beyond the our units' line of sight is hidden from us. The hidden area is called the *fog of war*. For this reason the algorithms can only estimate the state of the game. Exploration of the map becomes very important, for it provides information about the enemy moves. Moreover, in many RTS games (including StarCraft) there are invisible units, so even if we have the line of sight over the whole map, the game state remains unknown.

4. **Non-deterministic:** Some actions involve a random factor. For instance it is common for the units to deal damage within some random range or have a chance to miss.
5. **Huge state space:** While the chess is estimated to have around 10^{50} states and Go to have around 10^{170} states, the complexity of a standard StarCraft game is far greater. For instance, considering only the possible location of each unit in a game at any given time on a 128×128 map there are $16384^{400} \approx 10^{1685}$ states assuming there are just two players with two hundred units each [SO13].

With all the mentioned problems a RTS game AI has to face, there is a need for an entirely different approach than those that work well for the traditional games. Currently a common practice is to divide the problem into multiple subproblems and solve them separately.

Figure 2.1: Screenshot from Dune II [dun14]

2.2 StarCraft

StarCraft is one of the most popular RTS games in history. Released in 1998 by Blizzard Entertainment [sta08] it was sold in an immense number of more than 9.5 million copies. The popularity of the game was hugely affected by its well known competitiveness. Thousands of professional players competed in hundreds of tournaments over the years. In some countries (South Korea, for instance) they became celebrities [Dot13] and their salaries are comparable to those of professional sportsmen [Ear14]. StarCraft is one of the few games that started the constantly growing popularity of e-sports (electronic sports). Its successor, StarCraft 2 is currently the most popular RTS game with thousands of professional and amateur players worldwide.

Like most RTS games of its epoch, StarCraft is a two-dimensional game with isometric camera view. There are three different playable races: Protoss, Zerg and Terrans. Protoss are a race of highly advanced and intelligent humanoid aliens. Zerg are a swarm of aliens that pursue the genetic perfection. And finally, the Terrans are humans in the near future. StarCraft is famous for how well the races are balanced. Each race has its own strengths and weaknesses and the strategy for every race is different.

Protoss units are very expensive but strong. They are usually outnumbered by the enemy but Protoss units are superior. Like other races, Protoss units have hit points, which measure how durable a particular unit is but each unit also have shields. The shields regenerate over time and must be reduced to zero before hit points are subtracted. Protoss workers construct buildings by just placing them on the map, and thus the worker is needed just to initiate the construction. The Protoss buildings, however, can only be build around special buildings called Pylons.

Zerg units, on the other hand, are relatively cheap, but weak. They will most probably outnumber the opponent's forces. With a good economy a Zerg player can attack its opponent with huge wave after wave of units. Instead of constructing buildings, the Zerg workers morph, which means that the unit is sacrificed to construct the building.

Somewhere in the middle of those two extremes are Terrans. Their units are quite cheap and dependable but more expensive than Zerg and weaker than Protoss. They construct buildings by assigning a worker for the time of the construction, which is also sort of a compromise between 'placing' buildings by Protoss workers and the Zerg 'morphing'.

The game goal is to destroy the opponent's base. To achieve this each player must build his own base, gather resources, train unit and send the force to attack and defend his own base from enemy aggression. There are a number of possible commands that each unit can be issued such as attack, hold position, move or attack move. Some unit have also special abilities like burrowing to the ground, cloaking to become invisible or marking location for a nuclear strike. An important part of winning the game is issuing the right commands at the right time. Professional players usually have more than 200 APM (Actions Per Minute), what means that they issue more than three commands per second.

Because of its popularity, well balanced races, complexity and competitiveness the StarCraft is considered a test ground for the RTS games AI research. Accomplished advances in the field are verified in tournaments where bots (computer players, controlled by the AI) play against other bots or professional players.

Figure 2.2: Screenshot from StarCraft

2.3 Challenges in RTS Games AI

Human players are still far superior to any bot in RTS games [BC12]. Even the amateur players beat the best AIs, not to mention the professionals. A popular StarCraft ladder called iCCup ranks the players from the lowest rank E to the highest ranks A^+ and *Olympic*. Amateur players are usually ranked between C^+ and B, while professionals are graded from A^- up. For comparison, the best bots achieved ranks D and D^+. This shows that there is much to improve in this field of research and we are still far away from developing an AI that can compete with even the amateur players.

There are many aspects of the game that must be considered for the bot to be successful. Below we describe the most important ones, mentioned by David Churchill in his survey [SO13].

2.3.1 Planning

Due to a huge number of possible actions in each quant of time of an RTS game, the standard approaches, common for traditional games are usually not applicable to the RTS games. There are many aspects of the game where each one may require a separate, different approach. The common technique is to divide the game into a number of abstraction and then solve the problems each of them poses separately. The alfa-beta search may prove efficient in managing build orders but might struggle with managing unit movement.

2.3.2 Learning

There is a huge number of data that can be used for learning from in StarCraft, as well as many other RTS games. Replays are particularly useful. Build orders, ability usage, building locations, units positioning, it can call be extracted from the replays and used for learning purposes. The problem lies in how to extract that data and how to make a good use of the received information.

2.3.3 Uncertainty

Choosing the right actions while having a partial information about the game state is another challenge for the AI. The bot need not only to scout to explore the map and get the information about the opponent but must also predict what the enemy is likely to do. Getting the information about the opponent and acting accordingly is crucial for winning the game.

2.3.4 Spatial and Temporal Reasoning

Spatial reasoning refers to the proper units and buildings positioning. The base should be easy do defend while maximizing the efficiency of resource gathering rate. Units are much more efficient when they attack from the higher ground, so the bot should provoke situations when the enemy is on the lower ground. Proper positioning also includes moving units to positions where as many of them as possible can shot at the enemy while minimizing the return fire. Much of the spatial reasoning tasks involve terrain analyzing.

Temporal reasoning is connected to timing the attacks, choosing the right strategy and changing it if necessary, retreating when the battle would be lost and deciding whether is it better to chase the enemy or wait for the reinforcements. Decisions associated with upgrading units, researching technologies, expanding the base to improve the economy are all within the temporal reasoning aspect of the game.

2.3.5 Abstractions

When professional players talk about their strategies, they usually use two abstractions: macro and micro. The first one refers to the overall strategy, while the latter to the controlling individual units. We decomposed the gameplay further, though, because the division was to general for the purpose of the AI programming. Figure 2.3 we show levels of abstraction: strategy, tactics and reactive control along with estimate reaction time. For example it takes approximately three minutes to adjust the strategy, as high level decisions take time to take effect, but it takes just about a second to adjust the reactive control. What is more, the higher level of abstraction, the less knowledge about the state of the game is known. Because of that, higher abstraction decisions rely mostly on estimations and predictions, while lower level abstractions have the privilege to use more reliable data.

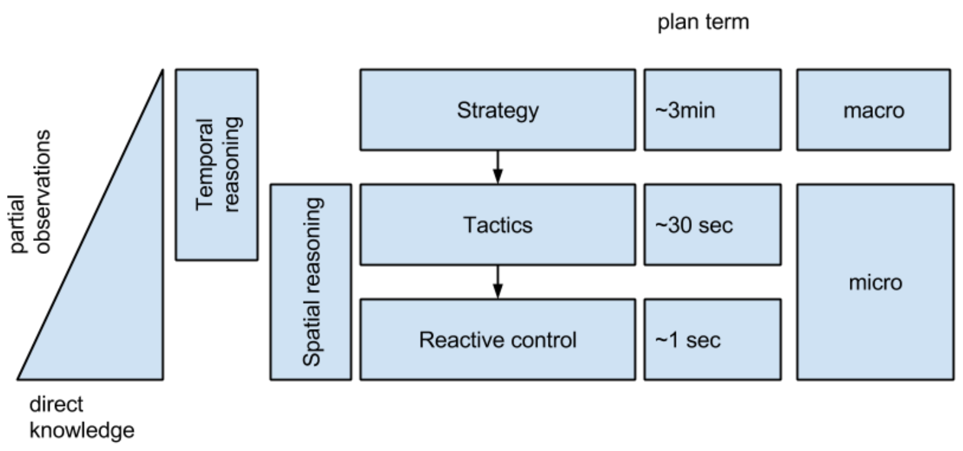


Figure 2.3: Levels of abstraction [SO13]

2.3.5.1 Strategy

This is the highest level of abstraction, a player would call it macro. Strategy answers the question: how are we going to win the game? Within the borders of strategy we decide whether we want to rush the opponent (attack as soon as possible with small force), focus on expanding (building more bases and strengthening the economy), attack with air units and so on. The strategy defines indirectly all of our future actions.

2.3.5.2 Tactics

Tactics is the implementation of the strategy. Players would still call it macro. Tactics consists of training units, building buildings, deciding whether to attack, but it does not operate on the individual units.

2.3.6 Reactive Control

Reactive control is called by StarCraft player 'micro'. It is where we issue orders to individual units, where we use abilities, hit and run techniques and so on. This is the implementation of tactics.

2.3.6.1 Terrain Analysis

Terrain analysis is neither micro or macro, but it affects both. Everything from choosing the strategy to using the ability is influenced by the map. Rush strategies are not effective on large maps, as it takes more time for the units to reach their destination. Small maps force players to play more aggressively. Narrow valleys and passages require little resources to be fortified well, while open terrain is very hard to defend.

Professional players learn the tournament maps, as the one who uses terrain to his advantage has much higher chances to win that the one that does not.

2.3.6.2 Intelligence Gathering

In RTS games both players are encouraged to constantly spy on their opponents. There is no best strategy in StarCraft, or at least nobody has found one so far. The most efficient strategy is the one that counters the one used by the opponent. To achieve that, there must be enough information provided, to predict the enemy actions. If there is no information, we can only guess what actions are right and we are most likely to be countered by our opponent.

2.3.7 Collaboration Between Multiple AIs

There has been very little research in the field of AI collaboration in RTS games. Maybe the reason is that the tournaments for the professional players are played almost exclusively one versus one and the bot tournaments try to mirror the real ones. Another reason might be the fact that there is still no AI good enough to beat even an average professional player, and the collaboration is not a priority right now. Nevertheless, sooner or later this will become a problem to be dealt with.

2.4 State of the art

Until recently the field of RTS games AI has been quite unpopular among the AI researchers. Since Michael Buro's call for research in this area [Bur03], it has been getting more and more attention. As the topic is wide, the particular papers address generally some specific problems like terrain analysis or unit movement and rarely provide a complete solution in form of a working AI.

There are number of papers concerning different aspects of the field. Gabriel Synnaeve proposed computing the complexity of the game by looking at the branching factor and the depth of the game instead of estimating the state space [Syn12]. Comparing the state space might shed some light on the difference between the complexity of RTS games and traditional games but has little use in a comparison of two RTS games, because of a huge numbers and corresponding estimation errors.

M. Mateas and B. G. Weber in their paper [MW09] presented how to use data mining approaches to predict the opponent's strategy and acquire the domain knowledge. E. Dereszynski and others used Hidden Markov Models (HMM) to address the problem of learning the build orders.

R. Houlette and D. Fu showed how to apply a finite state machines (FSM) to the RTS games to facilitate the implementation of hard-coded strategies [HF03].

There were also attempts to apply case-based reasoning (CBR) [AP94] to the RTS games AI such as D.W. Aha and others to plan selection in a Wargus game (Blizzard's Warcraft II clone) [DWAP05], J. Hsieh and C. Sun work that explained how to analyze replays to extract build orders [HS08] , F. Schadd and others paper about modeling the opponent based on his behavior applied to the Spring game [FSS07] or U. Jaidaee and others CBR approach to learning goals in RTS games [UJA11].

A significant amount of work has been done around the Bayesian models applied to the AI in RTS games. G. Synnaeve and P. Bessiere presented a Bayesian model to predict opponents' opening strategy [SB11a]. Even though, the model was applied to StarCraft it is viable for any resource based RTS games. Parameters for the model were learned from game replays. Another work by the same authors concerns Bayesian model for units control [SB11b], that is able to deal with the uncertainty of the game environment. This was also done in StarCraft environment.

Julian Togelius and others used evolutionary algorithm to generate StarCraft maps [JTY10] in a process called procedural content generation (PCG). They used several fitness functions to produce a tradeoff result. The work is interesting not only from the perspective of

RTS games but the whole game industry and maybe also other domains, where virtually created reality takes a part.

There is also an interesting summary of the progress in AI research provided by D. Churchill and others in their survey [SO13]. Over the recent years there was also a significant improvement in StarCraft bots performance, thanks to the tournaments organized annually at the University of Alberta and the CIG conference. The field of RTS games AI is becoming more and more popular with increasing number of related work every year. The progress is still slow, however, and there is still much to be done before RTS AIs could compete with human players.

3 Bot Architecture

3.1 Objectives

In this chapter we describe bot architecture and explain why our soulution is based on an existing code, rather than building from scratch. As the chosen base for our solution is UAlbertaBot, this chapter is mainly about its architecture, modules and design.

3.2 Motivation for Basing on Existing Bot

There were two alternatives we had been considering. First one was to build our bot from the scratch, second was to improve one of the existing bots. Each choice had its adventages and disadventages.

Developing our own solution would give us full control of the bot's code. The downside would be that existing bots have already well written modules with functionalities we would have to develop ourselves anyway. Much time would have been spent on something that has been done already.

The other solution provided us with functional bot to build upon. With core modules working and ready to be used and improved. The problem was that the the code would not have been written by ourselves and before it could be improved, it must had been understood first.

After evaluating both choices' pros and cons we finally decided to improve an existing bot. We assumed that we would have spend too much time on developing basic things like building placement or unit movement. Instead of pushing the field of RTS games AI a bit forward, we would have invented the wheel again.

As the decision to improve an existing bot was made we faced the problem of choosing the right bot. There were several bots to be chosen from. What we needed was a bot with good, modular architecture that could be easily extended. Most of the StarCraft AI tournaments bots fulfilled our criteria.

There were a few factors that we had to take into consideration. We wanted the bot to be one of the best available. It limited our choices to the bots that succeeded in tournaments, which were but a few. We wanted the bot to have elegant architecture, which most of the tornament bot had. And finally, we were looking for a bot with clear, well-written and documented code. The last factor showed us the winner: UAlbertaBot.

3.3 UAlbertaBot Background

UAlberta bot was developed by David Churchill at University of Alberta [D.14d]. The bot has been proven to perform well as shown by its tournament standings. It achieved second place at the AIIDE 2011 Competition, third place at AIIDE 2012 Competition [D.14a] and second place at CIG 2012 Competition [SO13].

The bot plays the Protoss race. Its main strategy is to rush the opponent with Zealots (Protoss melee unit), which means the bot tries to attack an opponent as soon as possible. If the enemy has not built early defense (which happens quite often) the game finishes quickly. After the initial rush UAlbertaBot tries to overwhelm the opponent with more Zealots and Dragoons (Protoss ranged unit).

3.4 Architecture Design

Due to a complex nature of RTS games it would difficult to write a bot that consists of only one module controlling everything. Even though there were some attempts to apply a hollistic approaches to RTS games [HO13] none of them proved to be successful in the StarCraft AI. It is very common to divide game management into smaller modules where each module manages a certain task. Those modules are usually more or less indepentent which makes them easier to control and maintain. There are many adventages in this approach. The code is cleaner, easier to understand and to modify. Modules can be tested indepentently and the chances to find and correct bugs are higher. Another adventage of modular approach to the problem is that different AI techniques may be easily implemented in different parts of the code.

UAlbertaBot consists of twelve main and indepentent modules. Only a few of them send commands directly to StarCraft. Some modules communicate with each other, some manage other modules and some share the same data. Many bots have some main control module that oversees other modules and UAlberta bot is one of them. There is a Game Commander that invokes and manages all the other modules.

We present the most important UAlbertaBot modules in the following sections.

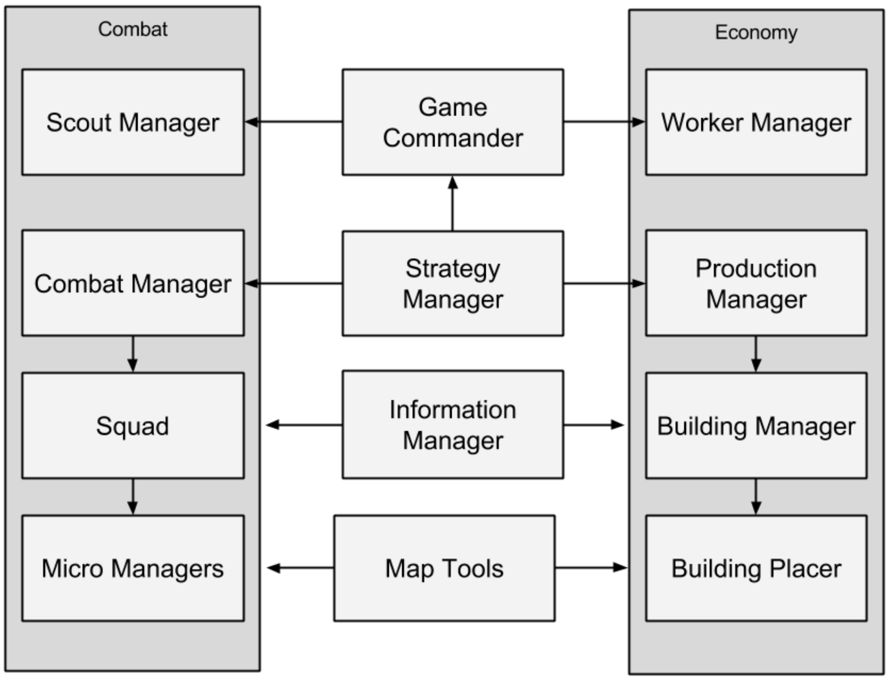


Figure 3.1: UAlbertaBot Schema [SO13]

3.4.1 Game Commander

Game Commander is the main module. Its purpose is to invoke update methods of all the other modules and to measure time within they are executed. It is a simple yet important task. If a certain module is to be turned on or off, it should be done from the Game Commander.

3.4.2 Information Managerr

As the name suggest, Information Manager provides other modules with number of relevant data about the game. There is information about enemy units, the regions that the enemy occupies. The data is stored about different kind of threats like air threat, cloaked units threat, enemy detectors and so on. The module does not send any commands to StarCraft.

3.4.2.1 Update

Information Manager is updated the same way as other managers - from the Game Commander module, every frame. There are units (self and enemy), buildings and start locations updated. If an enemy base has been seen it is updated immediately. The same action applies to enemy units and buildings. Due to the existance of the fog of war, very common in RTS games, the Information Manager has always more or less outdated information. Even outdated information may prove invaluable though. Base locations do not change often. Built units do not disappear without being destroyed and buildings (excluding flying Terran bases) do not change their location. Therefore the information is partial but not false.

3.4.3 Map Tools

The module provides tools for analyzing the game map. Similarly to the Information Manager Map Tools module does not directly interact with StarCraft but is used widely by the other modules wherever map information is needed. Building Manager gets the next expansion location from the Map Tools module. Units get data about distance (straingt line or ground) to the enemy bases.

3.4.4 Strategy Manager

There are five tasks that Strategy Manager is responsible for. First task is to define opening build orders and choose the right one. Sesond task is to execute build order search as soon as soon as the current build order queue becomes empty. Third task is to inform Combat Commander whether it is time to attack or not. Next task is to choose the right time to expand (build a new base to improve economy).The last task is an optional reading and writing data containing feedback about used strategy efficiency against a particular opponent to the file. This functionality may be very useful in tournaments, where many games are performed between same bots.

3.4.4.1 Defining the Opening Strategy

At the beginning of the game the bot executes the opening strategy or opening build order. It is an order according to which units are built. Build order is a sequence of actions. Each race has its own set of available actions.

Opening strategy defines a build order that is executed one by one (or simultaneously if building and resource requirements are met). Opening build orders have huge impact on a game. The difference between the good and the bad one may be a difference between winning all the time or loosing all the time. A proper build order may not win the game alone but wrong one will undoubtedly loose it.

As StarCraft is a competetive, mature, multiplayer game with millions of players worldwide, the expert knowledge about the subject is enormous. Countless build orders have been tested against countless opponents and strategies. It has been proved by trial and erros of the players that there are not as many efficient build orders as one might have thought. There are but a few dozens effective opening build orders for each race.

Chosen opening build orders are hard-coded in Strategy Manager. UAlbertaBot by default has three different openings implemented: Protoss Zealot Rush, Protoss Dark Templar and Protoss Dragoons.

3.4.4.2 Build Order Search

UAlbertaBot implements its own algorithm for build order search. Because details about its implementation go beyond the scope of Strategy Manager it is described in a separate section.

Within the scope of Strategy Manager build orders are constructed from goals. Each action (producing unit, researching technology or upgrading) may be set as a goal in Strategy Manager. Multiple goals can be set at a same time. It is build order search job to return a proper build order from a given goals. For instance if we wanted Terran Vultures and we set the goal to build them computed build order should contain all of the required prequisites in a correct order (in this case Barracks, Factory and possibly Supply Depots) along with Vulture itself. In short, returned build order is a set of actions from the current state of the game to the state described by the goal. Such solution enables us to define a strategy at a very high level without the need to assign each action one by one.

3.4.4.3 Attack Permission

It is Strategy Manager's responsibility to choose the right time to attack. Rushing strategies tend to build units fast and attack as soon as they are ready. Tech strategies prefer to attack only after signifficant force has been trained.

All the constraints are held in Stragegy Manager doAttack method which is later called by Combat Commander to check if a specific unit or units should attack or not. By default the only constraint is the number of units available.

3.4.4.4 Order to Expand

Time at which base expansions are build plays an important role in StarCraft. Expanding too fast results in too weak defense to fight off the enemy offense. Building too late results in too weak economy to sustain constant unit production and base development. There is no one proper timing. It all depends on the strategy chosen, enemy's strategy, map and many other factors.

3.4.4.5 Learning

In the tournament environment each game setup (map, opponents etc.) is played multiple times. Usually hundreds or thousands of games are played. Bots are allowed to learn the opponent and they may store data about how well each strategy performed.

UAlbertaBot uses UCB (Upper Confidence Bounds) algorithm [EKG12] to determine which strategy to use. The algorithm task is to choose the empirically best (read from file) action possible while still exploring the environment for a better strategies [PAF02].

After each game the file with results is updated (or created if there was none). The more games played the better the best strategy estimation.

It worth to mention that there are no new strategies or build orders produced from collected data. The pool of available strategies is always the same. After the bot was comiled no new strategy may be used.

3.4.5 Build Order Search

After the opening build order has been finished (or interrupted) the early game is over and searching for a new build order begins. The build order search algorithm is explained exhaustively in the paper by David Churchill and Michael Buro [CB11].

Searching for the optimal build order classifies to the field of automated planning, which is a central problem in artificial research with many real-world applications such as solving a Rubik's cube or building a submarine [CB11]. Improving efficiency in this area may save millions of dollars and man-hours due to its indurstrial value.

Searching for the optimal build order is a algorithmically hard planning task. It can be defined as finding the optimal way from the start state to the goal. The goal may be composed of multiple sub-goals (and usually is, as the main goal is without a doubt to destroy the opponent). Let assume that the goal is to build a Terran Wraith. It requires Starport to be built before. Starport requires Factory and Factory requires Barracks. Of course all units require minerals, gas and, excluding buildings, supply as well. What is more it takes time to complete a unit. Minerals and gas are gathered by workers. The more workers the faster resources are gathered. But building workers also cost minerals, supply and time. It is also uncommon to have just one goal at a time. Most often there are several different goals. We end up with quite a complex task.

There are many factors that must be considered when searching for a new build order. We can try tu rush the opponent but if it fails our economy will suffer. We can try to tech (which means invest early in the economy, upgrades and research) but at the cost of army and defense. What is more the same goals can be sometimes achieved in different ways. Because there are no turns in RTS games the time within a new build order is found is crucial. An ideal algorithm should provide the optimal build order within one game frame, which is approximately 24ms. Every delay gives our opponent an adventage. It is better to have not optimal build order in time than the optimal one late.

UAlbertaBot treats a build order as a set of actions. Every action (excluding combat actions) requires some resources and provides some resources. There are four types of resources: Require, Borrow, Consume and Produce [BL10]. Required resources must be available when an action is issued. Borrowed resources are consumed when an action is issued but then returned as soon an action is finished. Consumed resources are like borrowed resources but are never returned. Bots units and buildings are referred as units and both units and consumables (minerals and gas) are considered resources for the purpose of the search.

Each action is represented by tuple $a = (\delta, r, b, c, p)$. Duration δ is measured in game frames count while r, b, c and p mean action preconditions sets: required, borrowed, consumed and produced item p respectively. In a notation provided Terran Marine would be described as $a = ("ProduceTerranMarine")$, $\delta = 360$, $r = \{\}$, $b = \{TerranBarracks\}$, $c = \{50\text{minerals}, 0\text{gas}, 2\text{supply}\}$, $p = \{1TerranMarine\}$. Borrowed item set is empty because Marines do not have any other prequisites than Barracks, which is their production building (borrowed).

To effectively track progress to achieve a particular goal UAlbertaBot uses states that take the form $S = (t, R, P, I)$, where t is the current time in game frame count, R is a vector of each possessed resources along with information if they are currenlty borrowed and till what time, P is a vector that stores actions in progress and I is a vector that holds income data.

Income data stored in vector I needs some explanation. As mentioned before, minimalizing time withing build order calculations are finished is crucial in RTS games. Many other simulations are dominated by actions connected to resource gathering [CB11]. . To deal with the problem, resource gathering rates are estimated from emirical data and are equal to 0.045 minerals and 0.07 gas per worker per frame. In some situations it might not be accurate; for instance when distance between resources and base is long. In most cases the abstration is accurate enough and its impact on algorithm complexity is huge.

A common case where esitmation would be truly inaccurate is when a worker is assigned to construct a building. To compensate time lost by worker movement, 96 frames (four seconds) are added to the game state's time component. Again, because map data is not taken into account added time is not truly accurate but simplifies the computation much.

To perform build order search a depth-first branch algorithm is used. For a given starting state S, the depth-first recursive search is performed on on the descendants of S in order to find a state which satisfies a given goal G [CB11]. Searching can be halted at any point to return the best solution found so far. It is an important feature in the RTS games environment where the time is limited. Another adventage of the algorithm is that it uses linear amount of memory with respect to the maximum search depth.

Before children of the current state are generated, posiible actions must be checked for their legality. An action is legal if without executing any other actions, the simulation of the game will finally lead to the state where all of the required resources are available.

Given the abstractions the action is legal if all the required prequisites and resources are either available or will be available within some time, without issuing any other actions. For example if we are currently constructing Terran Barracks and we have less minerals that is required to train a Marine, the action is legal. Is is legal because Barracks will be build without issuing any other actions and the minerals required for the Marine will be collected without issuing any other actions. In the given example an action to train a Terran Siege Tank would be illegal because game simulation alone would not lead to Tanks being available. There must be an action to build a Factory issued first, which in a given example is legal.

Unlike many traditional board games, RTS games allow the user to take no action at all. Even though no action is taken, the game state changes. Minerals are aquired, buildings are constructed and units are trained. It results in a search depth that is linear in the makespan of our solution instead of the number of actions taken. To eliminate the need for null actions a fast-forwarding simulation technique was impelemted.

In StarCraft it is usually not recommended to save resources as resources do not win the game. Units do. Although there are some cases where saving resources is recommended. Typically in the late game when we want to save large amounts of minerals and gas to build fast many powerful units as soon as they become available. For the sake of UAlbertaBot build order search saving resources is not considered. An optimal build order is defined by one where an action is issued as soon as possible.

There are three state transitions used by UAlbertaBot search algorithm. Transition $S' \leftarrow Sim(S, \delta)$ simulates the game state progression in time δ which means gathering resources, training units, researching technologies and so on. Transition $\delta \leftarrow When(S, R)$ returns the earliest time when the requirements R will be available. Transition $S' \leftarrow Do(S, a)$ executes an action a in state S assuming that the resources required for the action are available. Issuing of the action means that required resources are consumed or borrowed. Resulting state S' is the state just after the action was issued.

A feature that is inseparable from RTS games is action concurrency. Whether two concurrent actions when sequentialized result in the same state or not might be co-NP hard to decide [BK07]. In StarCraft and many other RTS games multiple actions issued simultaneously are independent of each other. Due to this property, there is no need to iterate over all possible sequences of concurrent actions.

To improve quality of computed build orders, UAlbertaBot uses macro actions which are considered during build order search. Macro actions are specific rules observed in professional players build orders such as building units in bundles.

After build order has been computed it becomes available to the Production Manager which then executes the build order.

3.4.6 Production Manager

Production Manager strongly relies on the data provided by the Strategy Manager. Wherever new build order is needed Production Manager calls the Strategy Manager to get required data.

Responsibility of the Production Manager is to maintain constant production. Events such as deadlocks, destroyed workers, appearance of cloaked units are all handled from here. Those events usually invoke new build order search.

Production Manager does not send commands directly to StarCraft. It is kind of a buffer between higher Strategy Manager and Building Manager below (which does send commands diretly to StarCraft).

Central Production Manager's property is the queue. It stores the current build order actions in the correct order. Each action has its priority. The higher the priority the sooner it will be built.

The three most significant Prodution Manager's methods are presented below.

3.4.6.1 Update

Update method is called every frame directly by the Game Commander.

At the beginning there is a call to build order management method. Next the update method checks if the queue is empty. If the queue is empty, searching for a new build order begins. Otherwise there is a check if a dedlock occured. At the end there are some special cases considered like inserting Photon Cannons into the queue if cloaked units were spotted. This is achieved by setting a new item as a queue highest priority item. In most cases, however, it is not recommended to use such insertions because it is done outside the scope of build order search. Thus the inserted items may require additional resources that are not yet available or actions that have not beed issued yet. Nevertheless it is the fastest way to change the build order without wasting time on a new search.

3.4.6.2 Perform Build Order Search

The update function executes searching for a new build order. Next, the goal is retrieved from the Strategy Manager and passed forward to the class responsible for performing search algorithm. Finally, returned build order actions are then saved to the queue after being converted to the proper format.

3.4.6.3 Manage Build Order Queue

Build orders are filtered and adjusted within this method. At the beginning there is a check if queue is not empty. In the case it is empty, the method returns, as it has nothing to do until a new build order is fould.

While there is something in the queue, each item is checked if there is a producer for it and if there are enough resources. If either is false the item is skipped in this frame. Under some circumstances units may be filtered (removed from the queue). For example if there are too many refineries already and another one is not needed.

After queue item passed through all the filters and constraints, it is passed to a proper manager (or trained in the case the item is a unit) and removed from the queue. Skipped items remain to the next iteration.

3.4.7 Building Manager

In the chain where Strategy Manager defines goals, Production Manager manages build order queue the responsibility to actually construct the buildings belongs to the Building Manager. It is a class that sends commands directly to StarCraft through BWAPI.

It is for higher management modules to decide what buildings are to be built. How the base will actually look, however, is all to in the hands of the Building Manager. In most RTS games and especially in StarCraft building placement has huge impact one the game. A tower in right place may defend the workers agains early rush with no casualties on our side. The same tower in a different location may do nothing to stop the enemy from destroying all the workers and buildings but a tower.

There are number of strategies used by professional StarCraft players to effectively position the buildings. For example it is common for Terran players to build the wall. It is a blockade built on a ramp (narrow, steep pass from lower to upper ground) which stops enemy units from entering the base. It is extremely effective against early Protoss

Zealot or Zerg Zerglong rushes as the Terran Marines can shoot from behind the wall while stayin out of the enemy range.

Information about buildings under construction is stored in a special structure called buildingData of ConstructionData class. The structure holds data about the construciton state (unassigned, assigned and under construction), how much resources reserved for building do we have, how much space do we want between the buildings, which units are currently assigned for construction and so on.

3.4.7.1 Update

During Building Manager update, a sequence of six steps is performed.

the first step consists in check if any worker died while moving to the construction site. If the worker has died the current building is cancelled and removed from the data structure. The check does not count Terran workers that died during the construction.

Next step involves assigning workers to the buildings. A worker is assigned for each building marked with the unassigned state. After that the building potential location is validated. If the location is invalid the current building is skipped. Otherwise a worker is taken from the Worker Manager. Then the location is checked once more. If the check ends with success the space for the building is reserved and the building state changes to assigned.

The goal of the third step is to execute build command to the assigned workers. If building location has not been explored yet, the worker is ordered to move to that location. If the location is not reachable, workers are returned to the Worker Manager. If there are no mentioned obstacles, the worker is given BWAPI command to start the construction in a given location.

The fourth step purpose is to update building data structure. Final position is updated. Reserved resources are returned as they should be spent already. Protoss workers are returned back to the Worker Manager. And the builing state changes from assigned to under construction.

The fifth step is to assign new workers if any died during the construction. As UAlbertaBot plays Protoss by default, this step is not implemented and there is no such functionality. If a Terran worker is destroyed during the construction the building is abandoned and never finished.

The final step checks if any building in under construction state is finished. Every completed building is removed from the Building Manager. This step concludes constructin a building.

3.4.8 Building Placer

Building Placer is a helper class of Building Manager. It does not send commands directly to StarCraft but provides Building Manager with the data about building positioning. There are multiple checks to be performed before a building can be constructed. First, the location must be valid or even specific for some buildings (like refineries). Second, the buildings cannot overlap and there must be some space left between them for the units to move freely. Third, some buildings can be extended so they may need additionial space later (for instance Terran Factories). What is more units that occupy a tile may also interrupt the construction.

The module is responsible for checking all the constraints and return the data back to the Building Manager.

3.4.9 Combat Commander

Units in UAlbertaBot are divided into squads. By default there is maximum of one attacking and one defending squad . All units assigned to that squad are counted as combat units. Combat Commander job is to assign proper units to the squad and give them proper orders (attack, defend, regroup or none).

3.4.9.1 Update

Combat Commander update invoked every 24 frames. Update sequence consists in three steps. Firstly squad data is cleared. All previously assigned units are removed from the squad. There is no memory about what happened before. Next workers are freed from Combat Commander control and returned to the Worker Manager. This happens only in early stages of the game when there are no combat units and our worker is assigned to attack enemy scout. Finally units are assigned to the squads.

3.4.9.2 Assign Squads

Units are assigned to the squads in four steps. At the beginning scout defense squads are assigned. Each occupied region (small rectangular part of the map) is checked for the presence of enemy units. As scout defense aquads are actually one-unit worker squads they only defend against enemy scouts. Therefore, the procedure checks if the only enemy in a region is a single worker. If is correct our worker is assigned with an order to defend the threatened region.

Secondly defense squads are assigned. They consist of normal, combat units that are to defend our region. Combat units are chosen by the Game Commander (so units like Protoss Observers are counted too) and passed as an argument of the Combat Commander update method. The method now begins to estimate how many defenders are needed. For every enemy worker in a region there is one unit assigned do defend. For each enemy combat unit there are three units assigned. Defenders are also divided into ground and flying defenders, counted separately. For instance to prepare defense against two Probes, two Zealots and a Carrier there would be ten units assigned. Seven with ground weapons (one to coutner the Probe and six to counter the two Zealots) and three units with air weapon (to destroy enemy flying unit: the Carrier). Finally, defense order is given to the squad. If for some reason there are still some idle units, they are ordered to defend.

Then the attack squads are formed. They are formed of all the combat units left to assign (what means not assigned to defense squads in the previous step). After the units were assigned it is checked if Strategy Manager allows us to attack. If it does the attack is performed: units are ordered to attack region, enemy buildings, visible units or to explore.

3.4.9.3 Attack

Attack is performed with all units assigned to attack. If our units are not near attacked enemy region, they are ordered to move towards it. If they are close enough, they will attack enemy buildings and units with priorities defined by the specific Micro Managers. Lastly if there is no enemy region known attacking units will begin to explore the map beginning from the least explored region (with the largest frame count since it has been last seen).

3.4.10 Squad

Combat Commander divides units into squads and gives them orders but what happens next is up to the Squad class. There are many units available for each race in StarCraft and there are a number of strategies that apply to each unit type. How units behave in combat is called micro. Sometimes two unit types can share the same micro. For instance micro for Protoss Dragoons may work quite well for Terran Marines. Both units are ground units with ranged weapon that can shoot to flyers. The similarities end here, though. Marines are much cheaper, have lower hp, shorter range and may be healed with medics. They can also enter Bunkers for extended range and protection. What is more, Marines can be upgraded to use Stimpacks, which is obligatory to use them effectively.

To deal with the problem of diverse unit behavior there are specific Micro Managers which all inherit from the Micro Manager class. The main purpose of the Squad module is to assign all given units to the proper Managers. The Squad module does not directly send commands to StarCraft. It is done by the Micro Managers which are registered in the squad.

3.4.10.1 Update

Squad update is called by the Combat Commander at the end of its own update. At the beginning Squad unit vector is refreshed to ensure that unit data is up to date. Next it is checked if the squad should regroup or attack. In the former case units are ordered to retreat towards the starting location. In the latter case the units are given order passed to the Squad by the Combat Commander. Specific unit Micro Managers are invoked one by one.

3.4.10.2 Set Manager Units

To give a unit order the unit must be assigned to some Micro Manager before. Function iterates over all completed, existing units and adds them to a proper vector, passed to the Micro Manager at the end. Method allows us to easily modify unit behavior by changing the manager from one to another. Implementing new manager implies changes to this method as well. Units are assigned to managers from the most specific to most general constraints. For instance Terran Marines are to be assigned before ranged units, because all Marines are ranged units as well but not all ranged units are Marines. Each unit type must be added to one and only one Micro Manager.

3.4.11 Micro Managers

Micro Manager is a class which all the specific micro managers derive from. It implements several methods invoked inside Squad module or used by specific micro managers. Micro Manager stores information about units assigned to it and a given order. It is a bridge between specific micro manager and the Squad module. The module also sends commands directly to StarCraft through BWAPI.

3.4.11.1 Execute

Execute is the main method of Micro Manager, called by the Squad. At the beginning it is checked if there are units assigned to the manager. At the start of the game there are no combat units trained at all so all the managers have nothing to do. Manager also responds only to attack or defense orders. Other orders are ignored and the method returns.

Defense order works only in some radius around the order location. Any units out of that radius are excluded. Both attack and defense orders attack every units on its way. It is done by executing a unit specific micro manager.

3.4.11.2 Smart Move, Attack and Attack Move

BWAPI provides interface to send StarCraft orders to the unit such as move, attack, follow or cloak. It does not provide any checks, however. The smart methods are kind of wrappers around BWAPI commands. They ensure that commands are issued once per frame and that no more than one command is sent to a unit per frame. In debug mode they also provide visual information about the order position.

3.4.11.3 Specific Micro Managers

By default there are four micro managers implemented: melee, ranged, transport and detector. The Transport Managers is not finished actually, so there are three usable managers. Squad module makes sure that each manager gets a proper set of unit. Protoss Dragoons, Terran Marines and Zerg Hydralisk go to the Ranged Manager. Protoss Zealots, Dark Templars and Zerg Ultralisks go to the Melee Manager and so on.

Every manager has to have executeMicro method implemented, which is called by the Squad through the general Micro Manager module. Managers are allowed to implement any number of helper methods though. For instantce it is commonly used tactic to "kite" the opponents with ranged unit. It means that if we outrange the opponent's units we can try to keep the distance while still shooting at the enemy. Ranged Manager implements this tactic. The same behavior would prove useless when applied to the melee units, so the Melee Manager controls its units differently.

New managers can be easily created by making a new class deriven from the Micro Manager. It is recommended for each unit type to have its own manager to maximize its effectiveness.

4 Expert-Designed StarCraft Strategy for the Terrans

4.1 Goal

Despite UAlbertaBot is able to play all the three StarCraft races, it was designed to do it well with only Protoss. The Zerg and Terran races are but an additives. What all three races share in common works well for the Zerg and the Terran. Unfortunately much of what is race specific works properly only for the Protoss.

Our goal was to extend UAlbertaBot in such a way that it would be able to play one of the other races (we have chosen Terrans) well enough to complete with the base, Protoss UAlbertaBot. To achieve this goal we provided several extensions and upgrades to the base code. Some parts were slightly changed, some were rewritten completely, others were removed or added. As there was a number of changes, in this chapter we present modifications that we found the most relevant. We observed that even very small changes, like changing unit flee radius or modifying unit attack priorities, have frequently huge consequences to the final game result.

4.2 Changes to the Production Modules

To adjust UAlbertaBot to work well with Terrans provided several modifications to the Production Manager, Building Manager and the Strategy Manager.

4.2.1 Addons

We observed that build order searching does not work properly for Terrans. Addons were ignored completely. They were not even considered as a requirement action for other buildings. It prevented us from applying various tactics including usage of Siege Tanks, Spider Mines tech or Battlecruisers. Many Terran upgrades and techs are also researched in addons.

To deal with the problem we modified we modified Building Manager to treat addons in a different way than other buildings. They have to be treated separately, because they are a special type of building that is not built by a worker but by the building itself.

They must be attached to the parent building to work properly. If damaged, they are repaired by an SCV, as the other buildings.

It did not solve the problem completely though. Build order search still ignored them in the search and we had to provide a helper method that injects addons into the queue wherever there is a need to do so.

4.2.2 Research and Upgrades

We wanted to use abilities extensively in most of our strategies. Abilities usually require a proper technology to be reesearched before they can be used. Base UAlbertaBot does not use neither research nor upgrades, so any flaws in this area are irrelevant for the sake of default strategies efficiency. We observed however, that upgrading or researching tends to block the production queue completely. During this time, units are not trained and buildings are not constructed. The build order waits until the research or upgrade is done. For some upgrades (those which are done fast) it might not be a serious problem but there are upgrades that last for a very long time. It resulted in the base staying in a "frozen" state for a hundreds frames.

The solution we found proved to be quite simple. As soon as an upgrade or tech was started, we remove it from the queue. Build order does not wait until it is done because it cannot see it. It would not work for actions that are required as a perquisite resource for other actions but it is not the case with the upgrades.

4.2.3 Strategy Manager

There was only one opening build order provided for Terrans by default. And it was a simple one, executing a Marine rush strategy. We wanted to have more options and diversity. To achieve that we implemented a dozen of different build orders. All of them are based on the expert knowledge.

Build orders are then added to the vector that stores available strategies. Only openings that are effective against the opponent's race are added. The rest of them is not available. Some strategies are versatile enough to be used against any opponent race.

4.2.3.1 Mid Game Indicator

It is not uncommon to try to rush the opponent and if it fails, focus on building the economy. We consider the game before the rush as an early game and after the rush a mid game. We want to rush the enemy as soon as possible but if the attack fails, we may - according to the chosen strategy - suppress further attacks until we build a significant force. We added an indicator that tells us whether the mid game has already started. The strategies can then adjust the goals and attack orders according to the state of the game.

4.2.4 Winning Indicator

Knowledge of who is currently winning the game is important to the strategy but not always obvious. Winning side should seal the victory by proceeding with the attack while loosing side should rather try to fight off the enemy from safer positions and attack only when the enemy was repelled.

Simple counting units killed by each side is not a good indicator. Player who lost ten Zerglings to kill five SCVs is on the winning side. He weakened opponent's economy at the cost of loosing some irrelevant units.

To estimate if we are winning or loosing we count the mineral and gas balance of the units lost by each side. We count total mineral price of units we killed and then we add the total gas price of those units multiplied by a constant value of 1.5 (because gas is more valuable than minerals). Then we do the same for the we lost to the opponent. The difference between the two numbers indicates who is currently winning.

To compensate for losses in the past, we add a constant number equal to 1000 to our winning score every ten thousand game frames. Without the compensation losing a battle in the early game could force us to defend for the rest of the game and never attack. This way we try to attack even if we are loosing and quickly withdraw if the attack does not go as planned. The equation for the winning indicator is the following: $\text{winning\_indicator} = \text{killed\_units\_score} - \text{lost\_units\_score} + (1000 * (\text{game\_frames}\ \mathbf{div} 10000))$.

4.2.4.1 Terran Build Order Goals

The main purpose of the Strategy Manager is to create effective build orders. We added several methods that set up goals for different strategies such as Terran MnM Rush or Three Factory Vulture Rush. Not all of the openings have their own build order goals but some goals may be used by several different openings. Each strategy has also its own methods responsible for attack permissions and base expansions.

4.2.5 Queue Injector

We observed that relying only on the build order search frequently causes production delays. Units were built sequentially rather than concurrently. Gas and minerals were hoarded while production was idle. Production Manager waited for waited for some unit to complete doing nothing, instead of proceeding to the next action. Such situation resulted from the build order search not being adjusted to the Terran race well enough. We wanted the production to be constant and the resources to be spent on a regular basis.

To deal with the problem we added code to the Production Manager to inject some units directly into the empty queue. The solution was simple but not versatile at all. Training SCVs is good when there are too few of them. Training Marines might be good at the beginning or to fill the bunkers but obsolete in the end game. Siege Tanks perform very well in the end game but not against flying units. We needed much more adjustable solution, described in the next section.

4.2.6 Queue Constructor

To effectively manage build order injections we implemented a whole new class. We kept the build orders inside the Strategy Manager module as simple as possible and transferred most of the build order management to the new class. The class constructs a queue in the Production Manager format and no further conversions are needed. When a proper Queue Constructor method is invoked it replaces the queue inside the Production Manager. Therefore, the new queue works exactly the same as the default one and changes are invisible for the base UAlbertaBot modules. Resource management and action issuing remained unchanged.

As every strategy needed its own method to set the goals, now every strategy needed also equivalent method inside the Queue Constructor. Fortunately several different strategies could use the same method to inject actions into the queue. We provided a default queue constructor and focused on a few specific ones for our most commonly used strategies.

Because inside the Queue Constructor we tailored build orders apart from the Build Order Search, we needed our own mechanism to manage the action prerequisites. Simply inserting a Marine into the queue would not work because there would be no Barracks to train it.

4.2.6.1 Action Prerequisites

For each action we provided a method that automatically adds the prerequisites to the queue. Each method adds only its direct prerequisites. Next each prerequisite adds requirements for its own. The chain continues just to the most basic actions that have no requirements. Assume that we want to built a Machine Shop. This is a Factory addon and its only prerequisite is a Factory. We queue the Factory. The Factory has its own direct prerequisite though, the Barracks. Next we check prerequisites for the Barracks. As there is only a Command Center it is one of the basic buildings so we add them directly to the queue. As a result we get a queue with the Barracks as the highest priority item, Factory in the middle and the Machine Shop at the end. This way we ensure that when we reach the Machine Shop, all the requirements will be met.

4.2.6.2 Technologies and Upgrades

Apart from the constraints for a particular actions there are sets of actions that usually go together. Upgrades and technologies are especially concerned. Strategies that rely on bio units (which means biological units like Marines or Zealots, contrary to mechanical units like Vultures or Dragoons) sooner or later require those units to be upgraded. For instance Marines are much more effective with Stimpacks technology researched, Infantry Armor and Weapons upgraded and U-238 Shells upgrade. Armor and weapons are improved in three stages. The first stage requires just the armory. Second stage however, requires stage one upgrade and the Science Facility.

To make the upgrading easier we provided methods that group the related upgrades and technologies together and take care of all the prerequisites and order. Similar to the buildings and units constraints we made it in a form of chain of requirements from most advanced to the most basic actions. For instance Terran Weapons Upgrades is a part of Terran Marines Upgrades, while Terran Marines Upgrades is a part of terran Bio Upgrades. Unit upgrade management is much easier this way.

4.2.6.3 Cleaning the Production Queue

Many different actions have the same requirements. During the queue construction any repetitions are not considered. The same prerequisite may be added multiple time. Let assume we want to queue the Starport and the Machine Shop. They both require Factory and Barracks so those buildings will be added twice. The final queue is contaminated with unnecessary buildings. Because of that, just before the queue is returned to the Production Manager it is cleared of all the obsolete actions. Maximum number of some buildings (such as the Armory) is also limited by the cleaning method to some constant number.

4.2.7 Uncompleted Buildings

Unlike the Protoss and the Zerg, Terran workers can be killed while constructing a building. If such a situation occurs, the building remains uncompleted and abandoned. This leads to build order complications as UAlbertaBot still considers the building as under construction. An uncompleted building should either have a new worker assigned to or be destroyed. Which choice is better depends on the situation.

We experimented with both options and finally chose to destroy the building. Sending a new worker to the construction site resulted frequently in that worker being killed too. Destroying the building proved to be a safer option. Minerals and gas are fully returned when the building is destroyed during the construction (except for the Zerg), so the only cost is the time.

4.2.8 Workers Behavior

The way how the buildings and units are healed differs from race to race. The Protoss have health and shields. They do not regenerate health nor do they have units that repair or heal other units. However, their shields regenerate constantly with time. The Zerg units regenerate health fast. The same applies to buildings. In case of Terrans, there are specialized units for healing other units. Biological units are healed by Medic unit ability, while buildings and mechanical units can be repaired by the workers (SCVs).

While healing biological units is covered by the Medic Micro Manager, mechanical units and buildings need a special treatment from an SCV. We modified the Worker Manager and Combat Commander module to assign workers to the damaged buildings and units. Enabling Bunkers to be repaired proved to be a significant improvement during the early game stage. Especially against the Zealot rush strategy.

4.3 Changes to the Unit Management

We have found that the few micro managers implemented into the UAlbertaBot do not meet our expectations. They work well but are too general to unleash the full potential of any specific unit type. We implemented our own micro managers, based on the default ones. Moreover, we made a few changes to the Squad module and created some classes to help us with the unit management and desired behavior.

4.3.1 Squads Management

Before we started implementing the specific micro managers, there were some issues with the Squad module that we wanted to solve first. One of the most important consisted retreating.

4.3.1.1 Retreating

During the combat, UAlbertaBot estimates the battle outcome through the combat simulation. The simulation is done by a separate library called SparCraft [D.14c]. The outcome determines the retreat order. Winning prediction results in proceeding with the attack whereas loosing forces the units to retreat from combat and head towards the friendly base.

Unfortunately, as mentioned before, losing the combat (having all the units killed) not necessarily means that the outcome was negative. Sacrificing units to destroy target that is more strategically valuable may, in fact, win the game.

There was also another issue. Retreating does not stop the units from taking damage. During their journey back to the base, retreating units might be completely wiped out. Especially if enemy has faster units.

What is more, as the game is only partially observable, the simulation can only count units that are visible to us. As the combat proceeds units run in and out of sight what disturbs the simulation. Part of an overwhelming enemy force may not pose a threat to our force so the units engage. As soon as they uncover the rest of the enemy army, they fall back again immediately. The result of this is that the units take damage while dealing none.

To solve the problem we disabled the retreating behavior. We provided the solution based on our winning estimation and attack permissions granted by the Strategy Manager. More sophisticated retreating behaviors, such as kiting (running from the enemy while shooting at him), are handled by the specific micro managers.

4.3.1.2 Adding Units to the Unit Manager

Apart from assigning the proper micro managers to the particular unit types, we modified the Squad module to assign units also to our newly created Unit Manager class, described in detail in another section. The Squad module is responsible for setting all the properties required by the Unit Manager to effectively manage that unit such as type of the order, current strategy or unit movement mode. The Unit Manager cannot access any units that were not previously added by the Squad, which is desirable for the units that do not require the extended functionalities provided by the Unit Manager.

4.3.2 Unit Data and Unit Manager Extensions

To implement some more complex behaviors for Terran units we needed a module to control the state of our units. Squad module manages groups of units and can only distinguish one unit type from the other. Multiple units of the same type are not distinguished. The micro managers, on the other hand, manage and control specific, individual units but they do not store any data about them. They only iterate over assigned units, check the input and provide the output.

Some tactics, such as moving near map borders for the Wraiths, required us to store additional information about the particular unit. To deal with the problem we created two classes: Unit Data class, that stores the additional data and Unit Manager which manages that data.

4.3.2.1 Unit Data Extension

The UnitData is a simple container class. Every information that may be needed by some micro managers should be stored within. Each unit has its own ID assigned, so the Unit Manager can easily distinguish one unit from another. Depending on a unit, different data is stored to the manager. For the Terran buildings that can lift off, the landing position is remembered. For the Wraiths there are waypoints and the final destination location. For the Vultures there is information about their Spider Mines, and so on.

Without the UnitData class many strategies used by the micro managers would be hard to implement as there would be no unit states. Thanks to this data we can adjust our units behaviors to their previous actions. For example when in one frame Vulture Micro Manager orders the Vulture unit to put a mine, which takes a dozen or hundreds of frames, he will remember next frame that an order was already given and react accordingly to that. Otherwise the Vulture would try to put a mine in a different location every frame, resulting in a weird and inefficient behavior.

4.3.2.2 Unit Manager

The only module that has direct access to the UnitData class is the Unit Manager. While the UnitData stores units' states, the Unit Manager provides a sort of interface for other classes to access those states and data. Another responsibility of the Unit Manager is creating and storing UnitData vector, as well, as checking the correctness of the stored data. What is more, storing, accessing and managing waypoints, used by some of the micro managers, is also managed by the Unit Manager.

Another important role fulfilled by the class is a proper positioning of the lifted Terran buildings that are about to land. While a building has an addon attached it is left on the ground when the building lifts off. To make the addon usable afterwards, the building must land in the exact same location as it stood before.

4.3.3 Waypoint Creator

Most units use share a simple algorithm to move around the map. They move the shortest way towards the order position (which is usually the enemy base). In case of flying units it is always a straight line from their current position to the target position. Some strategies, however, demand a different approach.

To make implementation of new strategies easy, we provided a class called Waypoint Creator. The class responsibility is to create a set o waypoints (map positions) for a given unit. The waypoints are stored in a UnitData vector stored by the Unit Manager.

We implemented waypoints as a stack, so the positions are added in the reversed order to that which the unit will move. The order destination is pushed onto the stack at first, while the first waypoints to be visited is pushed last. This way, when a unit reaches a waypoint proximity, the waypoint is popped off the stack and the units moves towards the next one.

After the waypoints are created, they are managed by the micro managers, through the Unit Manager. The Waypoint Creator module is used only once, just after the UnitData for a given unit was created.

4.3.4 Micro Managers

As we mentioned before, there are only four micro managers implemented into the UAlbertaBot by default: melee, ranged, detector and transport manager, where the latter is not actually finished. Both melee and ranged managers are working very well but are way too general for our purposes. The detector manager is not useful for the Terrans, because it was written with Protoss Observer unit in mind, which has no Terran equivalent.

To improve the effectiveness of our units, we provided a separate micro manager for each unit type we considered important for our strategy. In the following sections we describe the behaviors we implemented into our managers. With the exception of Science Vessel Manager, they are all created around the UAlbertaBot's Ranged Manager.

There are several similarities shared by all of the micro managers. There is a central method of every manager, called execute micro. This the only specific micro manager method that is invoked by the Squad module. The execution of a micro consists of selecting unit targets, choosing the target with the highest priority and attacking the target. Where attacking the target may range from simply executing BWAPI attack command to sophisticated behaviors such as kiting, using abilities, evading enemy fire and so on.

4.3.4.1 Kiting

Many Terran units benefit greatly from using kiting, which is a technique used commonly for the ranged units. To kite an enemy means that we try to shoot at the opponent's units while constantly running away from them to minimize the damage caused by the return fire.

Not all targets can be kited. We must outrange the opponent's unit to kite it effectively. Otherwise it is us who are being kited. The ideal situation for kiting is when we not only outrange the opponent but while we also move faster. As kiting may decrease our damage output, unit that cannot attack (like workers, Protoss Observers or buildings) should not be kited.

Our kiting implementation is based on the UAlbertaBot's Ranged Manager one, which we improved and adjusted for the particular unit types. Because kiting works differently for each unit type, we describe kiting for each unit separately in the following sections.

4.3.4.2 Marine Manager

Marine is the most basic Terran unit. Regardless of the chosen strategy, Marines will very likely appear in the game. They are cheap and expendable. In large numbers and supported by Medics, they become a significant threa for opponentst. What is more, Terran defensive building, the Bunker, requires Marines or other biological unit to operate. The Bunker compensates for the Marine's low hit point value and extends the range of its weapons, transforming it into a good defense unit.

Marines are somewhere in between other races' basic units. They are stronger than the Zerg Zerglings but weaker than the Protosss Zealots. Unlike the mentioned counterparts though, Terran Marines are ranged units, which grants them some possibilities, especially a potential for kiting.

Behavior of the Marines depends on the strategy chosen by the Strategy Manager. For the Vultures and Tanks and Wraith Rush strategies, Marines serve only as a defensive unit and they will never attack the enemy base. Among other strategies they are considered an early rush unit and will attack as soon as they are recruited. Unless the Strategy Manager revokes the attack permission.

Regardless of the chosen strategy, before Marines are ordered to either attack or defend they will be directed to fill the empty Bunkers. There is a space for four Marines in every Bunker and each Bunker is filled to the full capacity, before moving Marines are redirected to another one.

When attacking the enemy, Marines execute the kiting technique. They try to shot at the enemy, then while weapon cooldown is active, run away from the threat, and then shoot again. The point towards which the Marines are retreating lies exactly in the opposite direction to the average enemy position, from the view of a particular Marine.. By default it was the opposite position of the target unit, but it resulted sometimes in situations where a Marine tried to kite one Zealot but was running towards the several other.

Marines can have also their range upgraded and the ability to use Stimpacks researched. While the former is just an useful upgrade, the latter may change the result of the game. Using a Stipack causes the Marine to immediately lose ten hit points but its movement speed and attack speed is doubled temporarily. Along with Infantry Weapons upgrade, which boosts their damage by 50%, Marines are able to inflict a tremendous number of damage.

However, Stimpacks should be used with caution. Every ability activation costs ten hit points but the effect timer is resetted, not extended. And the boosts are temporary, while hit point loss is permanent, unless healed by a Medic.

Thus the Marine Manager activates Stimpack always and only if there is an enemy and we are in its weapon range and the enemy is not a Protoss High Templar. We do not activate Stimpacks before we are in the enemy's range to save hit points when they are needed to chase or running away from the opponent's unit.

In StarCraft a unit damage output does not depend on the number hit points the unit currently has. Thus it is better to destroy one unit completely instead of wounding two separate. This is why Marine Manager orders the Marines to always attack a unit that has lowest hit point value. For Protoss units this is a sum of hit points and shields.

With the implemented tactics, the Marines are a good defensive unit that can be very effective in large numbers supported by Medics or against some particular foes (especially slower, melee units) .

4.3.4.3 Vulture Manager

Vultures are a type of a very fast Terran ranged, mechanical unit .They are more expensive than Marines but still cheaper than the Protoss Zealots. Their speed can be further upgraded to achieve the highest value of all StarCraft units. The most interesting and strategically valuable is their ability to put mines, though.

With the Spider Mines technology researched, every Vulture can put up to three mines. The mines are invisible to the enemy (revealed by a detector) and automatically attack enemy ground units within some range. When a mine explodes, it damages all nearby units. The closer to the explosion center, the greater the damage. Properly placed mines can wipe out entire enemy force at once. They work particularly well against Zealots.

When Vulture is out of mines (or the technology has not been researched yet), its behavior is very similar to that of Marine. Vultures try to kite enemy units if they can. Even though Vultures are ranged units, they cannot attack flying targets, so contrary to the Marine, enemy flying units are not counted as the Vulture targets.

Vultures prioritize targets that are faster than them (which happens only if their engines has not been upgraded). Among slower targets always the one, that has the greatest range will be chosen. Slow targets with short range weapons can be easily kited by the Vultures or destroyed by the Spider Mines, thus they are attacked when no greater threat is present.

Vultures that have mines at their disposal, prioritize putting them above everything else, as the mines are much more valuable that the Vulture itself. When an enemy is spotted, the Vulture will either put a mine and retreat or retreat and put a mine. The choice depends on the distance to the enemy unit. Mines need some time to burrow, during which they can be easily destroyed. That is why if the enemy is close to the Vulture, the manager will order a unit to retreat and put the mine away farther from the enemy. When Vulture tries to put a mine, the state is recorded by the Unit Manager. Doing so prevents the Vulture Manager from issuing commands to the unit until the mine is placed.

If the enemy unit chooses to chase the Vulture, it will step right into a newly placed mine. If the enemy decides to run away, the Vulture will follow it and attack. Multiple Vultures are very hard to deal with a force of melee units. Even stronger ranged units can be easily lured into an exploding trap and wiped out completely. Due to high speed and kiting, it may take long time to destroy the Vultures, during which more and more will be produced. Vultures and Tanks strategy uses the Vultures to slowly push the opponent towards his base and keep him there until the Siege Tanks arrive.

4.3.4.4 Tank Manager

Siege Tanks are powerful, ranged units that are able to enter Siege Mode when the proper technology is researched. They are expensive, but the cost is justified. Tank have high armor, hit points and damage. Furthermore, with Siege Mode researched they have the highest range among all StarCraft units in addition to the splash (area) damage.

As the name might suggest, Siege Tanks are excellent for base besieging. They stay out of range of any turret in the game and, thanks to the splash damage, they can destroy groups of enemy units trying to defend their base at once.

Siege Tanks have are vulnerable to air unit though, because they have only a ground weapon. Another its weak point is the minimal range when in Siege Mode, so if the enemy approaches, the Tanks are harmless until they enter the Tank Mode. However when in Tank Mode, Siege Tanks can kite enemy units well, as they can shoot almost without stopping.

We use Tanks mostly with our Vultures and Tanks strategy. They enter the game to make a final push towards the enemy base. Vultures and Spider Mines take the role of scouts to reveal the enemy units, because Siege Tanks have shorter vision than the weapon range.

Tank Manager ensures that a proper mode is chosen to a given situation. When the enemy is spotted, the tank enters Siege Mode, but if enemy get close and is still approaching, Siege Tank enters normal mode and begins kiting. Closest enemies are always prioritized. Tanks that are idle or have been given defense order will immediately enter Siege Mode.

In large numbers and supported by other units like Vultures, Wraiths or Goliaths and a Science Vessel for detection, the Siege Tanks form a near unstoppable formation, called the "deatchball" by StarCraft players.

4.3.4.5 Wraith Manager and Border Movement

Wraiths are the Terran air-superiority fighters. They are expensive and very situational. Although Wraiths can attack both air and land targets, their ground weapon is very weak. We use them either as an anti-air unit to support other units, such as the Siege Tanks or as a rush unit in one of the Wraith Rush strategies.

The Wraith Manager will prioritize flying targets and detectors. Powerful end game units like Terran Battlecruisers and Protoss Carriers are the top priority targets for the Wraiths. Because Wraiths' ground weapons are so weak, they prioritize workers over other ground units, which can destroy enemy economy while the other units deal with enemy forces.

Even though Wraiths' ground weapon is weak, an early attack with air units against an enemy that has no air defense can easily win the game. This strategy if further enhanced by the Cloak ability, that allows Wraiths to become invisible for some period of time. To attack such cloaked Wraith, the enemy needs not only anti-air unit but also a detector.

We improved rush strategies further, by implementing border movement for the Wraiths. Waypoint Creator creates a set of waypoints that lead from the Wraith to the nearest border. The Wraith flies from waypoint to waypoint along the edges of the map, until it reaches the enemy base from behind, where the base is often most vulnerable. Then the Wraith attacks the enemy workers, disturbing the opponent's economy. If we surprised the enemy, it would be very hard for him to fight back the cloaked Wraiths.

The Border Movement algorithm works as following. At the beginning the order destination is set. Then attacker (Wraith) and the target closest border positions are designated. Next the border waypoints are computed. There are eight waypoints in total. One in each corner of the map and one in the middle of each map edge. After that the target border position, which means the closes border location from the target, is pushed onto the stack. Then it is decided whether to move clockwise or counterclockwise, whatever path is shorter. Next the border position waypoints is pushed onto the stack, then the next waypoint and so on, until the closest waypoint to the attacker is reached, what ends the algorithm. As the result we get a vector of waypoints along the edges, from the attacker to the order position.

However, Wraith Rush strategies are risky, because should the rush fail, we would be put into a disadvantageous position with our poor economy and weak Wraiths.

4.3.4.6 Battlecruiser Manager

Battlecruiser is a very powerful Terran late game unit. Due to a very long technology tree, a high resource cost and because they are potent only in groups, the Battlecruisers are mostly used in the very late game to seal the victory.

The unit has strong armor, huge firepower and the highest hit points value in StarCraft, making it the only unit capable of surviving a Nuclear Strike. After the proper research is conducted, the Battlecruisers are also equipped with the Yamato Gun ability, which makes 260 damage to the single target. The Yamato Gun is capable of destroying most lower tier units in one shot and causing significant damage to any others.

The Battlecruisers prioritize other late game units, especially the Protoss Carriers. They use Yamato Gun to destroy turrets in one shot and to help them deal with carriers. After high tier units are destroyed, Battlecruisers focus on other flying units and ground units with anti-air weapons. In case of attacking Protoss buildings, Battlecruisers destroy Pylons first, which turns nearby Protoss buildings inactive. Because they are one of the most powerful units in the game they do not kite, but try to destroy the enemy completely.

When a force of Battlecruisers is built, the game is usually won. The problem lies in building that force.

4.3.4.7 Science Vessel Manager

In our strategies we use Science Vessels as a mobile detectors. They are much more than that, though. Fully upgraded Science Vessel is a powerful support unit witch several abilities that can either weaken opponent's units or strengthen our own forces.

As Science Vessel cannot attack on its own the Science Vessel Manager takes care that the units under its control stay out of the enemy fire. Vessels always move with at least one another friendly unit and try to stay away from enemy fire while still detecting cloaked units. They never go towards the enemy alone.

Long technology tree and high cost of the unit makes Science Vessels a late game unit. They are very useful in detecting enemy units but cannot solve the problem of early game Dark Zealot rushes (Protoss permanently invisible unit), due to a simple fact that it takes much time to build them.

5 Experiments and Results

5.1 StarCraft Tournaments

Before StarCraft became a test-bed for AI methods, there were other RTS AI competitions. The "ORTS RTS Game AI Competition", held from 2006 to 2009, was, arguably, the most popular one. Since StarCraft has become the platform of choice for testing RTS AI methods (thanks to the development of BWAPI library), two new tournaments appeared on the stage. The first one is "AIIDE StarCraft AI Competition" (held since 2010) and the second is "CIG StarCraft RTS AI Competition" (held since 2011). Both are organized annually are held as a part of international conferences concerning AI in games topics: "AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment" (AIIDE) and IEEE Conference on Computational Intelligence and Games (CIG), respectively.

The AI competitions were made to reflect real StarCraft tournaments, played by the professional players. Because of that, the tournament bots could be easily evaluated against human opponents, if necessary. Below we enumerate the most notable tournament rules that the bots have to follow.

1. **Cheating is not allowed.** Unlike the default AIs, built into StarCraft, the tournament bots had to play as human players do, with limited information about the game state and without any additional resources.
2. **Internet access is forbidden.** The bots are allowed to gather and store information during the tournament, but only from the tournament itself. Accessing remote databases or additional computing power is now allowed.
3. **Games are played on the "fastest" setting and delays are counted as a loss.** "Fastest speed" means that there are around 24 frames per second. This is the default speed setting for at professional tournaments. Bots that cause delays are disqualified, which means that they must be able to process at least 24 game frames per second. Taking into account the complexity of some AI algorithms this is a serious constraint.
4. **There is a 86400 frames time limit.** Every game is stopped after the number of frames that corresponds to the one hour of simulation. If there has not been a winner before the game ends, the bot with the higher game score wins. This

constraint prevents exploiting strategies when one player would hide in some hardly accessible position on the map and wait till the game ends. Another reason is a purely practical one: without a time limit, some games could never end.

Both tournaments use a special software to automatically play scheduled games. This tournament manager software is provided by David Churchill specifically for the purpose of those tournaments [tou14]. The software is written in Java and uses a client server architecture.

To evaluate the effectiveness of our solution, we wanted to test it against other bots and default StarCraft AIs. Fortunately for us, our bot was accepted for participation in two largest StarCraft AI tournaments: "2014 AIIDE StarCraft AI Competition" and "CIG 2014 StarCraft AI Competition". What is more, both competitions follow exactly the same set of rules [com14], so our bot could participate without any changes in both of them, making the results comparable. We called our bot TerranUAB to imply that this is our Terran extension to UAlbertaBot.

5.2 AIIDE StarCraft AI Competition Results

This year[1] in AIIDE StarCraft AI Competition there were ten tournament maps and eighteen participants. Each bot played 1139 games in total, which means 67 games against every other bot except itself. The tournament lasted for almost one week and provided us with very useful data, which we present in the following sections.

Table 5.1 show the list of participants in this year's AIIDE tournament, while Tables 5.2 and 5.3 show the participants in 2013 and 2012 respectively.

It is worth to note that this year's tournament had as much participants as the two previous combined. Not to mention that out of eight bots that participated in 2013 tournament, only two were new. In the recent tournament, there were as much as ten new bots: Bonjawa, CruzBot, HITA, LetaBot, MassCraft, MooseBot, NUSBot, Oritaka, Yarmouk, and our own participant, TerranUAB. The other eight were veterans of the previous competitions. The 2014 tournament was definitely the largest event of its kind so far.

Another interesting observation is that out of eighteen participants, only one bot (HITA) played the Zerg race. The number of bots playing other two races were almost equal, with nine Protoss and eight Terran participants.

[1]year 2014

Bot Name	Author	Affiliation	Race
Aiur	Florian Richoux	University of Nantes	Protoss
Bonjwa	Dustin Dannenhauer	Lehigh University	Terran
BTHAI	Johan Hagelback	Linnaeus University	Terran
CruzBot	Daniel Montalvo	UC Santa Cruz	Protoss
HITA	Hiroto Takino	University of Electro-Communications	Zerg
IceBot	Kien Nguyen Quang	Ritsumeikan University	Terran
LetaBot	Martin Rooijackers	Maastricht University	Terran
MaasCraft	Dennis Soemers	Maastricht University	Protoss
MooseBot	Adam Montgomerie	University of Bristol	Protoss
Nova	Alberto Uriarte	Drexel University	Terran
NUSBot	Author List	Institute of Systems Science, NUS	Protoss
Oritaka	Yoshitaka Hirai	University of Electro-Communications	Terran
Skynet	Andrew Smith	Independent	Protoss
TerranUAB	Filip Bober	Poznan University of Technology	Terran
UAlbertaBot	David Churchill	University of Alberta	Protoss
Xelnaga	Ho-Chul Cho	Sejong University	Protoss
Ximp	Tomas Vajda	Independent	Protoss
Yarmouk	Abdelrahman Elogeel	Independent	Terran

Table 5.1: 2014 AIIDE StarCraft AI Competition participants [aii14]

Bot Name	Author	Affiliation	Race
Aiur	Florian Richoux	University of Nantes	Protoss
BTHAI	Johan Hagelback	Blekinge Institute of Technology	Terran
ICE	Kien Quang Nguyen	Ritsumeikan University	Terran
Nova	Alberto Uriarte	Drexel University	Terran
Skynet	Andrew Smith	Independent	Protoss
UAlbertaBot	David Churchill	University of Alberta	Protoss
Xelnaga	Ho-Chul Cho	Sejong University	Protoss
Ximp	Tomas Vajda	Comenius University	Protoss

Table 5.2: 2013 AIIDE StarCraft AI Competition participants [aii14]

Bot Name	Author	Affiliation	Race
AdjutantBot	Nicholas Bowen	University of Central Florida	Terran
Aiur	Florian Richoux	University of Nantes	Protoss
BroodwarBotQ	Gabriel Synnaeve	College de France	Terran
BTHAI	Johan Hagelback	Blekinge Institute of Technology	Zerg
Nova	Alberto Uriarte	Drexel University	Terran
SCAIL	Jay Young	University of Birmingham	Protoss
Skynet	Andrew Smith	Independent	Protoss
SPAR	Simon Chamberland	Universite de Sherbrooke	Protoss
UAlbertaBot	David Churchill	University of Alberta	Protoss
Xelnaga	Ho-Chul Cho	Sejong University	Protoss

Table 5.3: 2012 AIIDE StarCraft AI Competition participants [aii14]

5.2.1 Overall Tournament Statistics

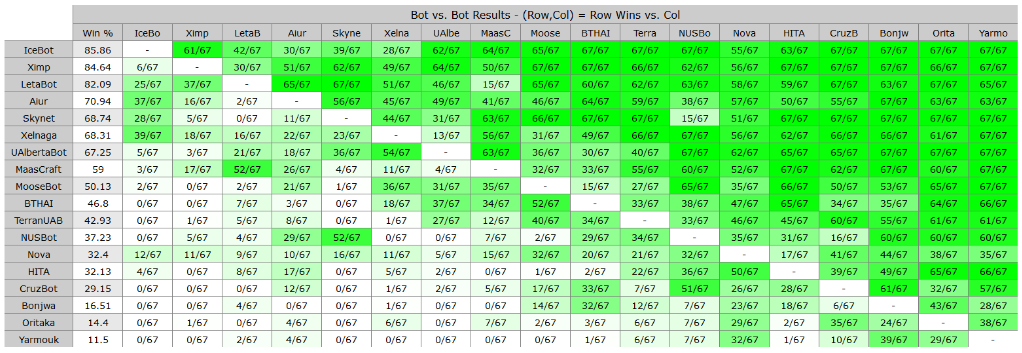
The winner of 2014 AIIDE StarCraft AI Competition was the IceBot, playing Terrans, with 85.86% win rate, as shown in Table 5.4. The top three bots were very close to each other with 84.64% win rate (Ximp) and 82.09% win rate (LetaBot). There was only 3.77% difference between the winner and 3rd place. Between the third and 4th place there was a 11.15% gap, then there were another group of four bots: Aiur, followed closely by Skynet, Xelnaga and our Protoss counterpart, UAlbertaBot. The difference between the best and the worst within this group was only 3.79%.

				Overall Tournament Statistics				
Bot Name	Games	Win	Loss	Win %	AvgTime	Hour	Crash	Timeout
IceBot	1139	978	161	85.86	14:39	9	21	0
Ximp	1139	964	175	84.64	16:24	10	28	7
LetaBot	1139	935	204	82.09	11:34	20	2	0
Aiur	1139	808	331	70.94	13:41	30	1	0
Skynet	1139	783	356	68.74	11:04	11	0	0
Xelnaga	1139	778	361	68.31	16:16	59	34	0
UAlbertaBot	1139	766	373	67.25	11:13	18	16	24
MaasCraft	1139	672	467	59	13:53	35	22	0
MooseBot	1139	571	568	50.13	12:33	25	14	5
BTHAI	1139	533	606	46.8	18:17	67	290	0
TerranUAB	1139	489	650	42.93	14:12	25	34	0
NUSBot	1139	424	715	37.23	12:52	25	162	23
Nova	1139	369	770	32.4	10:50	33	389	3
HITA	1139	366	773	32.13	10:45	1	506	8
CruzBot	1139	332	807	29.15	17:07	70	0	10
Bonjwa	1139	188	951	16.51	14:45	81	0	0
Oritaka	1139	164	975	14.4	13:59	58	0	0
Yarmouk	1139	131	1008	11.5	14:06	57	42	0
Total	10251	10251	10251	N/A	13:47	317	1561	80

Table 5.4: Overall tournament statistics [D.14b]

Out of the new bots, LetaBot proved to be very effective and was the only new participant in top 7. The next best new bot was MassCraft placed 8th and then MooseBot, only 0.87% behind. After that there was BTHAI, a veteran of previous tournaments and then another new bot, our own TerranUAB. The new participants achieved 37.51% win rate on average. Veteran achieved much better, 65.53% win rate on average.

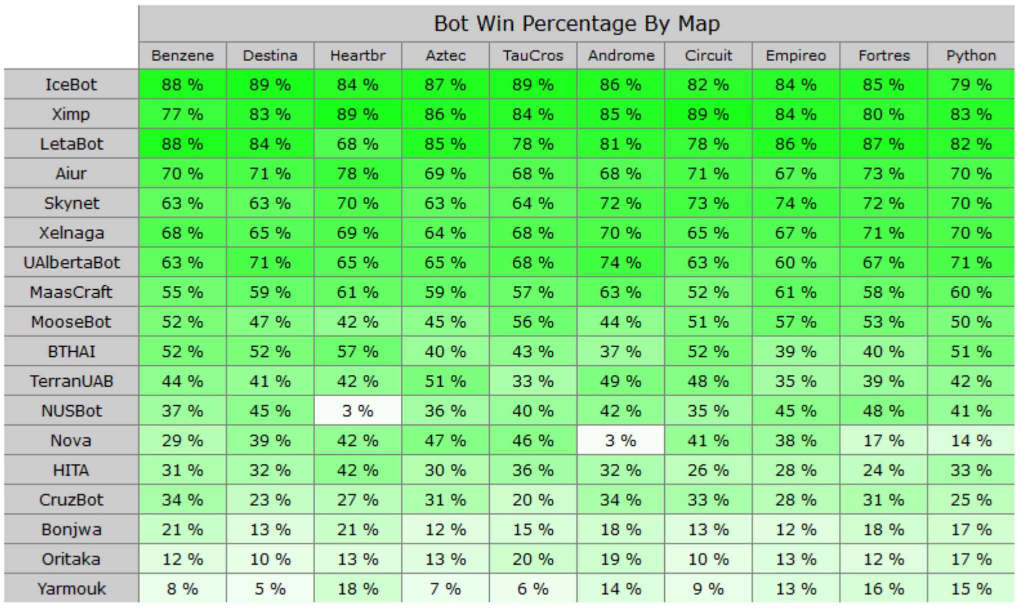
Interestingly, the best and the worst bots played both the Terran race. However, an average Terran bot had 41.47% win rate, while average Protoss one had much better 59, 49% win rate. This is likely due to the fact that the veterans of previous competitions, played mostly Protoss and, as was already shown, veteran bots performed much better than the new ones.



Bot vs. Bot Results - (Row,Col) = Row Wins vs. Col	Win %	IceBo	Ximp	LetaB	Aiur	Skyne	Xelna	UAlbe	MaasC	Moose	BTHAI	Terra	NUSBo	Nova	HITA	CruzB	Bonjw	Orita	Yarmo
IceBot	85.86	-	61/67	42/67	30/67	39/67	28/67	62/67	64/67	65/67	67/67	67/67	67/67	55/67	63/67	67/67	67/67	67/67	67/67
Ximp	84.64	6/67	-	30/67	51/67	62/67	49/67	64/67	50/67	67/67	67/67	66/67	62/67	56/67	67/67	67/67	67/67	66/67	67/67
LetaBot	82.09	25/67	37/67	-	65/67	67/67	51/67	46/67	15/67	65/67	60/67	62/67	63/67	58/67	59/67	67/67	63/67	67/67	65/67
Aiur	70.94	37/67	16/67	2/67	-	56/67	45/67	49/67	41/67	46/67	64/67	59/67	38/67	57/67	50/67	55/67	67/67	63/67	63/67
Skynet	68.74	28/67	5/67	0/67	11/67	-	44/67	31/67	63/67	66/67	67/67	67/67	15/67	51/67	67/67	67/67	67/67	67/67	67/67
Xelnaga	68.31	39/67	18/67	16/67	22/67	23/67	-	13/67	56/67	31/67	49/67	66/67	67/67	56/67	62/67	66/67	66/67	61/67	67/67
UAlbertaBot	67.25	5/67	3/67	21/67	18/67	36/67	54/67	-	63/67	36/67	30/67	40/67	67/67	62/67	65/67	65/67	67/67	67/67	67/67
MaasCraft	59	3/67	17/67	52/67	26/67	4/67	11/67	4/67	-	32/67	33/67	55/67	60/67	52/67	67/67	62/67	67/67	60/67	67/67
MooseBot	50.13	2/67	0/67	2/67	21/67	1/67	36/67	31/67	35/67	-	15/67	27/67	65/67	35/67	66/67	50/67	53/67	65/67	67/67
BTHAI	46.8	0/67	0/67	7/67	3/67	0/67	18/67	37/67	34/67	52/67	-	33/67	38/67	47/67	65/67	34/67	35/67	64/67	66/67
TerranUAB	42.93	0/67	1/67	5/67	8/67	0/67	1/67	27/67	12/67	40/67	34/67	-	33/67	46/67	45/67	60/67	55/67	61/67	61/67
NUSBot	37.23	0/67	5/67	4/67	29/67	52/67	0/67	0/67	7/67	2/67	29/67	34/67	-	35/67	31/67	16/67	60/67	60/67	60/67
Nova	32.4	12/67	11/67	9/67	10/67	16/67	11/67	5/67	15/67	32/67	20/67	21/67	32/67	-	17/67	41/67	44/67	38/67	35/67
HITA	32.13	4/67	0/67	8/67	17/67	0/67	5/67	2/67	0/67	1/67	2/67	22/67	36/67	50/67	-	39/67	49/67	65/67	66/67
CruzBot	29.15	0/67	0/67	0/67	12/67	0/67	1/67	2/67	5/67	17/67	33/67	7/67	51/67	26/67	28/67	-	61/67	32/67	57/67
Bonjwa	16.51	0/67	0/67	4/67	0/67	0/67	1/67	0/67	0/67	14/67	32/67	12/67	7/67	23/67	18/67	6/67	-	43/67	28/67
Oritaka	14.4	0/67	1/67	0/67	4/67	0/67	6/67	0/67	7/67	2/67	3/67	6/67	7/67	29/67	2/67	35/67	24/67	-	38/67
Yarmouk	11.5	0/67	0/67	2/67	4/67	0/67	0/67	0/67	0/67	0/67	1/67	6/67	7/67	32/67	1/67	10/67	39/67	29/67	-

Figure 5.1: Bot versus bot results [D.14b]

Figure 5.1 shows detailed bot versus bot table. We can check how many games a bot from a particular row won against every other bot. For example our TerranUAB bot has lost all 67 games against the competition winner, IceBot, but won 61 of 67 games with Yarmouk. The table is ordered from the bot with highest to the lowest winrate. From this figure we can observe some anomalies. For instance the overall strong LetaBot won only 15 out of 67 games with much overall weaker opponent, MaasCraft. The similar situation can be seen for Skynet and NUSBot. As we explained in previous sections, such situations usually happen when a particular strategy meets its counter-strategy. While players adapt and adjust their strategies immediately to the opponent's, in case of bots it usually happen by accident. There is no perfect strategy and even the best ones will lose against their counter-strategies. This lack of adaptation seen in bots is one of the main reason why players are still much better than any AI in RTS games.



Bot Win Percentage By Map	Benzene	Destina	Heartbr	Aztec	TauCros	Androme	Circuit	Empireo	Fortres	Python
IceBot	88 %	89 %	84 %	87 %	89 %	86 %	82 %	84 %	85 %	79 %
Ximp	77 %	83 %	89 %	86 %	84 %	85 %	89 %	84 %	80 %	83 %
LetaBot	88 %	84 %	68 %	85 %	78 %	81 %	78 %	86 %	87 %	82 %
Aiur	70 %	71 %	78 %	69 %	68 %	68 %	71 %	67 %	73 %	70 %
Skynet	63 %	63 %	70 %	63 %	64 %	72 %	73 %	74 %	72 %	70 %
Xelnaga	68 %	65 %	69 %	64 %	68 %	70 %	65 %	67 %	71 %	70 %
UAlbertaBot	63 %	71 %	65 %	65 %	68 %	74 %	63 %	60 %	67 %	71 %
MaasCraft	55 %	59 %	61 %	59 %	57 %	63 %	52 %	61 %	58 %	60 %
MooseBot	52 %	47 %	42 %	45 %	56 %	44 %	51 %	57 %	53 %	50 %
BTHAI	52 %	52 %	57 %	40 %	43 %	37 %	52 %	39 %	40 %	51 %
TerranUAB	44 %	41 %	42 %	51 %	33 %	49 %	48 %	35 %	39 %	42 %
NUSBot	37 %	45 %	3 %	36 %	40 %	42 %	35 %	45 %	48 %	41 %
Nova	29 %	39 %	42 %	47 %	46 %	3 %	41 %	38 %	17 %	14 %
HITA	31 %	32 %	42 %	30 %	36 %	32 %	26 %	28 %	24 %	33 %
CruzBot	34 %	23 %	27 %	31 %	20 %	34 %	33 %	28 %	31 %	25 %
Bonjwa	21 %	13 %	21 %	12 %	15 %	18 %	13 %	12 %	18 %	17 %
Oritaka	12 %	10 %	13 %	13 %	20 %	19 %	10 %	13 %	12 %	17 %
Yarmouk	8 %	5 %	18 %	7 %	6 %	14 %	9 %	13 %	16 %	15 %

Figure 5.2: Bot win percentage by map [D.14b]

In Figure 5.2 we can see bot win rate by map. There were ten maps and in most cases the win rate of a particular bot show only slight differences across different maps. There are some anomalies though. NUSBot was almost always beaten on HeartbreakRidge map, similar to Nova on Andromeda. Even for our TerranUAB bot the difference between the highest (51% on Aztec) and lowest (33% on TauCross) win rate map was quite significant. Some of those differences come from lack of good terrain analysis or base planning, what leads to incorrect building placement on some maps or very good placement on the others. Another reason might be the size of the map. As we mentioned in previous sections, some strategies, such as the Zealot Rush, are very effective on small maps, but they are weaker the larger the map size. Again, professional players have the advantage over RTS AIs, because they can easily chose the right strategy for the right map. What is more, correct building placement is easily executed correctly even by the amateur players, regardless of the map, if they only know some general guidelines, such as "place the buildings near the ramp to block the enemy rush" . For the AI however, this task pose a significant challenge and even small differences between the two maps could affect the building placement algorithm greatly.

Group	Wins	Losses	Total	Win %
Protoss	182	421	603	30.18
Terran	262	207	469	55.86
Zerg	45	22	67	67.16
Protoss (veteran)	37	298	335	11.04
Terran (veteran)	80	121	201	39.80
Zerg (veteran)	0	0	0	N/A
Protoss (new)	145	123	268	54.10
Terran (new)	182	86	268	67.91
Zerg (new)	45	22	67	67.16
Total (veteran)	117	419	536	21.83
Total (new)	372	231	603	61.69
Total	489	650	1139	42.93

Table 5.5: TerranUAB statistics

Statistics for our TerranUAB bot are presented in Table 5.5. In each row there is information about TerranUAB's performance against a particular group of opponents. Our bot won 489 and lost 650 games in total. Protoss opponents were significantly (by 25.68%) harder to beat than those playing Terrans. This is observed in matches against both new and veteran opponents. The difference in performance against Protoss and Terrans is lesser (by $14,95\%$) in matches against new bots, however. It is difficult to estimate our bot's performance against the Zerg race, because there was only one Zerg bot in the competition. Our bot performed much better against the new bots than the veterans, achieving 61.69% win rate on average in matches against the former ones and only 21.83% on average against the latter.

From gathered data we conclude that in general our solution an effective one among the new bots but is surpassed by the older ones. It is also more effective against Terrans. This is surprising, because our main strategy is based on Vultures, which are very effective counter to Protoss Zealots. We suppose that it is not our strategy that is more effective against Terrans, but the Protoss bots that performed better on average in this competition.

5.3 CIG 2014 StarCraft AI Competition Results

In Table 5.6 we present results from the this year's CIG StarCraft Competition. There were thirteen participants. All except one, WOPR, competed also in AIIDE tournament. Because the tournament used the same software and set of rules as the AIIDE Competition, the results are also very similar. Due to similarities between the two tournaments and the lack of data about CIG tournament details, we restrict ourselves to present only the final results.

Bot Name	Race	Main Contributor	Institution	Win %
ICEBot	Terran	Kien Nguyen Quang	Ritsumeikan University	83.06
Ximp	Protoss	Tomas Vajda	Independent	78.06
LetaBot	Terran	Martin Rooijackers	Maastricht University	68.47
AIUR	Protoss	Florian Richoux	University of Nantes	66.11
UAlbertaBot2013	Protoss	David Churchill	University of Alberta	60
WOPR	Terran	Soren Klett	University of Bielefeld	56.53
MaasCraft	Protoss	Dennis Soemers	Maastricht University	55.14
NOVA	Terran	Alberto Uriarte	Drexel University	38.89
MooseBot	Protoss	Adam Montgomerie	University of Bristol	38.33
TerranUAB	Terran	Filip Bober	Poznan University of Technology	34.03
BTHAI	Terran	Johan Hagelback	Linnaeus University	31.53
NUSBot	Protoss	GU Zhan	Institute of Systems Science, NUS	21.53
CruzBot	Protoss	Daniel Montalvo	UC Santa Cruz	18.33

Table 5.6: CIG 2014 StarCraft AI Competition results

Again, ICEBot won the competition, followed by Ximp, LetaBot and AIUR. Our TerranUAB bot took 10th place. It is worth to note, that this time BTHAI was beaten by our bot and was placed just behind us. There is the opposite situation with NOVA, which was beaten by our bot in AIIDE but performed better in CIG. We do not know what is the reason for the observed differences.

5.4 Default AIs Results

To evaluate our solution against the default StarCraft AIs, we played three games, one for each race, on all of the AIIDE Competition maps. As the tournament was played across ten maps, we have thirty games in total. It proved to be enough to make conclusions about our bot performance.

As we can see in Fig. 5.7, our bot performed very well against the default StarCraft AIs. Out of thirty games, only three were lost. All the three defeats were due to the Wraith Rush strategy, which proved to be ineffective in a situation where the enemy fought off the early rush. It is worth to remember, that the default StarCraft AI is cheating.

How the default AI cheats was described by Shamus Young [You08]. In his experiment, he observed the performance of AI players on the same map with the same settings multiple times. Start locations were randomized and there were always two Protoss, two Terran and two Zerg players. The first thing he observed was that if the AI spends all its resources and bankrupts, it automatically gives itself 2000 minerals and gas. Another interesting fact was that the AI's games are very predictable. They have kind of rhythm; at first, each AI build builds an initial force, then attacks a random enemy, then it waits until this force is dead, and then builds defenses and another attack force and so on, wave after wave. If two enemy forces meet in the third's player base, they will wipe out each other, leaving the buildings alone.

Shamus Young expected that Zerg race would work best "in hands" of the default AI, but his experiment shown that it were Protoss which faired best, while the Terrans are the worst, even though in professional players community they are all considered well balanced (proved by StarCraft [kes12] and StarCraft II [wcs14] rankings). He was intrigued by the fact and observed, that default AI is extremely poor at managing Terran units and considerably better with Protoss.

In our experiment, however, we have lost one game out of ten against each race and Protoss AI has not proved better than the others. The advantage of the default Protoss AI might have been compensated by our main strategy with Vultures, which are extremely effective against Protoss Zealots, commonly used by that AI.

In our experiment we had no time limit. Two games were likely to last forever, because the enemy was almost completely destroyed but our bot could not find this last building and was running in circles. We restarted the match in those cases and recorded the latter result.

Map	Result	TerranUAB Score	Enemy Score	Enemy Race
Benzene	Victory	54200	27830	Protoss
Benzene	Victory	58982	36499	Protoss
Benzene	Victory	107638	65528	Protoss
Destination	Victory	16435	2838	Protoss
Destination	Victory	43573	25452	Terran
Destination	Victory	91097	50765	Zerg
HeartBreakRidge	Victory	25351	7481	Protoss
HeartBreakRidge	Victory	47032	27050	Terran
HeartBreakRidge	Victory	74052	47216	Zerg
Aztec	Victory	67674	41205	Protoss
Aztec	Defeat	38087	84962	Terran
Aztec	Victory	66401	33158	Zerg
TauCross	Victory	83903	51314	Protoss
TauCross	Victory	30827	15473	Terran
TauCross	Victory	61473	30347	Zerg
Andromeda	Victory	76990	44379	Protoss
Andromeda	Victory	68585	42875	Terran
Andromeda	Victory	94347	60058	Zerg
CircuitBreaker	Defeat	136395	192675	Protoss
CircuitBreaker	Victory	65953	41320	Terran
CircuitBreaker	Victory	92229	53028	Zerg
EmpireoftheSun	Victory	72130	33248	Protoss
EmpireoftheSun	Victory	29138	14737	Terran
EmpireoftheSun	Victory	13298	4940	Zerg
Fortress	Victory	49686	26885	Protoss
Fortress	Victory	27884	14956	Terran
Fortress	Defeat	12919	25986	Zerg
Python	Victory	95350	58019	Protoss
Python	Victory	55726	23960	Terran
Python	Victory	77063	33065	Zerg

Table 5.7: Results with the default StarCraft AI

6 Summary

6.1 Conclusions

The field of RTS games AI is a complex one. There are many different problems related to this area that must be solved, before the AI can get even close to the human effectiveness. Even though current bots still struggle in comparison with human players, the RTS games AI field of research is becoming more and more popular and every year we observe new fascinating advances. In recent years there has been a significant increase in the number of papers related to this field. The progress in this area is very important, because many of the problems encountered in developing an AI for an RTS game are very similar to the obstacles of other research areas and real-life problems as well. RTS games provide an excellent environment to test the proposed solutions.

In this work, we managed to implement a fully functional and effective Terran bot for StarCraft. Our solution is based on the existing architecture of a Protoss bot called UAlbertaBot. We participated in two largest StarCraft AI tournaments: "2014 AIIDE StarCraft AI Competition" and "CIG 2014 StarCraft AI Competition". We placed 11th out of 18 and 10th out of 13, respectively, which is a good score for a first attempt. Our solution achieved 90% win rate against the default StarCraft AI.

6.2 Future Work

The bot we implemented proved to be effective. Nevertheless it has many weaknesses and there is much to improve. First of all, we implemented only a few strategies, which restricts our options to execute good counter-strategies and enables our opponent to counter us easily. Just implementing more strategies would not be enough, though. There should be some algorithm to choose the right strategy against a particular opponent. Algorithms from the machine learning field of research might provide answer to this problem.

Another thing that might be improved are the opening build orders. We implemented over a dozen of those, based on an expert knowledge. However, because opening build orders are a sequence of actions, an genetic algorithm might be able to find many better ones.

We implemented AI for Terrans. Base UAlbertaBot plays Protoss. There is still lack of the implementation for the Zerg race, the least popular among the tournament participants.

One of the main problems of all current RTS games AI solutions is lack of adaptation. This is a trait that human players excel at; they change and adjust their strategies to surprise the enemy and counter his actions. This is also something that bots struggle with; they are predictable for the human opponents, easily deceived and lured into traps. Maybe analyzing human players behaviors would result in some progress in this area. Bayesian models suggested by many researchers might be a step in the right directions.

The adaptation in the scope of StarCraft might be later extended to more general solutions, that could adapt to any kind of RTS game and then, maybe, to any kind of computer game or even real-life situation as well.

A
BWAPI

The Brood War Application Programming Interface (BWAPI) is a C++ framework used to interact with StarCraft: Broodwar by Blizzard Entertainment [bli14]. BWAPI provides an interface to control individual units, read the game state and adjust the game settings.

The appearance of BWAPI was one of the main reasons why StarCraft became a test-bed for AI methods. Without the provided interface it would be hard to implement functional StarCraft bot and every researcher would need to put much effort into developing his own input and output methods. Every CIG and AIIDE competitions participant implemented his AI upon the BWAPI library.

BWAPI library is completely free and open-source [bwa14]. It requires, however, a copy of StarCraft to work. As there has not been StarCraft code published and most likely never will, BWAPI relies on a technique called code caving, commonly used by hackers. In short, this works by replacing original, running StarCraft executable instructions at the appropriate places in the memory, with 'goto' commands. This way, BWAPI can read and write data to the StarCraft process memory via pointers.

Because manipulating such pointers would still be uncomfortable for AI programmers, BWAPI provides a number of classes and methods to effectively manipulate StarCraft data. Every action that a player could do is available through BWAPI. For instance if a player wanted to move Terran Marine to a specific place on the map, he would click on that location with the Marine selected. With BWAPI we would just use move method of that particular unit, providing X and Y coordinates. BWAPI does not provide any data processing algorithms, however. The only thing it does, is sending our input to StarCraft and providing us with the output (game state).

The framework is still under development. The current version is 4.0.1 Beta, but for our bot we used version 3.7.4 . We made the decision to use older version because UAlbertaBot is compatible with version 3.7.4 and we wanted to have a working bot to extend. This year's CIG and AIIDE tournaments permitted both mentioned BWAPI versions but most participants used the older one (only two of eighteen AIIDE participants used the new version).

B Technical Issues

B.1 Missing Units

As explained in section 3.4.5, every unit, research or technology is an action. Actions are very important for UAlbertaBot and are used in many parts of the code. To maximize the efficiency on a lower level the set of actions is implemented as a 64-bit sized bitset.

Every unit, tech or building is represented by a number it occupies in a vector stored inside the StarcraftData class. For the unit to be accessible and recognized by other modules it must be added to this before. The vector is hard-coded and is constructed at the beginning of the game, when our race is known. As each race has different units, the vector is different for each race.

By default there are 30 actions available for the Protoss but only 12 for Terrans. It may be enough to implement a particular strategy but we wanted our solution to be more flexible. To get access to every unit, building, tech or upgrade we needed 50 actions. It turned out that actions with a number greater that 31 are simply ignored. That meant we were not able to access some actions at all. Some technologies could not be researched, units could not be trained and buildings could not be constructed.

We found that error lies deep in the bitset class. It was supposed to store and operate on a 64-bit number but due to some internal errors the number was only 32-bit. The class implemented number of bitset operations where majority worked properly only on a 32-bit number. It limited actions vector size to the size of only 32.

As we needed much more we have rewritten the whole class. We changed the set container from long long int type to std::bitset and rewritten all the methods to works properly. To achieve backward compatibility methods names remained the same. There were also some global macros defined which we have rewritten too.

C
CD Content

The CD attached to this thesis contains:

- TerranUAB bot source code.
- TerranUAB DLL library.
- Detailed 2014 AIIDE StarCraft AI Competition results.
- An electronic version of this work.

Bibliography

[aii14] AIIDE StarCraft Competition Files starcraft aiide tournament website. [on-line] `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/media.shtml`, 2014.

[AP94] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. Artificial Intelligence Communications, vol. 7, no. 1, pp. 39–59, 1994.

[BC12] M. Buro and D. Churchill. Real-time strategy game competitions. AIMagazine, vol. 33, no. 3, pp. 106–108, 2012.

[BK07] M. Buro and A. Kovarsky. Concurrent action selection with shared fluents. AAAI Vancouver, Canada, 2007.

[BL10] A. Branquinho. and C. Lopes. Planning for resource production in real-time strategy games based on partial order planning, search and learning. In Systems Man and Cybernetics (SMC). [on-line] `http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?reload=true&arnumber=5642498`, 2010.

[bli14] blizzard entertainment. [on-line] `http://eu.blizzard.com`, 2014.

[Bur03] M. Buro. Real-time strategy games: A new ai research challenge. International Joint Conferences on Artificial Intelligence, pp. 1534–1535, 2003.

[bwa14] BWAPI github. [on-line] `https://github.com/bwapi/bwapi`, 2014.

[CB11] David Churchill and Michael Buro. Build Order Optimization in StarCraft. [on-line] `http://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/viewFile/4078/4407`, 2011.

[com14] StarCraft AI Competition Rules starcraft ai competition. [on-line] `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/rules.shtml`, 2014.

[D.14a] Churchill D. AIIDE 2012 Tournament Results starcraft ai competition. `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/2012_official_results/`, 2014.

[D.14b] Churchill D. AIIDE 2014 Tournament Results starcraft ai competition. `http://webdocs.cs.ualberta.ca/~scbw/2014/`, 2014.

[D.14c] Churchill D. SparCraft google code. `https://code.google.com/p/sparcraft/wiki/SparCraft`, 2014.

[D.14d] Churchill D. UAlbertaBot david churchill computing science, university of alberta. `http://webdocs.cs.ualberta.ca/~cdavid/ualbertabot.shtml`, 2014.

[Dot13] The Daily Dot. Sounth Korean 'StarCraft' star becomes the highest-earning player in eSports.
http://www.dailydot.com/esports/jaedong-starcraft-highest-earning-player/, 2013. [Online; accessed 2014-09-11].

[dun14] ggk.gildia.pl. [on-line]
http://ggk.gildia.pl/wp-content/uploads/2014/01/feat-Dune-2.jpg, 2014.

[DWAP05] M. Molineaux D. W. Aha and M. J. V. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. ICCBR, 2005, pp. 5–20, 2005.

[Ear14] Esports Earnings. Player Earnings. http://www.esportsearnings.com/players, 2014. [Online; accessed 2014-09-11].

[EKG12] O. Cappe E. Kaufmann and A. Garivier. On Bayesian Upper Confidence Bounds for Bandit Problems. [on-line] http://machinelearning.wustl.edu/mlpapers/paper_files/AISTATS2012_KaufmannCG12.pdf, 2012.

[fir09] StarCraft AI Competition expressive intelligence studio. [on-line]
http://eis.ucsc.edu/StarCraftAICompetition, 2009.

[FSS07] S. Bakkes F. Schadd and P. Spronck. Opponent modeling in real-time strategy games. GAMEON, 2007, pp. 61–70, 2007.

[HF03] R. Houlette and D. Fu. A data mining approach to strategy prediction. The ultimate guide to fsms in games, AI GameProgramming Wisdom 2, 2003.

[HO13] Søren Tranberg Hanse and Santiago Ontanon. A Software Framework for Multi Player Robot Games. [on-line] http://vbn.aau.dk/files/45925438/A_Software_Framework_for_Multi_Player_Robot_Games.pdf, 2013.

[Hol13] K. E. Holsinger. Decision Making Under Uncertainty: Statistical Decision Theory. [on-line]
http://darwin.eeb.uconn.edu/eeb310/lecture-notes/decision/decision.html, 2013.

[HS08] J.-L. Hsieh and C.-T. Sun. Building a player strategy model by analyzing replays of real-time strategy games. IJCNN, 2008, pp. 3106–3111, 2008.

[JTY10] N. Beume S. Wessing J. Hagelback J. Togelius, M. Preuss and G. N. Yannakakis. Multiobjective Exploration of the StarCraft Map Space. 2010 IEEE Conference on Computational Intelligence and Games, 2010.

[kes12] Korean e-Sports Association ranking wikipedia.org. [on-line]
http://en.wikipedia.org/wiki/StarCraft:_Brood_War_professional_competition#KeSPA_rankings, 2012.

[MTTM09] K. Khalili Damghani M. T. Taghavifard and R. Tavakkoli Moghaddam. Decision Making Under Uncertain and Risky Situations. [on-line]
http://www.ermsymposium.org/2009/pdf/2009-damghani-decision.pdf, 2009.

[MW09] M. Mateas and B. G. Weber. A data mining approach to strategy prediction. IEEE Symposium on Computational Intelligence and Games (CIG), 2009.

[PAF02] N. Cesa-Bianchi P. Auter and P. Fischer. Finite-time Analysis of the Multiarmed BanditProblem. [on-line]
http://link.springer.com/article/10.1023%2FA%3A1013689704352, 2002.

[PG03] S-H. Poon and C. W. J. Granger. Forecasting Volatility in Financial Markets: A Review. Journal of Economic Literature Vol. XLI (June 2003) pp. 478–539, 2003.

[PKJHA] William M. P. Klein Paul K. J. Han and Neeraj K. Arora.

[Pow08] M. Power. Organized Uncertainty: Designing a World of Risk Management. [on-line] http://ideas.repec.org/b/oxp/obooks/9780199548804.html, 2008.

[SB11a] G. Synnaeve and P. Bessiere. A Bayesian Model for Opening Prediction in RTS Games with Application to StarCraft. Computational Intelligence and Games (CIG), 2011 IEEE pp. 281-288, 2011.

[SB11b] G. Synnaeve and P. Bessiere. A Bayesian model for RTS units control applied to StarCraft. Computational Intelligence and Games (CIG), 2011 IEEE pp. 190-196, 2011.

[SO13] A. Uriarte F. Richoux D. Churchill M. Preuss S. Ontanon, G. Synnaeve. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. [on-line] `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6637024`, 2013.

[sta08] StarCraft's 10-Year Anniversary: A Retrospective web archive. `http://www.dailydot.com/esports/jaedong-starcraft-highest-earning-player/`, 2008.

[Syn12] G. Synnaeve. Bayesian Programming and Learning for Multi-Player Video Games. Ph.D. dissertation, Universite de Grenoble, 2012.

[tou14] Tournament Manager Software starcraft ai competition. [on-line] `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/tm.shtml`, 2014.

[UJA11] H. Munoz-Avila U. Jaidee and D. W. Aha. Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. Proceedings of the ICCBR Workshop on Computer Games, pp. 43–52, 2011.

[wcs14] WCS 2014 Global Standings battle.net. [on-line] `http://wcs.battle.net/sc2/en/standings/2014`, 2014.

[You08] Shamus Young. StarCraft: Bot Fight shamusyoung. [on-line] `http://www.shamusyoung.com/twentysidedtale/?p=1597`, 2008.