Developing a StarCraft: Brood War Agent

Carsten Nielsen

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of the thesis is to ...

Summary (Danish)

Målet for denne afhandling er at ...

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring an B.Sc. in Engineering.

Starcraft is one of the strongest game franchise ever created. Professional players compete in Starcraft 2 tournaments with large price purses. Perhaps less known is that former developers of Othello and chess playing programs have taken on the task to develop AIs for the game. Most developers use the relatively old Starcraft Brood War as all the necessary infrastructure is readily available: `http://code.google.com/p/bwapi/`. Championships for AIs, bots, are arranged yearly, see for example `http://www.sscaitournament.com/` and `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/`. In addition, ladder systems are also available, see `http://bots-stats.krasi0.com/`.

The project focuses on developing a new bot and is aimed at 20 ECTS.

The project involves:

- studying and implementing AI methods for handling large search spaces.
- developing high-level strategy reasoning algorithms and heuristics.
- solving unstructured optimization problems.

Lyngby, 30-June-2015

Not Real

Carsten Nielsen

Acknowledgements

I would like to thank my....

Contents

Chapter 1

Introduction

Artificial Intelligence in *real-time strategy* (RTS) games offer the challenge of limitless decision space, incomplete world information and varied strategies in a shifting *metagame*. A computer controlled player, called a *bot*, must both be efficient and effective. Compared to other games, RTS focuses on a single agent.

StarCraft: Brood War, released by Blizzard in 1998, is one such RTS game that has been the focus of research. In addition to usual RTS elements, StarCraft is asymmetric with three different factions to play as and slightly asymmetric maps. Since the advent of the unofficial *Brood War API* (BWAPI), people have been able to develop their own bots for the game, pitting them against each other and spawning a few tournaments. The tournament scene is still maintained even though both the game and API have become dated.

This project will both focus on creating a competitive bot by following successful contemporary bots, and investigate how prevailing bots are designed and how they overcome the aforementioned challenges. The bot will be spar against other bots in the SSCAI (Student StarCraft AI) tournament, possibly competing in the 2015 tournament (depending on when it starts).

Original Project Plan

Original project plan.

Revised Project Plan

Revised project plan.

Brief self-evaluation.

Chapter 2

Project Process

The project was developed with the agile development process, but no particular agile method was used. Design and implementation was iterated upon across short week-long sprints. After each there was an evaluation of the agent's current status and new goals would be identified for the next sprint. The project plan took form as a backlog, kept during development, and was used when determining the status and goals in scope of the whole time-plan. There were a few milestones throughout development, usually a month apart. They determined some requirements the agent should satisfy at specific points in the development, with regards to its performance. The milestones served as a medium between sprints and the overall goal, ensuring the bot improved consistently, and that a satisfactory end-result would be met.

Testing was done in two steps. First the robustness of the implementations was tested by playing the bot with random built-in StarCraft bots, across all maps used in the competitions. This was to ensure the stability of the bot and that the implementations worked as required, across all the maps it would play on. The second step was uploading the bot to the SCCAIT tournament page, where the bot would play against other bots. This would serve to test the strategic integrity of the bot, and was done rarer than the first step in order to retrieve enough statistical data. This was much slower as the matches were tested in real-time, but necessary to ensure the opponents were up-to-date. The opponents on the SSCAIT website should be as recent since the last tournament (January, 2015), and would therefore be the best candidates to test against the agent's abilities.

Because it is time-consuming to retrieve data from rounds on the SCCAIT, the first step was needed to determine crash issues and such technicalities, in order to not waste time.

2.1 Project Plan

The project started in mid-January and development ceased early June, with four milestones in between. We will briefly explain them:

The first milestone were attaining the minimum viable product by the end of February. The requirements for this was an agent capable of defeating a passive opponent, implying means to gather resources, build units, scouting and attacking. This would imply the agent had a non-zero chance at winning a match.

Second milestone was generalizing and polishing the first solutions at the end of March, executing the agent's strategy more efficiently. This included efficient resource gathering and production, simple combat prediction and information manager of enemy units.

By late April, the third milestone required more strategic elements such as base-defense, better unit grouping with retreating and expansions.

The final milestone by late May to mid-Jun required the use of advanced units and technology upgrades, with possible features such as improved expansion, scout harassment or enemy economy disruption. It was somewhat open-ended, as it was unknown which features would be most needed at the time.

The first two milestones were completed with all required features. By the third milestone however, the expansion feature proved much more time-consuming to implement than predicted. Meanwhile, preliminary results with advanced units suggested a lot of work was required to make them more efficient than just using basic units. For those reasons, advanced units as a feature was scrapped in favor of completing the expansion feature. As with advanced units, expansions were only worthwhile if they were efficient, which required a generalized implementation, better expansion locations and the Maynard slide technique. As a result, the last milestone goals were not satisfied, as improved expansion had been implemented, but the rest was not. On the other hand, in the final version of the agent, all the features had been implemented with good gameplay performance, few and minor bugs and with no reported crashes.

The final features probably took longer than predicted, since they had to be at least as efficient as the rest of the agent's features to be worthwhile. It was easy to implement a simple version of the features, but gameplay performance demanded the consideration of many edge-cases to be efficient. Implementing a single feature well would have a better chance of increasing win-rate.

Chapter 3

StarCraft

StarCraft is a *real-time strategy* (RTS) game released by Blizzard Entertainment in 1998. Controlling structures and troops, players seek to expand their economy, technology and army until they can eliminate all opponents. Players take the role as one of three different races: *Terran*, *Zerg* or *Protoss*. These determine which buildings and troops are available to the players, and while mechanically similar they are very different strategically. The expansion *StarCraft: Brood War*, also released in 1998, increased the rooster of troops available among other things.

Players interact with game through *units* which take the form of troops and buildings. These can be given *commands* by players, such as move, attack or build. Buildings can be commanded as well, capable of producing additional non-building units or upgrade to new technologies. Resources must be acquired to build structures, train troops or upgrade technologies. Resources are placed around on the map, and must be gathered by worker units, the only units capable of doing so.

The map is covered in the *fog-of-war* which limits players' visibility to only their own units and their surroundings. Everything within the fog-of-war is hidden, including opponents' units within.

In multi-player matches, each player starts with a single structure and a few worker units. Each unit is identified with a player color, controllable by only that player. The goals is then to destroy all the opponents' structures, which is done by amassing an army and attacking. Each match is played on one of many *maps*, which determines the terrain, resource and start locations. These factors have great impact on the pacing and player strategies. With a large amount of different unit- and building types available, the players compete by using the right types to counter their opponents' units. Players must adapt to the battlefield situation, as there is always a counter-strategy to every strategy.

Chapter 4

Bots in StarCraft

Before delving into the agent's design, this chapter will cover a few other bots, their solutions and the competitions in which they play. First we will explain the basics of strategies in StarCraft.

In most RTS games, StarCraft being no exception, there is a trinity of strategies in a rock-paper-scissor formation. These are the *rush*, *boom* and *turtle* strategies.

The *rush* strategy is when the player attempts to produce and attack with troops as fast as possible. There are many variations in StarCraft, where the most aggressive are called *all-in*, as the rushing player will lose shortly afterwards if the initial rush fails. This could happen if the opponent builds stationary defenses, which are usually much more cost-effective than the mobile units required to attack.

Closely related, a player could instead focus on developing economy or technology to gain advantage against slow strategies. This is usually called a *boom* strategy, which in StarCraft involves gaining an *economic advantage*. If successful, the player will attain superior units and easily replace lost ones, winning either through strength or attrition. The weakness is early game defense will have to be sacrificed which puts the boom player at risk to enemy rush strategies.

Finally, the defensive strategy called a *turtle* or *turtling* builds strong defenses early on. As mentioned, this will counter an enemy rush by costing fewer resources, used to expand in economy or technology. The resulting advantage is then used to win.

This is a slow strategy however, which fails against the riskier *boom* strategy. Since the defensive strategy does not put pressure on the opponent, they are free to completely spend resources on expanding to stronger units.

Matches in most RTS games can be divided into three stages: *early-*, *mid-* and *late-game*. These are somewhat vague definitions of time intervals in the game where different strategies apply. There is no match timer in StarCraft, so the stages are usually determined by the game-state, and therefore they arrive at different times across matches. Mid-game is when players have established their base and economy and usually built more bases. Late-game is once most of the map has been settled and all units are available to players. The three basic strategies we covered are used during the early game, but has strong influence on mid- and late-game.

4.1 Bots

Computer-controlled players in games are called *bots*. The main difference between agent and bot is that bots are always a replacement for players, and they are usually limited by the same rules as human players. The term is mostly used in first-person shooters, but can be extended to all video games. Both bot and agent is used in this paper to refer to AI-controlled StarCraft players.

Compared to other games however, even the best bots in RTS games are usually worse than competitive human players. This is a result of the difficult challenges bots must overcome in the games, particular the huge decision space. At every frame, a player is capable of giving one or all of their units new commands, most of which require a target unit or location. Couple a 55 ms. limit in the SSCAIT and similar in other tournaments, bots do not have very long to perform calculations on this decision space. While the same issue lies in Chess turns, there are many more frames in a typical StarCraft match, which is usually among the ten-thousands.

It is practically not possible to use any standard AI solution that scales with the decision space, like alpha-beta pruning, without massive simplifications. Bots usually resort to having a few, very rigid move-sets, with some hard coded adaption to usual enemy strategies.

Few bots ever reach mid-game or later as matches are often determined beforehand. As the game progresses to later stages, the races gain access to many kinds of units, which are difficult to control well for a bot. Most bots therefore focus on the early-game where only a few units are available, unable to perform well at later stages. The most advanced bots, capable of playing the later stages well, forces the match into mid-game by turtling, winning against the simpler bots. Notable examples of this strategy are Letabot and XIMP bot.

The following sections outline some of the challenges in StarCraft bot development and some solutions.

Incomplete Information

As the map is partially shrouded in the fog-of-war, players are left to guess what the opponents' are scheming. While humans are pretty good at this, it becomes troublesome when bots have to model the movement of opponents. The longer they are left unchecked, the more possible game-states are there to account for. Scouting opponents is therefore imperative, as this will reduce the amount of prediction needed.

Beyond the fog-of-war, upgrades are impossible to discover unless their effects are observed. Experienced players can often guess the state of opponents' research based on their strategy, and in some cases have memorized some unit match-ups upgraded and otherwise. The accuracy of combat predictions rely on this information, which are already difficult to do accurate without factoring in upgrades. Opponents' resources are also hidden from players.

Controlling Multiple Units

There is a lot more to RTS combat than commanding unit A to attack B. Positioning alone is a huge factor, which is completely dependent on the positioning of all other units in the area. The terrain of the map can be advantageous or otherwise. There are different ways to prioritize targets, none of which are conclusively superior than others. Humans have the advantage by easily reading the graphical display of RTS games, where the bot must use other methods.

One of the solutions practiced by ICEbot is using potential flows to direct units' positioning. A similar solution is also used when scouting in order to avoid enemy troops. Potential fields is used by the Berkeley Overmind for guiding its air-units around threats, and heatmaps have been used to detect army compositions.

Strategic Planning

To create more adaptable bots, there has been several attempts to use some dynamic decision making solutions. Once the opponents' game-state is known, it is possible to infer their strategy and future actions, which can be used to react and prepare a counter-strategy.

There has been multiple studies about using Bayesian models for planning and prediction.

Chapter 5

Agent Design

Developing a bot requires some forethought, even in agile development. The base of the bot must be both robust and versatile in order to accept future features. In this chapter the agents' gameplay strategy will be described as well as its architecture and implementation design.

Before any strategies for mid-game or late-game can be devised, the bot must first master early-game. If the bot loses in early-game, the match will never proceed to later stages. Compared to matches between humans, the game usually ends much quicker, rarely leaving early-game. This is because later stages of the game involves many more kinds of units and strategies, such that the AI must be more advanced. Therefore, it seems prudent to first perfect the early game of the bot. From there, if time allows, the bot could attempt mid-game tactics.

Simplest solution is implementing one strategy, which can be expanded into mid-game if the round drags on that long. Both boom and turtle strategies seek to push the match out of early game, so the remaining strategy which lies in early game is rush. This has seen a lot of use in the bot tournaments, probably being the most popular choice. It requires nothing but building the earliest combat unit and attacking. When moving on to mid-game, additions could include upgrading, expanding to new resources or moving on to more advanced units. Even a failed rush would put pressure on the opponent, allowing the bot to compete in later stages of the match. This makes such a strategy sustainable in case the bot reaches mid-game, but is also proven to be a strong early-game strategy.

Bots focus on a single race, as there is no advantage the capability to play more. In this case, this bot will play Protoss. Even though Protoss are the slowest of the three to produce the first troops, they have the strongest and easiest to use early troops. This makes it a strong rush candidate, as it can easily beat opposing races' rushes if prepared. While the main AI challenges between races are shared, gameplay elements differ quite a lot.

5.1 Architecture Design

The imperative when designing the bot is to lessen the decision space as greatly as possible. The problems the bot faces are easily divided into smaller, isolated problems. Its therefore possible to construct the bot out of individual *modules*, each solving some subset of problems. This isolation lessens the amount of concurrent decisions and information available to the agent, reducing the decision space.

These modules are structured in a *hierarchy.* A superior module can command a subordinate, however only in terms of a *black box*, without knowledge of its internal structure. By limiting the information available to the superior, we can easily limit the decision space with clever design. The clear benefit of this structure is easy replacement of individual modules, without damaging or refactoring neighbors. On the other hand, modules will inherently be limited with information, but we must trust that not all available data is required or even relevant for all decisions.

The UAlberta bot by Dave Churchill uses such a module structure, however it places them in an interesting hierarchy. The modules in the bot are in a *arborescence* graph structure, that is, there is a root module which has exactly one path to each other node in the hierarchy. In other words there are no cyclic dependencies, and it is related to a tree structure, except a single module can have multiple parents. Churchill alleges that it is based on "proven military structures" - in any case the UAlberta bot is high-ranking bot, victor of one competition and runner up in others. The data-structure must have been proven to work by now.

The benefits of this structure is lesser and easier dependencies, removing the inherent challenges in cyclic dependencies. It enhances the benefits of modular design, since modules has a stricter position in the hierarchy, making the modular design easier. It is also a great boon to agile development, as the tree will simply evolve upwards with newer modules as superiors to the older. The lower modules make smaller decisions with fewer resource such as building and gathering, and the higher modules control more information to make the larger scale decisions such as strategies and attacking.

On the other hand, the structure allows less complicated interactions. By limiting

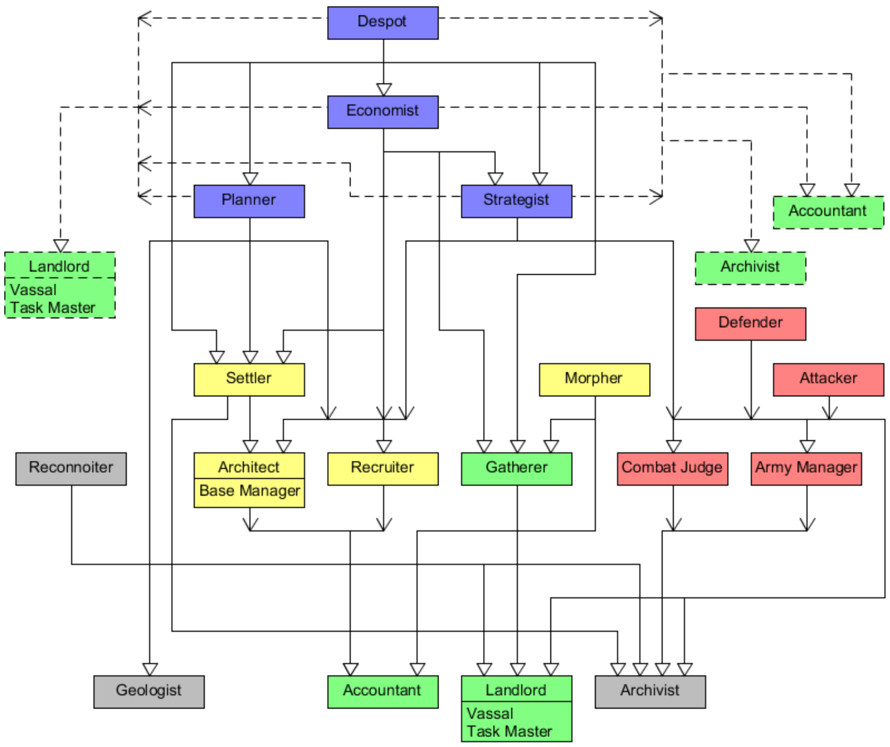

Figure 5.1: Diagram of the module hierarchy.

the hierarchy, the information available is limited as some modules must be at the bottom of the hierarchy. These are inevitably void of interactions with higher modules. This is also felt at the root of the structure, as at some point all the choices have to be made in the final module.

Figure 5.1 depicts the hierarchy structure, where higher modules are dependent on lower, shown by the arrows. The dependency is a one-way communication, such that the superior can query and operate its subordinates, but they are oblivious to their superiors. Modules which has direct subordinates only they are superior to are shown in the same box for simplicity.

The modules are separated in categories of interests, where likewise colored modules handle similar objectives. In the following chapters, we will explain the modules in the hierarchy and the underlying problems they solve.

- Chapter 6 (green) explains the concepts of resources and workers in StarCraft and how they are managed.

- Then, Chapter 7 (yellow) covers training and building units, which requires the resource modules.
- Information and its management is the subject of Chapter 8 (gray), which also covers scouting,
- which attacking, defending and general combat, detailed in Chapter 9 (red), is dependent on.
- At the root of the hierarchy is economy and strategy, which is described in Chapter 10 (blue).

Finally, Chapter 11 presents and discusses the results of the bot and the project.

5.2 Implementation Details

BWAPI is an API for StarCraft Broodwar and is injected upon startup by *ChaosLauncher*. It loads an AI module written in C++, which can retrieve information about the current match's map status and send commands to the game through BWAPI. This controls the player's actions and allows the development of bots for StarCraft.

Depending on the settings in BWAPI, the agent can be disallowed to retrieve information a player would not be able to. This means some units are in some levels of accessibility, where invisible and destroyed units are completely inaccessible. This has some limits however, as the agent can retrieve information about burrowed and cloaked units as if they were not. While a keen player can spot these units as they are not completely transparent, the agent has no limits as if they were completely visible, giving it an advantage. Additionally the agent is not limited to what is visible currently on the display, but can retrieve information from all over the map.

There is a popular library, the *Broodwar Terrain Analyzer* or *BWTA* for short, which pre-processes maps for locations of interest. Most importantly, it marks the optimal depot locations for gathering resources. While BWAPI already does this for start-locations, BWTA also does this for all resource clusters, marking viable expansion locations. The pre-processing of the map only has to be done once as the results are stored for subsequent games on the map. This library is automatically included in the 3.7.4 BWAPI test bot, so it is probable that almost all bots use the library.

While newer versions of BWAPI exist, the 3.7.4 version is still widely used as it is considered the most stable, and is compatible with BWTA. There does exist a BWTA clone for 4th versions, however it is not very stable. The downside to 3.7.4 is that it does not support C++11, so C++98 had to be used. There does not exist an equivalent to BWAPI for StarCraft 2, since technically BWAPI is a hack and using it would result in a ban. AI's have been scripted in StarCraft 2's own editor however, and some have even gone and interpreted the display output of the game for a bot.

Programming Details

Each module is its own class, where subordinates are given as arguments upon instancing. Beyond the root module of the hierarchy, there is the actual AI module called *Primary*, which is the one loaded into StarCraft. It receives all the callbacks from BWAPI, delivering them to the appropriate modules. It is also responsible for creating the hierarchy upon construction and updates it each frame. Modules are updated from bottom to top, such that subordinate dependencies are current at all times.

Almost everything a player is capable of interacting with through the game interface is handled as a simple method by BWAPI. Commanding units for example is simply calling the appropriate method on the unit-pointer. A more generic technique is used here however, where a unit, a target location and/or unit and a command type is given as arguments instead. This is to generalize the act of commanding, which requires a fair few checks such as whether the unit already has received a command this frame.

A simple version of the hierarchy was designed initially in the project, but later iterations were done with no design beyond the current feature(s) worked on. Every module has been refactored multiple times as a result of the agile development, especially those lower in the hierarchy such as the Landlord.

Some C++ STLs were used for their data-structures or writing to a stream. *BOOST* libraries were used for their macro for-loops, capable of some C++11 functionalities lacking in C++98. Use of dynamic memory was avoided as much as possible because of the risks of memory-mismanagement. Only the *Vassals* from Chapter 6 use them.

Every method has a brief comment describing its input, return and possible modifications to the object. Almost every scope beyond one line has a single line summarizing the current scope, since some methods can get quite large.

Chapter 6

Workers and Resources

This chapter covers the related to gathering resources, and how we handle this with the agent including the worker management architecture.

In early- and mid-game, *economic advantage* is the most important aspect of StarCraft strategy. This occurs when you have a higher resource intake than your opponent, such as if you have more workers that are gathering or more bases to gather from. The advantage is while maintaining an equivalent army to the opponent, one can replace lost troops faster, gain a larger army or upgrade the current one. If you eclipse your opponent in resources, you can perform worse in combat and still win. Without the advantage, upgrading troops would slow troop production down and the opponent could produce an equivalent or even stronger army at any time.

There are three kinds resources in StarCraft: *minerals*, *gas* and *supply*. *Workers* are the only units capable of gathering resources. Each race has one worker type and each player start the game with a couple of their race's type. Players also start with a *resource depot* or simply *depot*. This structure can produce new workers and is also the drop off point for gathering resources. The Protoss worker is the *probe* and their depot is the *nexus*.

Minerals are mined from mineral fields which usually are in clusters of 8-12. All units cost minerals, so it is the most important resource. Workers that mine these bring them back to the depot in batches of six mineral units. Depot's are limited in their proximity to the clusters, but at optimal distance there can be a maximum of three workers on each field. Mineral fields are usually placed in a half-circle formation, such that there exists an optimal depot location. While the first worker on each mineral field will gather at a linear rate, they will yield diminishing returns on the second and especially third worker.

Vespene Gas or simply *gas* is harvested from *refineries*. Each race has a distinct refinery structure, although they are very similar. These must be built upon *Vespene Geysers* of which there are up to two by each mineral cluster. It is harvested in batches of eight with a max of three workers per refinery at optimal depot distance. Advanced structures, units and all technologies depend on gas. The immediate cost of the refinery and lost mineral gathering is a liability against fast openings, but harvesting too late means fighting against stronger units.

The last resource, *supply* is a population limit and is not gathered like the other resources. Each unit reserves some amount of supply which is released upon their destruction. The only way to increase the supply limit is producing the race's supply unit. Each race has exactly one unit which increases the limit by a constant amount per instance, which is subtracted if they are destroyed. Units cannot be produced if they require more supply than the player has free. If the player happens to use more supply than their limit, they will be unable to build more units until more supply is acquired, called *supply blocked*. Although supply is specifically the Terran resource, it is used as the general term for all races. *Psi* and *control* are the Protoss and Zerg equivalents. The Protoss supply unit is the *pylon* structure.

While multiple workers can gather from the same field or refinery, only one worker can actively occupy it. The rest is either returning cargo, moving to the resource or waiting.

In the next sections we will first describe the architecture behind worker management, followed by how minerals is mined and gas is harvested. Building supply is first covered in Chapter 10, as this resources behaves much different than the others.

6.1 Managing Workers

The *Task Master* module contains workers related to a single depot. Every depot has its own instance of the module, separating the workers into resource clusters for easy gathering. The task master keeps the workers in sets and also designates a *task* for each worker. Initially a worker is tasked as `idle`, but other modules could re-task workers, essentially allocating and freeing workers. Each task has a related set of workers, stored as a dictionary of sets. Given n workers and a small amount of tasks, insertions, deletions and re-tasks operations has time complexity $O(\log n)$. We expect n to be in the range of $0 - 24$, so the operations are quite fast. The current set of tasks are `idle`, `mine`, `harvest`, `build` and `defend`.

Every task master is wrapped in a *Vassal* module, all of which are contained in a dictionary within the *Landlord* module. As the amount of needed Vassal instances are unknown, they are stored in dynamic memory. The Landlord designates new workers to or destroyed from the related Vassal. This allows for easier operations from superior modules to the worker pool.

The *Gatherer* module commands all gathering workers, superior only to the Landlord module and its immediate subordinates. Resource gathering was initially kept within each task master, however this makes it difficult to control gathering on a large scale. The resource priorities should not necessarily be uniform across all bases. Each frame, the Gatherer iterates through the vassals, commanding current gatherers and re-tasking all idle workers to gathering. How this is handled is explained the following sections.

6.2 Mining Minerals

StarCraft has a built-in worker gathering AI to help human players. Workers will automatically return cargo from resources (unless they are interrupted) and return to the same resource afterwards. When gathering minerals, they will move to another mineral field if the current one is occupied. This is an inefficient solution, as the worker could be stuck moving between minerals for long periods of time. It is also a liability for a bot, as the AI cannot be disabled. When harvesting gas they will wait until the refinery is unoccupied.

The simplest mineral gathering implementation is ordering idle gatherers to mine some arbitrary mineral. At some point, the built-in AI will ensure the workers are optimally scattered. It will however not be scattered immediately, and some workers will be very inefficient while moving from mineral to mineral. A simple but effective addition would be to scatter the initial workers.

By maintaining a queue of minerals, we can optimally scatter the workers. The first element of the queue is the mineral with fewest workers and the one in the back has the most. By continually assigning new workers to the first element and moving it to the back, we maintain a queue where the last mineral has at most one more worker than the first. Removing workers however requires finding the mineral in the queue, and should be done sparingly. Therefore, it is assumed any building or defending activity will be short and temporary, and workers assigned such will not be removed from the scattering. This might result in an ineffective scattering at some points. It is not clear however if optimizing the scattering at all times results in optimal resource output, as workers might be moved between minerals too often, resulting in less time mining.

If we maintain a dictionary of workers and their targets, we can assign new ones in constant time and retrieving targets in logarithmic time. This could be improved to amortized constant time with hashing. Removing a worker however requires a search through the queue which is linear time.

6.3 Harvesting Gas

Gas harvest is very much like mineral mining, but simpler. Rarely will players have fewer than three workers on each refinery, as refineries will be built only when needed. Furthermore, almost all late-game units require gas, and harvesting is slower than mining.

The implementation is identical to mineral mining. No AI tournament maps contain more than one refinery, but the gas harvesting still uses a priority queue. In this case however, all operations become constant time, so performance is not harmed while code is reused.

Chapter 7

Production

This chapter concerns the creation of units including the architecture for its management. First we explain the StarCraft terms of units and their production.

A *unit* is any player controlled entity, structure or not. Resources are also units, controlled by a neutral "player", but are both immobile and indestructible. Some spell effects, like the Terran nuke, are oddly also classified as units but they are neither selectable nor controllable. The term unit rarely includes these however.

Production here refers to the player controlled act of creating a new unit. Within StarCraft, producing a structure is called *building* and *training* if it is a non-structure. Once a unit is created, there is a duration where it is *constructing* before becoming *complete*. A unit cannot execute player commands and has no abilities during construction. Constructing structure are placed in the world when they are built, while non-structure are hidden within their constructor. When a non-structure is complete, it appears at the nearest free space around its constructor. A unit consumes resources upon production, including supply requirements. A constructing non-structure can be canceled for a full refund.

BWAPI notifies the agent whenever a unit is created or completed in separate events, which can be used to update internal data-structures.

As an additional note, BWAPI divides the map into *regions*, which contains resource and base locations. They are important because the regions border each other at *chokepoints*, which are either ramps or bottlenecked passages. Usually a region contains none or a single base location and resource cluster, however some regions contain multiple clusters with overlapping base locations. Because of these qualities, regions are very useful in both combat and economy, and is used throughout the implementation. They are needed for expansions, explained in Section 7.4. BWTA redefines the regions to contain the optimal base locations in expansion.

In the following sections we first briefly describe our location in the hierarchy. Then we begin with training of non-structure, moving on to building structures. Finally we cover expansions and expanding.

7.1 Production Architecture

The *Accountant* module is used for keeping internal track of spent resources and scheduled units. Commanding a unit to build or train will not spend resources until at least the next frame. Therefore it is required to keep internal records of these, otherwise different modules might spend the same resources twice. Unit scheduling is useful in later frames, such that modules do not request the same unit multiple times.

Training is handled by the *Recruiter* module, while building is done by the *Architect*. These are separate modules as the two jobs are very different in implementation. While the Recruiter is quite low in the hierarchy, but the Architect is not since it requires workers from the Landlord module. Both are superior to the Accountant.

There is a third module, the *Morpher*, which handles the rare case of a unit that executes a *morph* command. A morphing unit transforms from one type to another, which is most often seen in case of building refineries. Counter-intuitively, the geyser morphs into a refinery and changes ownership to the constructing player, but the refinery is still built by a worker with a build command. The Morpher exclusively monitors morphing units, while the Architect handles the building.

7.2 Training Units

The *Recruiter* needs to be low in the hierarchy, as it needs to be accessed by many other modules. It does not contain an independent AI, acting only through certain events and method calls. The current implementation is generic such that it can handle any train-able unit in the game.

To train units, the steps are twofold: issue the train command to a relevant, available trainer and then monitor the construction.

Commanding Trainer

Some units can train others, such as the depots that train workers. It is almost exclusively structures that can train. A unit can only train one at a time, although multiple can be queued up. Every unit is trained by at most one other unit, so a graph of trainers and trainees would be a *forest* structure.

Notice that it is inefficient to queue training, as this will lock resources. Instead, issuing the train command whenever the trainer is available will keep resources free without resorting to cancellation. It would be better to implement an internal queue if one was needed.

To command the training we require the trainer. For this we would like to keep records of all units capable of training. BWAPI inherently contains the trainer of any given unit type, so using a dictionary of trainers with their types as keys is sufficient and allows fast queries. From this we can search through the set of trainers until we find one that is available and then send the train command. The evaluation of trainers involve verifying the current existence and control of the unit, and ensuring it is not currently training another unit or has been commanded to in this frame.

Searching through the dictionary is logarithmic and iteration through the matching trainers is linear. Evaluating a trainer is constant. Given n trainers and m trainer matches, the time complexity for training is $\log(n) + m$, which could be improved to m with hashing. Neither n or m are usually very large, but m especially is only in the range of zero to four. Insertions and deletions of trainers are both logarithmic to their size.

Monitoring Trainee

Other modules require to know which units are currently scheduled when planning their actions. Units could also be destroyed before completion, in which case other modules might need to reschedule.

Once the train command has been issued, the Accountant is notified of the costs and the type.

Upon receiving the event that a unit has been created, the Recruiter frees the resource costs from the Accountant, as the game state by now has withdrawn the costs itself. The Recruiter inserts the unit into a set of incomplete units. When a unit is completed, the Recruiter is notified and removes the unit from the set and from the Accountant. The same happens if the unit is destroyed, which occurs if the trainer was destroyed.

Inserting and removing from the set is logarithmic to the size. The set is never expected to be very large.

7.3 Building Structures

Like training units, building structures is done in two steps: the structure is built and then it is constructed. Contrary to training, structures require both workers and a location to be built. Workers are the only units capable of building structures, and must move to the build location to do so. Terran workers must stay and construct structures until they are complete and Zerg workers are destroyed upon building, while Protoss require neither.

However, all Protoss structures must be in close proximity to a Pylon when built, and stops functioning without. The only structure exempt from is the depot and the pylon itself. This only becomes important in later stages of the game where Players might have to carefully manage their space. The agent is not expected to build enough structures for this to be an issue, so it is mostly disregarded. A Protoss structure is considered *powered* when in vicinity of a pylon.

Structure Placement

The Protoss player must place at least one pylon in any area he wishes to build in, and could distance pylons in a base to maximize coverage. Alternatively, if only a few pylons power a structure it creates a liability. The pylon could easily be destroyed to disable the structure, which is useful if it is a defensive structure or unit trainer. Fortunately, we expect the agent's bases to be compact enough such that we can place pylons somewhat arbitrarily and still manage to fit needed structures in the region. It is important however to place at least one pylon in a base before other structures can be built, but this is solved by the strategy modules.

Unless a building location is specified, the easiest solution is placing a structure as close as possible to the base location in the desired region. All the workers in the region are usually at the mineral fields by the depot, and therefore will not be far from the placement. The workers and depot become sheltered by the structures, such that the opponents troops are forced to move through bottlenecks to get the workers. Even if a region only has one exit, it might be desirable to spread structures as flying units can attack from any angle. Finally, the solution is easily implemented.

There are some exceptions. Depots will always be placed in new base locations and refineries can only be built on of vespene geysers. In both of these cases the building location will be specified by the superior module ordering the structure.

The Architect attempts all locations in the map in a spiral pattern around the depot, and returns the first location that is available. This is simple, usually cheap but very expensive asymptotically, as it will be linear to the map size. However we never expect to visit anywhere near all the locations, since few tiles in the map cannot be built upon. To determine availability of a location, it must be clear of units and the tile terrain itself must be able to be built upon. Both of these are handled by BWAPI. Note that this solution places structures in a square pattern, which is not actually the closest to the base location except in *taxicap geometry*.

Placing structures too close can block passage between them, especially for larger units. The hitbox is arbitrarily different between otherwise equal sized structures in StarCraft, such that some combinations next to each other will block some units but not necessarily all. This is because the structure hitbox usually does not cover its location completely, allowing some leeway for smaller units.

By placing structures at least one tile from each other, we ensure all units can easily pass through. This is done by keeping a map of all owned structure locations, where their dimensions are increased by one tile in all directions. If a build location overlaps any of the occupied tiles it is determined as not available. Registering new structures and querying this is constant time operations, but the space used is linear to the map size. It could be improved by only keeping a map for regions we have bases in, although not an asymptotic improvement.

Additionally the Architect avoids placing structures between resources and depots to avoid obstructing workers. By drawing the smallest rectangle including the depot location, geyser(s) and mineral fields, the Architect avoids blocking workers by not placing structures within this. Building the rectangle is linear to the amount of items in it and querying it is constant, but we expect the amount of items to be low (ten or less). This structure map is handled by the *Base Manager* module, which the Architect is superior of.

To summarize, if a location is not specified, structures are placed in a spiral pattern around the depot, distanced by one tile from others and outside the gathering-zone.

Aquiring Builder

Recall the Task Master module from Chapter 6. Since all workers are separated in base locations, we can easily pick a worker from the region the building location is. As the task master marks the jobs of all workers, we can pick a worker from either idle, mineral mining or gas harvesting (in that order). This way we do not interrupt other possibly important tasks. The Architect searches through the groups until it finds a worker that is not carrying resources, which then becomes the builder and is tasked as such in the Task Master.

The time complexity is linear to the amount of workers in these groups. However, it is probable that less than half the workers are returning resources at any given moment, so it is expected that we only need to check two workers before finding a viable candidate. In case there are idle workers, the operation is constant.

This solution does not work if we have no workers in the region. This is the case when expanding, where the worker must be specified by a superior module. This is the case when the *Settler* module builds an expansion, explained in Section **??**.

Executing Command

Once the builder has been retrieved, it will be commanded to build at the specified location. A player cannot build in hidden terrain, so the worker will first be moved closer to the target location in this case. When the entire placement has been revealed the builder is commanded to build the structure.

The build order will then be stored in a set, containing the structure type, builder and location. Every frame, the Architect verifies the validity of the build orders, including verifying the builder has not been re-tasked. It also reissues commands to builders if required. As with monitoring trainees, incomplete structures are important to keep for the same reasons. In case of structures, there are more things that could go wrong which would incur cancellation of a scheduled structure. The builder might be destroyed or the build location might prove to be invalid upon being revealed.

All invalid orders are removed, forcing superior modules to reissue the build order. This is desired compared to repairing the order, as the build order might no longer be required.

Monitoring Constructions

Since Protoss structures auto-construct, the implementation is identical to monitoring trainees with the same time and space complexities.

As mentioned prior however, refineries are handled by the Morpher module. BWAPI is notified when a unit morphs, upon which it is inserted in a set of incomplete morphs. A morphing unit is not considered constructing, although it is incomplete. When a unit is finished morphing, no event is called, therefore forcing the Morpher to verify all morphs every frame. However, there are very few expected morphing units at any given moment.

7.4 Building Expansions

Recall that *expansions* are additional depots beyond the initial one, built at new resource clusters. As explained in the chapter 6, it is often profitable and sometimes necessary to expand resource harvesting to new locations. To avoid transporting cargo all the way between regions, players have to build depots near resources they wish to harvest.

When BWTA analyzes a map, it marks all viable depot locations. These are positions in which a depot will be at optimal distance from nearby minerals and geysers. Usually every resource cluster only has one of these positions, but some locations may contain overlapping depots. Regions rarely have more than one base location. Given a region, we can obtain all internal base locations, including these depot locations.

This significantly simplifies the construction of expansions, however the base location to settle must still be determined.

The *Settler* module is responsible for all expansion behavior, and is also used to determine if expanding is possible. To build an expansion we need to specify a location and builder. These are handled by the Settler module, while the rest is done by the usual build procedure in the Architect.

Expansion Placement

To determine whether a base location is fit to expand it must not already be expanded upon or contain enemy forces. Additionally, it must be reachable by non-flying units and have a path of regions without enemies. If these are satisfied, we consider the base location available for expansion.

From the starting region, we visit neighbor regions recursively in order of proximity by using a priority queue, where the lowest priority is picked first. Initially the start region with priority 0 is inserted. When we visit a region, we visit all contained base locations. If any are available, we return one as the target location of expansion. In case none are available, but the region is unoccupied by enemy forces, we add all unvisited neighbors to the queue. The priority is equal to the distance between the two regions centers plus the current regions priority. If the region does have enemy units within, its neighbors are not added to the queue. The algorithm terminates when the queue is empty or an available candidate has been found.

This solution satisfies our expansion requirements, and additionally includes the starting region should the depot be lost. It is very crudely implemented however, and quite expensive. Determining whether a region is occupied involves checking all enemy units. Given r regions and n enemy units, the time complexity is rn. This could be optimized by first marking all occupied regions before the recursion.

The enemy units are retrieved from the *Archivist* module, detailed in Chapter 8.

There could exists more factors when determining expansion locations beyond distance. Resource quantity or defensibility are also important, but are not considered here. One could favor regions distant to the opponent for added defense or alternatively closer for better map control.

Expansion Builder

An expansion is often the first structure in a region, and therefore the player must acquire a builder elsewhere. The Architect only attempts to pick workers from the region the location is within, so the Settler must specify a worker beforehand.

In this case, we just pick a free worker from the main base. There could be workers closer to the destination in other regions, and in this case the solution is sub-optimal. Recursively checking neighbors in order of proximity would result in an approximation of the closest worker. It was not a priority however as the improvement would be insignificant.

The solution is cheap, and costs the usual to pick a worker. A worker considered available is either idle or tasked with gathering.

Chapter 8

Exploration

One of the challenges in StarCraft is the imperfect information players have of the world. It becomes necessary to deduce the opponent's strategy from a few units, and regular scouting is mandatory. The player with the best information coverage can make the most optimal decisions. This chapter describes how the agent solves the information and scouting problems.

All units, including buildings, have a line of sight equal to their *sight* value. Within this distance, the map is revealed to the players. Everywhere else, only the terrain features are visible or buildings in their last seen state. This is the *fog of war*, and all units within this are *invisible*. These units cannot be targeted or attacked, and the agent cannot access any data of them through BWAPI. It is up to internal data systems to store this information.

The entire map terrain, including initial resources, are revealed to players. All possible start locations are also known, but it is not known which the opponents occupy.

Ground units cannot see past cliffs or ramps, giving an edge to units on the high ground. Most start locations are located on plateaus, making them easily defended. Units or obstacles do not break line of sight however. Some units are *cloaked* or can *burrow* which renders them invisible to normal units even outside fog of war. BWAPI however does not limit the agent from accessing their data while they are within line of sight. Units with the *detector* ability reveal cloaked units within its line of sight.

The agent must track data about discovered units, which is handled by the *Archivist* and *Geologist* module, the subject of the first section. The second describes the scout AI, implemented by the *Reconnoiter* module.

8.1 Tracking Units

Two things must be recorded for each enemy unit: position and type. The former is needed to track movements of the enemy army and to record enemy base locations. The latter is needed since the unit type contains all relevant values such as speed, damage and maximum health. Some units can change their type even, requiring regular updates of changes. The Siege Tank for example changes types when it goes into or out of siege mode.

The *Archivist* handles all known enemy units. The solution is sets of units and dictionaries of their data, where units are keys to their values. Every frame, the Archivist checks all stored units, updating their values if they are visible. Whenever a unit is first discovered or destroyed, BWAPI will notify the Archivist, which will then modify the dictionaries and sets appropriately. This solution spends logarithmic time on queries, insertions and deletions. The position and type queries could be improved to amortized constant time with hashing.

Units are inserted into multiple sets, separated in categories. Some units are stored in more than one set such that constant time retrieval is possible, at the cost of more marginally more space. These sets are used by other modules when specific kinds of units are desired, such as structures or workers.

The Archivist is one of the lowest in the hierarchy, as it depends on no other module and is used by almost all others. Other modules query the archivist for the units or their values.

Resource Locations

As all initial resources and their positions are retrievable from BWAPI at any time. Since new minerals are never created, it never becomes necessary to keep track of them here, as only the Gatherer module from chapter 6 needs them.

Vespene geysers however have special behavior in StarCraft, as mentioned in chapter 7. When a refinery is built upon a geyser, it is morphed rather than constructed.

However, if the refinery is destroyed, a new, distinct geyser unit is created in its place. While all the initial geysers can be retrieved from BWAPI, new ones cannot. Otherwise, agents could deduce if any opponents lose refineries. This makes the tracking of geysers a bit more tedious than other units.

The *Geologist* keeps track of current known geysers in the map. They are stored in a dictionary of sets, where the keys are regions and the sets contain geysers from that region. This is because superior modules need the geysers separated in regions. The Geologist responds only to events, inserting new geysers or deleting refineries when discovery. All initial geysers are added on initialization. Since a refinery were the same unit as the late geyser, they can be used to remove them from the sets. Given r regions and g geysers, insertions and deletions cost $O(logr + logg)$ time. Since every region rarely has more than one geyser, the time is more like $O(logr)$, which is the cost of a query. As usual, this could be improved with hashing to amortized constant time.

8.2 Scouting

Scouting is important throughout the entire duration of the game. Early on, players need to know where the opponent is and which opening they are using. Later on, observing the opponent's army size, unit types, base expansions and tech level is important to properly combat their strategies. The opponent's race is known unless they pick random, in which case this also needs to be scouted. There are no bots playing a random race however, so this is not relevant to us.

Recall that all possible player start locations are revealed to players. Scouting for the opponent base is then just visiting other start locations than ones own. Maps used in the competitive bot matches contain no more than four start locations. It could therefore be the case that a player has to scout up to two bases before knowing the enemy start location. Usually a player wants to know more than where the opponent starts, so even when it is deductible where the opponent is, the player wants to scout the base itself.

Once the enemy base has been revealed, and with it his opening strategy, the scout is not useful any longer. It is a long trip back to gathering resources and the scout could be followed, revealing your own location. Harassing enemy workers however can disrupt the opponent's economy, even if none of them are destroyed. Attacking enemy workers forces the opponent to pull at least one worker from gathering to defending. Usually there are no base defenses or troops at this early in the game, except if the scout had to visit three bases. Scout harassment can be very advanced, as the scout can retreat while chased, possibly attacking passive workers. Killing even one worker is a big advantage this early in the game.

The *Reconnoiter* module contains the scouting AI. Using the TaskMaster module from chapter 6, the Reconnoiter picks some free worker as the scout. As usual, a worker is considered free if it is either idle or gathering resources. The scout is removed from the worker pool, as it will be a scout for the rest of its life span.

Usually a Protoss player will send the scout after the first pylon has been built, where the builder is used as a scout. As an approximation, the Reconnoiter will only acquire a scout if a specific supply total of 8 has been reached. This is because the first pylon is usually built around eight workers, such that the Reconnoiter should acquire a scout at the same time the pylon is built. The worker will never be picked however, as it is not considered free the moment the supply limit is reached.

If the Archivist has not recorded any enemy buildings and the Reconnoiter has no scout, it will attempt to acquire a scout. This implies scouting will proceed as early as possible and if all known enemy buildings has been destroyed, extending its use into later stages of the game.

While the Reconnoiter has a scout, it picks an unexplored base location and moves the scout towards there. A tile is considered unexplored by BWAPI if it has never been revealed for the duration of the game. Note that this removes the agents own starting location from the candidates. Once the scout reaches the destination the tile will be explored, removing it from the possible scouting locations. In case this is the opponent's starting location, their depot will be revealed as well.

There are alternative uses of the scout compared to harassing, such as building blocking the opponent's main base exit with defensive structures or building troop producers close by. These however must be planned in concordance to the grand strategy, as they involve spending resources. Some bots use these strategies, LetaBot for example can rush with bunkers.

At later stages in the game, faster units should be used to scout the map. In particular the Protoss Observer is a good scouting unit, as it is both permanently cloaked and a detector.

Chapter 9

Combat

This chapter explains how combat is handled in StarCraft, how its outcomes can be predicted and how the agent manages troops, attacking and defending.

Combat in StarCraft can easily become the most complex part of bot development. Multiple studies has been done on just small subsets of artificial combat scenarios, with specified units and simple terrain, but even these just scratch the surface of the field. Like the macroscopic strategies in StarCraft, there is likely no optimal command scheme for ones units. Predicting the outcome of a combat scenario, is therefore only done as approximations. This is also one of the areas where bots can excel against humans, since they can easily command different units in complex ways.

Troops or *fighters* here means units used for combat. Almost all non-building units beyond the workers are used in combat. Some units called *spellcasters* deal damage indirectly through their abilities, while *support* units help others fight better.

Every unit has some amount of *health*, and almost all units can *attack* to deal *damage* to others. There are a number of additional factors such as armor and damage types, splash and hit probability, but it is sufficient to know that a unit loses health relative to the damage it is dealt. When a unit reaches 0 health, they are destroyed. Units attack at different ranges, such that *melee* units need to be close and *ranged* units can attack at a distance.

In the following sections we will explain the base management of troops. Then we move on to combat prediction, followed by attacking and defending.

9.1 Managing Troops

The *Army Manager* is the troop parallel to the Task Master from Chapter 6. It contains all fighter units controlled by the agent and their current assigned *duty*, same as the *tasks* for workers from Section 6.1. The current types of duty are `idle`, `defend`, `attackTransit` and `attackFight`. Distinct from the Task Master, there is only one instance of the Army Manager, as troops are not separated into regions.

9.2 Combat Prediction

Predicting the victor of combat is very difficult in StarCraft and predicting the specific casualties is neigh impossible. Even beyond the numerous factors in combat, the optimal command of troops is completely Dependant on the opponent's commands. Unless the opponent operates in a recognizable pattern, they movement is unpredictable. Predicting combat is therefore more based on what units the opponent controls, how the terrain is and then either a theoretical upper limit of their damage output versus a prediction of our own.

Some of the most successful predictors simulates a simple form of combat and evaluates the result. This however assumes how the opponent will control its troops, or at least assumes it knows the optimal control scheme. Neither of these are possible, especially not with incomplete map information, but the results are usually close enough. SparCraft uses this method with good results, but even it ignores multiple factors including collision.

The combat prediction algorithm resides in the *Combat Judge* module, which is used only by the Attacker and Defender, covered in this chapter, and the *Strategist* covered in Chapter 10. Given a set of units, it returns a value representing the *strength* of them. The army with the higher value is predicted to win in a confrontation.

Following is a description of the prediction heuristic: let us assume a simple scenario: two units, a and b are attacking each other. We have the health values h_a, h_b and the *damage-per-second* (DPS) values dps_a, dps_b. Assuming both units start attacking each other at the same time, then we must have $TTK_a = h_b/dps_a, TTK_b =$

h_a/dps_b, where TTK_a and TTK_b are each unit's *time-to-kill* (TTK). This value represents the time in seconds it takes to kill the opponent. It must be the case that the unit with the lowest TTK becomes the victor, and with this we can predict the outcome of the simple situation.

Notice that we can simplify the solution and instead simply compare the values $s_a = h_a dps_a$ and $s_b = h_b dps_b$. This enables us to value the strength of an unit independently of the enemy, which is quite helpful when pulling workers one at a time in Section 9.4.

We would like to scale the solution to armies by using the sum of strengths as the army's strength. This requires the additional assumption that no units are destroyed during combat. If this happened the TTK would not be constant, changing the outcome. This is almost never the case in real combat scenarios, and can cause the prediction to be inaccurate. This happens when armies have the same strength value, but not the same amount of troops, where the largest army is overestimated.

The prediction was already inaccurate however, as it never takes terrain or range into account. For this reason, the prediction underestimates Terran units which are all ranged. Additionally, the prediction algorithm has two properties that is not true in-game. Firstly, the strength of armies increases linearly with its size, which is not necessarily true in-game. When grouped in combat, the units destroyed last will have been alive longer than if it had been alone, thus having dealt more damage in total. Secondly, the comparison between armies is non-transitive in-game. This means the strength of an army is relative to others, and cannot be accurately evaluated as an absolute value like here.

However, this solution has proven to be sufficient in the early game. Protoss has the strongest early unit, which mitigates the disadvantage of attacking when outnumbered. The units will usually be able to destroy one or more of the enemies, such that both players loses troops. The only case where this is not true is against other Protoss players, but here the prediction is perfect as both use the same early unit type.

The prediction is very cheap, having a time complexity linear to the size of the army and uses constant amount of space.

9.3 Attacking

A player will almost certainly expand to their natural expansion first and expanding to a third base is only viable beyond early game. The natural expansion is very close to the main base and is usually right outside the only exit of the main base. This proximity allows us to consider both bases as a single base. From these assumptions we can conclude the opponent will only have one base for the duration of the early game, which is the only stage we are concerned with.

In euclidean space the shortest path between two points is the same regardless of which is the origin. In terms of StarCraft, the shortest path between the opponent's and the agent's base is the same regardless of which the troops move from. Assuming each player only has one base and both players will use optimal pathfinders, we can model the map as a straight line between the bases. Both armies will only travel along the shortest path and there is therefore no case of armies moving past each other without colliding.

When rushing, it is favorable to move troops as close to the opponent's base as possible without confrontation, even before they are ready to attack. This pressures the opponent, scouts the map, attacks the earliest possible time and distances the opponent's troops far from the player's base. Even if the fight ends in defeat, the opponent's troops will have to move across the map to attack, and in case of victory, the player's troops can continue towards the opponent's base with minimum delay.

When the we predict a victory in combat against the opponent army, we attack. In some cases it might be prudent to wait until more units have been amassed, especially if we produce more troops than the opponent, as we will sustain fewer losses and be more certain of a victory. This is difficult to asses while also being very risky, so the safer option is just attacking immediately.

Attacking is handled solely by the *Attacker* module.

Targeting

Most real-time strategy games has the unit command *attack-move*, and StarCraft is no exception. A unit executing this will move towards a unit or location, but will attack any enemies along the way. The unit will prioritize attacking those that are damaging it. This command therefore auto-targets nearby enemies and also has simple, inherent prioritized targeting.

By targeting the enemy base location or structures with this command, troops will automatically fight the first enemy they meet, which is either the opponent's base or army. The most important targets are the enemy structures, since destroying all of them is the win condition. The Archivist from Chapter 8 records all known enemy buildings, from which the Attacker picks one arbitrarily. With the assumption that the opponent only has one base, the army will attack the foremost building regardless of target, since they are attack-moving. In case there are defensive structures, the attack-move command will automatically re-target to them once they damage the army.

This allows for constant time targeting retrieval. Alternatively some units could be prioritized, such as the enemy workers or troop producers, but this is otherwise the cheapest solution while still a good approximation of optimality.

Some bots manually prioritize during combat. There have been studies on using different priorities, such as attacking the unit with highest health to resource cost ratio. This is a greedy attempt at costing the opponent as much as possible. A similar one is attacking units with highest health to damage ratio, or simply the unit with lowest health. There are more advanced strategies however when considering large scale armies composed of different unit types, where counter-play between individual types are important. The usual order of targeting is the opponent's army, then economy and then unit producers.

Troop Rendezvous

Once any fighter unit has been completed, they are set to the duty `idle`. Every frame the Attacker will re-task all the `idle` troops to `attackTransit`. These units are commanded to move towards the enemy base, but not attack. Otherwise, they would attack the enemy base one at a time. Grouping troops is more efficient as the total health pool is larger, allowing the troops to deal more damage before they are destroyed. This has the opposite effect on the opponent's army, which has less time to deal damage. Therefore, it is more efficient to wait until more troops have arrived.

At some point the troops will be within some distance of enemy units where they will risk being attacked. This distance is a bit larger than the longest range of any unit. Once troops are within this distance, the Attacker will use them for combat prediction. By only counting these arrived troops, the module can tell when there are enough to beat the opponent. If they are strong enough, the Attacker will re-task them to `attackFight` duty. In this state, they are commanded to attack move the target found prior. It is important to attack immediately when possible, as waiting too long will allow the opponent to counter the attack with stronger units or defenses.

In case the `attackTransit` units within this danger distance is not strong enough, they will instead retreat towards the main depot. The result is troops moving back and forth right outside of enemy range, until enough troops have gathered. In case the opponent advances with a stronger army, the troops will retreat until outside their range. In case the enemy army reaches the base, the Defender module takes over the troops, which is explained in Section 9.4.

`attackFight` units never retreat, fighting to the death. This is because it is difficult to determine when retreating is efficient. While troops retreat, the enemy army has the opportunity to attack them without receiving any damage themselves. In some cases retreating might be impossible if ones units are too slow. In case the army would sustain heavy losses, it might have been a better choice to destroy as many of the enemy troops as possible. All troops with the `attackFight` duty outside of enemy range reverts to the `attackTransit` duty, moving towards the enemy base once again.

The Attacker only counts units currently within this danger distance of an enemy unit as arrived, that is, it does not record arrivals between frames. Testing proved the battlefield shifted too much to record arrivals and stopping troops until they were ready, as it lead them to becoming scattered across the battlefield sometimes. By only relying on the current situation, it is certain that the units are actually gathered.

This current implementation is expensive as every unit compares itself with every known enemy to detect whether they are within range. Given n troops and m enemies, the time complexity is a polynomial nm number of operations. All the remaining operations are asymptotically less than this however, so this is also the total.

There is not always enough space for all units to wait if the army is large, but it is usually not. A better solution would be needed however if the army was larger or contained large units. In this case, heat maps or cluster detection could be used to detect the density of the army.

9.4 Defending

If a player's base is under attack, different behavior is required compared to attacking. Even if outnumbered, it is favorable to fight the invaders. In case the defending player has no troops left, the workers function as a last line of defense. Usually the opponent will attack the workers first in any case, so they might as well fight back.

Since Protoss are quite slow, it is possible the opponent rushes first. In some cases, no troops might have been built yet, and it is up to the workers to defend the base. Zerg is very quick to produce troops and is capable of this, but other races could send workers to build a forward base. The usual case however is defending against the opponent's scout. Even if the scout simply stays in the base without attacking, it is important to destroy it quickly to starve the opponent of information.

Some strategies involve building static defense structures, which are stronger and cheaper than the mobile troops. These can be built very early if the player is determined, at the cost of expanding economy and not pressuring the opponent. Beyond early game, these defensive structures are usually always built to prevent ambushes and detect cloaked units. Rushing players however, favors mobile troops over these defenses.

The *Defender* module is capable of defending with nearby troops, including workers if necessary, against any non-flying, non-cloaked threats. The module is a sibling to the Attacker in the hierarchy, but has a higher priority as defending the base is more important than attacking. Therefore the Defender is updated before the Attacker in each frame.

Scrambling Defenders

The problem with acquiring defenders is related to the retreating problem. It must be avoided pulling troops back that would be better used attacking. Defending is an attempt at damage control, however the opponent must still be pressured with attacks as quick as possible. Some units might be too far away to participate significantly in the defense, in which case it should instead attack the opponent. In case both players are invading each others' bases, it becomes a race. In these situations, it is not always clear what the correct move is.

In line with the non-retreating paradigm from Section 9.3, we will assume that units already in combat should not be interrupted. Additionally, units within regions with expansions should defend, as the unit will be within reasonable traveling distance and is not currently attacking. Since the macro strategy is rush, units that are not within an region with expansions should continue to attack.

As it turns out from testing and because of the single path model, the agent very rarely have any units left outside its base if the opponent is currently invading, as all prior units would already have fought and been destroyed. In case there are units beyond, they will usually have reached the opponent's base before his army invaded, in which case the correct decision is continuing the assault. The current solution has therefore proved to be sufficient.

There is a slight issue because of the region clause. If the opponent has a large army which has not completely crossed into the defended region, the defenders will revert into attacking behavior upon destroying all those within the region. Sometimes the defending units will continually retreat and attack as parts of the enemy army moves into the region, sustaining more damage than is necessary. It is a difficult problem to solve however, as the Defender would need to detect and target the clusters of units rather than the individuals.

Once all defenders have been assigned, they are commanded to attack-move an arbitrary invader. The defending units will then prioritize the closest invaders. If the invaders are defeated, all the remaining defenders will be set as `idle` in the Army Manager.

Rallying Militia

Sometimes there are not enough troops in the base to defend. In this case, a common strategy is moving all the workers to defend.

Once all available defenders have been assigned, the Combat Judge will calculate the values of the invaders and defenders. Until there are no more free workers or the defenders are stronger than the invaders, the Defender will task additional workers to defend. They are commanded in the same fashion as the rest of the defenders. Once the fight is over, all the defending workers will be re-tasked as `idle` in the Task Master.

Alternatively, in case there are enough troops to defend, players might temporarily relocate workers to safe resource clusters until the conflict is resolved. This is rarely an issue in the early game however, and has not been a priority yet.

Chapter 10

Macro Strategy

Two strategy terms within StarCraft is *micro* and *macro*. The former is a players ability to micromanage units and their commands, where the latter is general stratagem and economic management. Human players have to balance the two, but bots are not so limited. Up until now we have almost exclusively covered micro behavior, such as how units are commanded, produced and tracked. This chapter concerns the bot's macro strategy and its architecture, which covers economy, strategies and production.

The first section describes the strategy module hierarchy with *Despot* as root module. The second section explains the opening strategies and how they are implemented in the *Planner*. The third section details the economic decisions dealt by the *Economist* where the fourth covers combat strategy handled by the *Strategist*.

10.1 Strategy Architecture

While the Architect and Recruiter handles the technical aspects of production, they never produce anything of their own volition. This is the concerns of the *Planner*, *Economist*, *Strategist*. The Planner executes build-orders if any is queued. The Economist commands worker and supply production, where the Strategist handles troop and their trainers' production. All three are direct subordinates to the *Despot*, which controls which of them are active. In case the Planner has a build-order queued, only it will be active every frame. Otherwise, the Economist and Strategist will be active while the Planner is not.

10.2 Build-order

An *opening* refers to the beginning strategy a player uses, for example rushing. Opening strategies usually include a *build-order*, which namely is an order of units to build. It is a strict progression of the early game that has been found through experience, exactly like chess openings. They are designed to work against current strategies in the environment, but do rely on the player's judgment on whether to change opening.

Every unit in the build-order is set to be produced when a specific amount of supply has been used. It is implied that workers are produced until the next supply threshold is reached, upon which the next item in the order is produced. Some build-orders are more detailed, specifying placement or techniques to be used.

Because of the dependency on the environment, like all kinds of strategies, they are known to change every now and then as different strategies become popular between players. This is also apparent in the bot scene, where the over-saturation of one strategy encourages counter strategies.

Build-orders are easily used by bots as they are a simple queue of orders. The *Planner* has such a queue, which is executed at the Despot's discretion. If the queue is not empty, as many of the next items are executed instead of using the Economist or Strategist. They become used again once the queue is empty. This allows for constant time execution per item, barring the time to produce the items. Thus the implementation is very cheap as every item is only produced once and spread across many frames.

Instead of implying the production of workers between orders, the build-orders are modified to have them expressed as items in the order.

The benefit using build-orders with a bot is easily adding multiple openings for the bot, depending on the match-up. Although the general strategy remains, different openings can vastly change the outcome of the game. Proper choice of opening is key to winning in early game, or at least not lose during. The downside however is the rigid planning structure, which can't allow for easy adaptations or responses. In case some units are destroyed, it is possible that the current build-order is no longer optimal, viable even. Humans can easily adapt to these changes, but this is not always so with a bot, where canceling the build-order can be necessary.

We avoid this by only using short build-orders, such that the opponent could in no way interrupt with any units other than harassing workers. Even shortened, they are still useful.

This solution is also used to prioritize production. Since the Economist and Strategist use a greedy strategy, it can be difficult to build high-cost units such as the depots. By enqueue the depot, all other production will be paused until it can be created. This solution is risky however, as the bot will not consider removing the depot from the queue under any circumstance. Practically, it does not seem to cause problems, as the bot will reach the required amount of resources quickly after ensuing, resuming normal behavior again.

10.3 Economy Strategy

Economy includes producing workers, refineries and depots and managing these. The *Economist* handles these aspects of the bot, updated at the discretion of the Despot.

Resource Gatherers

Recall from Chapter 6 that there is a specific maximum gathering workers on each resource. Any beyond this will not yield any returns. Therefore, the optimal worker amount is having the maximum allowed workers at each resource cluster.

Generally it is advised that a Protoss player keeps producing workers constantly until late-game, but some openings might temporarily stop worker production. This is not the case here however. Therefore, while there are not the desired amount of workers in a base, we attempt to produce a new worker at that base. This is verified checked every frame. The asymptotic time-complexity is therefore linear to the amount of bases and the time required to produce a worker (see Section 7.2). However we expect to only have one to three bases and the worker production is constant time since the depot is given as trainer.

Expanding

Recall the *expansion* from Section 7.4. Determining whether when to expand is difficult but pivotal to the match's outcome. It takes time and resources to build a new depot where it will be vulnerable before completion. A player with a stronger army can safely expand while guarding it, securing his current advantage or at keeping up with the opponents' expansions. Without the stronger army it is risky, but could become necessary. The resources in current bases will be depleted at some point, leading to a constant need to expand. Expanding before current resources are completely saturated with workers is a *fast expand*, sometimes used as an opening. This is very risky, but with an immediate payoff in economic advantage. The contrary formula, keeping a single base until mid-game or alike is called *one base play* and also possible as long as the opponent is blocked from expanding.

An expansion is desired by the Economist when all mineral fields have been saturated, as otherwise economic growth becomes impossible. Before this, it is sometimes necessary depending on the match, but this is difficult to determine. This expansion might be too late in some cases, but will at least ensure a consistent worker production throughout the game.

There have been tested three versions of bot with regards to expansion. The initial version never expanded, where the second version would always expand whenever desirable as described before. The third version would toggle between the behaviors, turning expansion behavior on if the opponent built defensive structures. These would usually only be used early in a turtle strategy, in which case the opponent is passive and expansions are less risky. The results and their discussion can be found in Chapter 11.

An earlier version of The bot used a fast expand opening, but it was less successful compared to its usual one base play. The fast expand is not always viable, and as such requires some modeling of the opponents' strategies to determine. It was down-prioritized in favor of the more versatile saturation expansion.

Since expanding is expensive, the Economist needs to save up resources. As mentioned in Section 10.2, an expansion is built by queuing it in the Planner, pausing other productions until it has been built.

Maynard Slide

Once an expansion has been built, it will initially have no workers at its resources. The optimal solution is have all workers equally scattered across all resources by depots, since this ensures maximum return value for each worker. If a base reaches maximum worker capacity, it can no longer contribute to worker production efficiently since completed workers will have to be moved. Additionally, older bases will not deplete resources as quickly and new bases will return their investments immediately.

Therefore, upon building a new depot, players will equalize the workforce between bases by moving workers to the new expansion, called a *Maynard Slide*. This has been subject to some statistical analysis, as it is a quite common strategy in StarCraft. When moving workers, there must remain at least one per resource since they already have a maximum return rate.

This is implemented in the Economist, which is notified by BWAPI once a new depot has been constructed. Iterating through vassals in a random order, it moves workers beyond the first per resource to the new base, until it either runs out of available workers or reaches two per mineral. It is not concerned with gas harvesting, as this did not see use.

Building Supply

Recall the *supply* resource explained in Chapter 6. A player can have a maximum of 200 supply, and it is required by almost every non-building unit in the game. The challenge issued to players is being the least possible *supply-blocked*, that is having reached the supply limit, as this will block production.

The simplest supply implementation is to build a supply unit whenever the limit has been reached. Production will however still be stopped while the supply structure is being constructed. Building supply before the limit has been reached will shorten or remove the pause, which can allow for unbroken production depending on the available resources. If the supply is built too early, a lack of resources might also block production. The optimal solution is where there is the least amount of time in which units are not produced. Observe however that this is alike the *Job shop scheduling* problem, but more complex with the additional factor of resources. As such the problem is NP-complete, even if future resource intake was predicted.

An improved version of the simplest solution, implemented in the Economist, is to build supply when the current limit is below reserved supply plus a constant. This somewhat works, as the agent quickly reaches a specified amount of unit producers in each game, thus requiring the same buffer. The constant was found through some testing, where the result is the supply is built a bit too early. This however impacts performance the least beyond an optimal solution, and has proven sufficient.

The same solution could be made dynamic by making the constant variable according to predicted supply needs. Since the required construction time for the supply unit is known, the Economist could predict how many units will be produced during this time, and set the variable accordingly. The prediction could be done without accounting for resource costs for simplicity.

10.4 Combat Strategy

Building structures and training units for combat is handled by the *Strategist*. This module decides which units become available for the Attacker and Defender, and therefore controls the overall strategy of the bot. As the bot is using a rush strategy, the goal of the Strategist is to train as many troops as quickly as possible. There is a balance between producing trainers and troops, as having multiples of the former helps the latter. Too many troop producers might reach more troops faster, but spends a lot of resources that could have become more troops. Too few producers, and the rush will be too late.

Rarely does the first rush win the game, the rushing player instead keeps pressuring the opponent with attacks. The goal of each attack is to either replace lost troops faster or destroy more units than the opponent, until they are overwhelmed.

Building Troops

The first unit the Protoss has access to is the *zealot*. Compared to the other units first available to the other races, the zealot is much more expensive but also a lot stronger. It is known for being very cost effective and a few of them are able to beat other races' rush attempts before more advanced units are available. This makes them ideal for rushing, even though this is seldom used in human competition, it has proven very effective against bots, though the reason is unclear.

Commanding units effectively is very difficult, and as Chapter 9 described, this agents grasp on units in combat is simple. Fortunately, the zealot's brute force strength, coupled with regenerating shield and melee attacks makes it easy to control effectively. In favor of avoiding the use of much more complex units, the Strategist only trains zealots.

This does have issues. While they are cost effective, they are still out-performed more advanced units, especially air- and cloaked units which they cannot attack. However, these units are only available later in the game, at which point the agent will have amassed a sizable army of zealots and attacked.

As this bot is very simple in terms of handling units, it is appropriate to use the first available units and attack as quickly as possible. The idea behind this strategy is, that since this bot lacks greater strategic reasoning and handles late game units poorly, then it should attempt to win the game before any of that becomes relevant. This can beat more complex bots, simply because their otherwise superior strategies are never allowed to hatch. Additionally, Protoss has the strongest and easiest to use early game units, and when massed will easily beat both Terran and Zerg.

The Strategists' production logic is just a greedy algorithm, attempting to produce more zealots each frame.

Against static defenses the zealots fare poorly. This is where the bot attempts to expand and instead overwhelm the opponent in zealots, which has mixed results. The economic advantage can sometimes be enough to outweigh the technological disadvantage. An earlier version of the bot attempted to use the second available Protoss unit, the *Dragoon*, but it was too inefficient with the simple combat techniques. The expansion feature was prioritized instead.

Building Troop Producers

Since the Strategist only uses one unit, it only needs one kind of structure called the *gateway*, which can train zealots. Again the Strategist uses a greedy algorithm, attempting to build more gateways every frame until it reaches a set threshold of troop producers.

It was found through testing that the starting location, with maximum worker capacity, can support four gateways constantly producing zealots. This gets more difficult to asses with expansions, and through testing it was found that greedily building limitless gateways was not efficient. The current limited solution is building four gateways without expansions, and six if there are any expansions. A more scalable solution has not been implemented as the game usually does not reach further.

Chapter 11

Results and Evaluation

The bot was submitted to the Student StarCraft AI Tournament (SSCAIT). While the tournament itself has not started yet, the bots still play against each other in live-streamed matches. There is a scoreboard where bots are placed accodingly to their win-rate, with total wins and losses marked. As mentioned in Section 10.3 there are three versions of the bot, all of which has been submitted. The versions are "One-Base", "Always-Expand" and "Condition-Expand". The first version never expands, the second expands whenever desirable. The third behaves like the first version, until the opponent builds defensive structures where it switches to the second. Table 11.1 contains the results of the three versions.

The initial One-Base version did very well, netting almost an 80% win-rate. The next version, Always-Expand, was not tested as thoroughly, but it is clear that it did not perform as good. It turned out that by far most of the bots were aggressive rushers, which often would destroy the first expansion. At that point

Version	Games	Wins	Win-rate
One-Base	780	527	79.23%
Always-Expand	104	78	75.00%
Condition-Expand	79	63	79.75%

Table 11.1: Sparring results from the SSCAIT website

the bot would be behind in troops and quickly lose. This spurred the Condition-Expand version, where it would not expand against rushers. Unfortunately, it only marginally improved the win-rate, and with few enough games for the results not to be conclusive. This is probably because the expansions was not enough to win against the defensive bots, in which case the outcomes of games were unchanged.

As one could expect, the bot does well against others using boom-strategies and proxy-strategies. It wins by far most of the matches against other rush bots, including mirror-matches, except UAlberta bot as mentioned. It fares poorly against the turtling Terran bots, which quickly goes for more advanced units. The expansions were an attempt to combat the turtle strategy, but it was not enough. While the bot does gain an economic advantage, it never uses it to train stronger units.

SSCAIT accepts bots from non-students which only play in mixed-division where student bots play both that and the student-division. The statistics above is from the mixed-division, and there is unfortunately not a student-division only statistic. The One-Base and Condition-Expand versions are usually among the top 8 mixed-division bots and contender with a few for the first place in student-division. This is difficult to ascertain without clear statistics and will first be determined with the tournament results.

The bot will be entered in the 2015 AIIDE and CIG competitions, and further development on the bot is planned.

Performance

Performance wise, the final version(s) of the bot do very well. SSCAIT monitors the frames as there are time limits bots must satisfy to upkeep the real-time aspect of the game. The rules are:

- no more than 1 frame longer than 10 seconds,
- no more than 10 frames longer than 1 second and
- no more than 320 frames longer than 55 milliseconds.

The bot has never lost a match due to breaking the time limits. In fact, it has not been observed to once exceed 55 milliseconds in a frame. Not unexpected however, since the bot has favored simple and very cheap solutions. There are certainly some algorithms that could be improved, such as detecting troops' proximity to the enemy. These are the minority however, and the current implementation allows for further development without reworking it.

Features

The bot never reached any high-level, complex AI solutions. The focus was creating a competitive bot, so development focused on areas that improved performance the best. This turned out to be low-level technical features such as managing and commanding units and internal production data-structures. It was much more important that the bot acted correctly than it acted optimally, so simple solutions was sufficient in almost all aspects. Further development should focus on improving combat predictions and map awareness with regards to enemy unit placement, such that the bot can play in later stages of the match. Additional features would be controlling more kinds of units and better adaption to opponents' strategies.

As the results show, the expansions did not improve performance noticeably. The economic advantage gained from expansions were not enough to offset the disadvantage of only using the earliest Protoss combat unit. By increasing the array of units used by the bot, the advantage of expansions may be clearer. In any case, expansions were needed at some point in the bot's development, as they are mandatory in longer games. In hindsight, it might have improved gameplay performance more if other features were pursued instead, like combat prediction or advanced units.

Chapter 12

Conclusion

The agent has been a success with a 79.23% win-rate in mixed-division on SSCAIT, playing Protoss rush strategy. It is contesting a few others for the top score in the student-division. The agent is capable of efficient resource gathering across multiple bases, managing production and executing an early rush strategy. It can complete build-orders, such as an early-expansion opening. The agent gathers troops before attacking the opponent, and retreats when outnumbered. It uses a simple heuristic to predict combat outcomes in determining its tactical choices. The project has met its goals of a competitive bot for StarCraft.

The bot will be submitted to the 2015 AIIDE, CIG and SCAIIT competitions and further development is planned at least until the competition deadlines.

Bibliography