# 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 M.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


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 bot was developed with the agile development process. The implementation and design was iterated upon during short week-long sprints. After each there would be an evaluation of the bot's current status and new goals would be identified for the next sprint. A backlog kept during development was consulted when determining the status and goals in scope of the whole time-plan. There were a few set milestones throughout development, usually a month apart. They determined some requirements the bot should satisfy in relation to its performance at specific points in the development. The milestones served as a medium between sprints and the overall goal, ensuring the bot improved and did so consistently.

The first milestone were attaining the minimum viable product. The requirements for this was a bot capable of defeating a passive opponent, implying means to gather resources, build units and attacking. This would imply the bot had a non-zero chance at winning a round, making it an actual competitor.

## 2.1 Testing

Testing was done in two steps. First the robustness of the implementations was tested by playing the bot with random built-in StarCraft opponents and

tournament maps. This was to ensure the stability of the bot and that the implementations worked as required. The second step was uploading the bot to the SCCAIT tournament page, where the bot would play against the other official bots. This would serve to test the strategic integrity of the bot, and was done rarer than the first step. Because it is more difficult to retrieve data from rounds on the SCCAIT, the first step was needed to determine crash causes and such technicalities.

The opponents on the SSCAIT should be at least updated since that last tournament, and would therefore be viable candidates to test the bot's strength. It is required to make ones bot open source if it is to compete, so there is no reason to keep the newest bot version a secret beyond a year.

Chapter 3

# StarCraft

*StarCraft* is a *real-time strategy* game released by Blizzard Entertainment in 1998. Players take the role as one of three different factions, the *Terran*, *Zerg* or *Protoss*. Controlling structures and troops, they seek to expand their economy, technology and army until they can eliminate all opponents. The expansion *StarCraft: Brood War*, also released in 1998, increased the rooster of units available among other things.

In the multiplayer mode, players start with a single resource depot structure and a few worker units. They must from there destroy all the opponents' structures. The matches happen on an array of different *maps*. Players must deal with imperfect map information as their visibility is limited to their units and their immediate surroundings - the rest is shrouded in the *fog-of-war*. With a large amount of different units and buildings available, the players compete by adapting strategies to counter their opponents. There is no perfect strategy capable of winning every game, as the decision space is nigh infinite.

Chapter 4

# Related Works

Bots in StarCraft

## 4.1 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.

A very important part of StarCraft strategies are the openings. Much like chess openings, StarCraft has a wide array of detailed steps to take in the first few minutes of a round.

Usually either of these strategies are used during the early game. Later on, players will have more resources to pursue multiple goals and a more balanced overall strategy. This takes form in StarCraft by players attempting to gain a long-term economic advantage, done by controlling the limited resource locations. While early game can be decided by a single battle, later stages require multiple.

## 4.2 Bots in StarCraft

Usually bots focus on only one strategy. Some of the more advanced bots identify whatever opening strategy the opponent is using.

Few bots reach the late-game content of StarCraft. Those that do usually focus on the incredible precision and speed which the bot can play compared to a human.

### API

*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 agents 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 inac-

cessible. 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 harvesting 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.

As an additional note, BWAPI divides the map into *regions*. These contain satellite data about the borders to neighboring regions, included resources and base locations. They are important because the regions are separated in *chokepoints*, which are either ramps or bottlenecked passages. Usually a region only contains a single base location and resource cluster, however some regions contain multiple clusters with overlapping base locations. Because of these qualities regions can be used to mark occupied locations by players.

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. 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.

## AI Competitions

Different competitions and their descriptions.

Competition rules: Setup, limits, goals.

Ties impossible

This bots involvement/predicted involvement.

Chapter 5

# Agent Design

In this chapter the agents macro-strategy will be described as well as its overall architecture and implementation design.

TODO Intro.

## 5.1 Overall Strategy

Before any strategies for mid-game or late-game could 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.

The bot will only focus on a single race. Even though Protoss are the slowest of the three, they also have the strongest and easiest to use early troops, so this is the race the bot will use. While the main AI challenges between races are shared, gameplay elements differ quite a lot.

## 5.2 Overall 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. A superior can command a subordinate module, however only as a black box. 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 datastructure 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 modu-

lar 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 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.

TODO Hierarchy diagram + description.

In the following chapters, we will describe the different groups of modules in the hierarchy and the problems they solve. While the modules are individually separate, some clusters of them are also separate. These can be separated in chapters as few of the modules has any connections between. The order of clusters is bottom-up so as to follow the chain of command. These groups are *resources*, *production*, *information*, *combat* and *strategy*.

Chapter 6

# Workers and Resources

This chapter covers the basics of gathering resources in StarCraft with a bot.

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. *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 3 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 secure more supply is building the race's *supply structure*. Each of these add some amount of supply to the total, 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 reserve more supply than their total, 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 structure is the *pylon*.

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 *TaskMaster* module contains workers at a single base location, and records their current tasks. It can return all workers with a specific task. Each base location is a wrapper *Vassal*. The *Landlord* module contains all vassals. None of these three modules have any agency, and only respond to function calls from

superior modules.

Gatherer module accesses the Landlord module and with it, all the subordinate TaskMaster modules. Gatherering must be contained in a single superior module rather than within each Vassal, as gathering is not necessarily uniform across all bases. This also lends greater control over gathering, as superior modules can control a single module instead.

## 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 and buildings, including the architecture for its management. First we explain the StarCraft terms of units and their production.

A *Unit* is any player controlled entity, building or not. Some spell effects, like the Terran nuke, are oddly enough also classified as units but they are neither selectable nor controllable. The term unit rarely includes these however. In this paper we sometimes refer to non-buildings as units, which should be clear from context. Buildings are sometimes called structures, usually to avoid confusion between building as a verb and a noun.

*Production* here refers to the player controlled act of creating a new unit. Within StarCraft, producing a building is called *building* and *training* if it is a non-building. 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 buildings are placed in the world when they are built, while non-buildings are hidden within their constructor. When a non-building is complete, it appears at the nearest free space around its constructor. A unit consumes resources upon production, including supply requirements. A constructing non-building can be canceled for a full refund.

BWAPI notifies the AI module whenever a unit is created or completed in

separate events, which can be used to update internal data-structures.

In the following sections we first briefly describe our location in the hierarchy. Then we begin with training and construction of non-buildings, moving on to building and constructing 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, the Architect is not since it requires workers from the Landlord module. Both are superior to the Accountant.

## 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 $n$ total trainers and $m$ matching trainers the final time for training is then $log(n)m$, 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

Ones 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 buildings 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 buildings 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 building, 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 buildings, 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 agent 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 buildings 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 of buildings 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 buildings 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 buildings 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 agent 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 agent 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).

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

## Commanding Builder

Recall the `TaskMaster` module from chapter 6. Since all our workers are separated in base locations, we can easily pick a worker from the closest base location. Usually our buildings are placed close to a base location, and in case of expansions we pick a worker from the nearest base. Additionally, the task master marks the jobs of all workers, so to avoid interrupting other jobs it picks a

worker from idle or gathering groups (in that order). Finally, the agent searches through the groups until it finds a worker that is not carrying resources. This becomes the builder.

The operation time is linear to the amount of workers, as they are usually gathering resources. 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.

Finally, we order the worker to build the construction. A player cannot build in fog of war, therefore the worker should first be moved closer to the target location. When it has been revealed the builder is commanded to build the structure.

## Monitoring Structure

While the structure is constructing but not finished, we need to store it somewhere in the agent. The agent would need to know which structures is soon available, otherwise the events which spurred the construction of the structures might be repeated. There is a lot of frames between scheduling the structure and actually receiving it, where unexpected problems could occur.

If at any point the structure is unable to be completed, it is desirable to cancel the order rather than repairing it. This is because the unexpected event which canceled the structure might have changed the overall strategy and remove the need for the structure. This way, the meta strategy of when to build structures are moved upwards in the hierarchy, reducing independent AI in this module and making it more accessible by other modules.

Cancellation of a build order might occur if for example the builder is destroyed or the build location is invalid.

Both Zerg and Protoss structures auto-construct while Terran require a worker. So the agent only needs to keep a dictionary of all structures that are constructing. While the structure has not yet been built, it can only be identified by its type as no unit exist yet. This implies the need of a multi-set, assuming we want to be able to build multiple structures of the same type simultaneously.

The current implementation spends logarithmic time to query, insert and remove new build orders and constructions, but this could be improved to constant with hashing. Both containers are linear in size. Usually a player is not building very many structures at the same time so these improvements are not important.

To ensure tasked workers are building the required structures, the data structure has to be updated every frame. Units have a bad habit of canceling commands in some cases and workers especially do this by retreating from combat. It might also be the case that a worker was given a command in the same frame it was tasked to build, in which case it could not receive the new build command yet.

Since refineries are built on top of geysers, they never trigger the event `create` but `morph` instead. Additionally, when they are "completed", no event is triggered. Completion checks are required every frame to monitor the construction of refineries.

The building, construction and monitoring of structures is handled by the module `Architect`, which also include acquiring workers and placing buildings. The Architect is an independent manager, since it has to command the workers. It has carefully been low in dependencies, to keep it low in the module hierarchy. The building map used to avoid collisions and preserve distancing is contained in the `BaseManager` module. The `Accountant` module is updated with the current schedule of structures by the `Architect` which is queried by other modules. Monitoring refineries is done in a separate module called `Morpher` which handles all morphing units.

## 7.4 Building Expansions

*Expansions* are additional depots beyond the starting one, built at other resource clusters. One of the most important concepts in StarCraft, expansions allow players to harvest more resources, more efficiently than with one base. These become necessary as the match progresses, especially since the start locations inevitably run out of resources.

Every tournament legal map has a specific expansion called the *natural expansion*. This refers to the closest base location to a player's start location and is easily defended compared to other expansions. It usually has less resources than other expansions to compensate.

As noted in the resource chapter, it is often profitable and usually necessary to expand resource harvesting to new locations. To avoid transporting cargo all the way between regions, players have to build resource depots near resources. If the internal data structures behind building structures is limited to regions it may be challenging, and it is in our case.

To build an expansion we need the location and the resources.

## Expansion Placement

There are a few things to take into account when expanding. Obviously we need to expand into a location which is not occupied by enemy forces, especially not if they already have a depot at the desired location. Additional factors are proximity to existing bases, defensibility and resource quantity. Expanding to the natural expansion is usually always the first choice, as it scores high marks on both proximity and defensibility. Usually, the richest base location is in the center and is difficult to defend.

When the agent looks for the next expansion location, it recursively searches neighboring regions from the starting one. The first region which is not already occupied is picked.

## Building Depots

When BWTA analyzes a map, it marks optimal base locations, where a depots distance to a mineral cluster and geysers are minimized. Usually each region only contains one base location, at least this is the case for all the AI tournament maps.

Building an expansion is then handled by ordering the depot construction at the found base location with a worker from a nearby region. Currently, the agent always picks a worker from the main base, which is not always optimal. A better solution would be recursively searching neighbor regions' base locations in a priority queue, visiting shortest base-to-base distance first. Since workers are approximately at the same position as their related depot, it would find the optimal worker source.

While the `Architect` handles building the depot like any other structure, the `Settler` module contains expansion logic and orders the construction. It is not an independent module and does not decide when to expand.

Chapter 8

# Information Management

One of the challenges in StarCraft is the imperfect information players have of the world. It is necessary to extrapolate the strategies of opponents from a few sightings, and regular scouting is mandatory. The player with the most perfect information can make the most optimal decisions.

In terms of bot development, there needs to be database systems for tracking information, and there needs to be a scouting AI.

## 8.1 Tracking Units

Initially, two things must be recorded for each enemy unit: position and type. The former is obviously needed to track movements of the enemy army to detect proximity of ones own army, and to record enemy base locations. The latter is needed as some units can change their type, such as the Protoss Archon. As an odd effect of the StarCraft engine, the Siege Tank changes types when it goes into or out of siege mode.

This was easily implemented with dictionaries with units as keys. Every frame, the bot updates all stored values. New unit are added based on events, as a

newly revealed unit calls the `discover` and `show` events. The implementation uses logarithmic time on queries, additions and removals, but this could be improved to amortized constant time with hashing.

As a curious addition, geysers behave very oddly in the StarCraft engine. When a refinery is built upon a geyser, it actually transforms the geyser object. If the refinery is destroyed, a new geyser object is created. While all the initial geysers are known to players, new ones are not. This makes the tracking of geysers and refineries a bit difficult, as geysers must be checked every frame, and the new geysers must be detected.

The `Archivist` keeps track of all opponent movements. `Geologist` keeps track of current known geysers in the world.

## 8.2 Scouting

Scouting is important throughout the entire duration of the game. Early on, players need to know where the opponent is, what faction they are playing and which strategy they are employing. Later on, observing the enemy army size, unit types, base expansions and tech level is important to counter strategies.

The opponents faction is important when deciding on opening strategies. The factions behave very different early in the game. Openings are a decisively slower when one has to defend against every possible attack. Only if the opponent picked random as a faction will it be hidden. No units are shared across factions, so the first unit discovered will reveal it, which usually is their scout or main-base.

### Initial Scouting

The map terrain, along with resource locations, are completely revealed at match start. This includes all possible player start locations. Scouting for the opponent base is then just going through all other start locations. Maps used in competitive matches are usually no larger than four-player size, meaning there are four possible start locations. A player then has to take into account that they might have 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 might be followed, revealing your own location. Harassing enemy workers can put a dent in the enemy economy, even if none of them die. Simply by attacking enemy workers forces the opponent to pull two from gathering to defending. From there, the harassing scout can retreat until it is no longer chased, or lead chasing workers around while attacking passive gatherers. If the scout manages to kill a worker the opponent will fall behind in economy. However, compared to competitive human players, bots are usually too inefficient to take full advantage of this, but every bit helps.

There are additional uses to the scout however compared to harassing. These must all be considered in the grand strategy, as they involve spending resources. These strategies include proxy bunker, photon cannon, barracks or gateway, manner pylons or gas stealing. Proxy structures involve building right below the enemy ramp or even inside their base. The usual distinction here is whether the structures keep the opponent in with defensive towers or rush attack with front line troop producers. Manner pylons are used in conjunction with these as Protoss, where the required pylon for forward bases are placed within the enemy mineral line, blocking and possibly caging enemy workers. Gas stealing involves building a refinery on the enemy geyser, blocking gas harvesting and forcing tier one unit use. While proxy troop production has not seen much use in bots, both gas steal and proxy towers has been used for varying effect.

### Implementation

Usually build orders include specifically when to send out a scout. In case of Protoss, the scout is often a worker that has just warped in a structure. This would however require a build-order system capable of containing other elements than builds. A simpler solution which is used here is sending a scout when a specific supply limit has been reached. This is noted in build orders, allowing for only slightly inaccurate build-order implementations.

When picking a scout, the bot searches through different worker groups. First the idle, then the mineral miners and finally the gas harvesters. When it comes across a worker not currently carrying any resources, it assigns it as a scout, removing it from the `TaskMaster` in the mean time. Contrary to building or defending, scouts are assumed to not return, so it can safely be removed from the local worker pool and harvesting. If the scouting manager is unsuccessful in finding a worker, it retries next frame.

While we have a scout and do not know the opponents position, we pick an

unexplored base location and move the scout there. A tile is unexplored if it has never been revealed for the duration of the game. Thus, our home base will not be considered for scouting, and once the scout reaches the destination the tile will be explored, removing it from the possible scouting locations. Exploration is handled by BWAPI.

This implementation scouts as long as no enemy buildings are known, extending its use into late game. Usually fast or cloaked units are used to scout later in the game, but it is not necessary.

Chapter 9

# Combat

Combat in StarCraft can easily become the most complex part of the bot development. Numerous studies has been done on just small subsets of combat scenarios, where we are given specified units and enemies, but even these just scratch the surface. Like the macroscopic strategies in StarCraft, there is probably 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 excel against humans, since they can easily command different units in complex ways.

The `ArmyManager` is the combat parallel to the `TaskMaster`. It contains all fighter units controlled by the agent and their current assigned `duty`, same as `tasks` for workers. A fighter unit is either idle, moving towards the opponent, attacking or defending.

## 9.1 Attacking

A player will almost certainly expand to their natural expansion first and 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. All this together we can assume the opponent has only one base for the duration of the early game, which is where the bot mainly operates.

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 to the enemy base, is also the shortest path from the opponent's base to yours. So assuming each player has only one base and both players will use optimal pathfinders, we can model the map as a straight line between the bases. Ours and their army will travel only along this path, and there is therefore no case of armies moving past each other without collision.

As a rushing bot, it is in our favor to move our troops as close to the opponents base without confrontation before they are ready. This pressures the opponent while also providing some scouting, and allows us to attack as quickly as possible. 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` manager. Predictions are handled by the `CombatJudge` module, which given a set of units outputs their strength value.

## Prediction

Predicting combat is very difficult in StarCraft. Even beyond the numerous factors in combat, the optimal command of ones troops is very Dependant on the command of the opponents'. 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 it has proven to be close enough to the correct result.

It is assumed the opponent has only a single army which is always gathered in

close proximity. This is not very useful in late game where flanking maneuvers and harassment is viable tactics, but it is sufficient for early game. The worst case is we will overestimate the opponent strength, which is a much better case than underestimating it. It would be possible to detect these individual groupings however with a cluster detection algorithm.

TODO Prediction heuristic.

## Targeting

Usually RTS games has a command called *attack-move*, and StarCraft is no exception. A unit executing this command will move towards a target or location, but will attack any enemies along the way. In StarCraft the unit will prioritize units that are attacking it before others. So beyond auto-targeting nearby enemies, this command also carries a simple prioritized targeting.

By targeting the enemy base location or structures with this command, we have simple prioritized targeting. We avoid targeting the opponents units, as our army might be lead astray by a decoy. The most important targets are the enemy structures since a player loses when they have no more of them.

Since we already have a dictionary of enemy buildings, the agent just picks one arbitrarily from the list. Assuming the opponent has only one base, the army will attack the foremost building regardless of target since they are attack-moving. Since the natural extension is usually the only exit from the main base, this assumption is valid while the opponent has two or fewer bases, which is true for the entire early game.

Some bots go beyond the attack-move prioritize. A useful alternative is attacking the unit with highest health to resources cost ratio, targeting damaging units first. This will ensure an attack deals a size-able blow to the opponent's economy, trading resource loss as favorably as greedily possible. A similar one is attacking units with highest health to damage ratio. There is a lot more to prioritization. This is especially true when considering large scale armies composed of different units, where counter-play between individual types are important. Beyond fighting the enemy army, the order when attacking is usually opponent's army, then economy and then unit facilities.

## Troop Rendezvous

As all troops are sent off immediately from production, its necessary to avoid sending them into the enemy base one at a time. The strength of an army increases faster than linearly with its size, as troops will die slower while enemies die faster. This means our troops get more time to deal damage while their troops get less.

At some point troops in transit will be within some danger distance of the enemy base, where they will risk being attacked. This will be the minimum distance from the opponent where they can safely gather. By only counting these arrived troops in our combat predictions, we can tell when enough troops have gathered to attack. Currently, the bot only counts units currently within this danger distance of an enemy unit as arrived, that is, it does not record arrivals. Testing proved the battlefield shifted too much to record arrivals and pausing them until attack, which lead to troops being scattered across the battlefield rather than actually gathering. This solution proved sufficient however for the few and small units the bot uses. A better solution would be needed with a larger army composed of larger units. This could be solved with a cluster detection system, to detect when ones army is actually gathered.

## Fighting

There is a lot of depth in unit combat in StarCraft. Maneuvers and strategies involving the precise commanding of units is called *micro*. There has been multiple studies in this area in specific scenarios.

The bot uses the Zealot unit which is a very efficient unit but also a very simple unit. All we need to do attack move each zealot, which will then auto-target to the nearest enemy. This is surprisingly effective against other units, mostly because other units are a lot more complicated, especially if they are ranged. There is a lot of room for improvement, but most of the bot development focused on the macroscopic strategy than the micro.

## Retreating

Sometimes because of incomplete information, the combat prediction turns out to be wrong and the opponent had in fact more units hiding the fog of war. In some cases, it is optimal to retreat. Depending on how tangled ones units are

with the opponents, retreating becomes a less desirable option as some units will die without fighting back. However the opponent would not lose very much value in troops by staying in combat, compared to how much you would lose by retreating, then it is the better option to retreat. Calculating this is not easy however, and would also require a strong combat predictor, something that we don't have access to.

The bot will therefore not retreat any units that has been assigned combat, only units that are in transit. The logic here is that we don't know how tangled our combat units are to the opponent, but the transit units will usually be outside enemy range, such that we lose no troops to retreat.

Retreating units move towards the start location, which will path them outside the enemy base. This is usually the optimal solution barring cloaked or flying units, or enemy units that need to be avoided. Fortunately, its rare that the opponent has some troops that can be avoided if they are on the retreat path, and the bot does not utilize cloaked or flying units. A solution to these problems however would be using a custom pathfinder using heatmaps of enemies or potential fields.

## 9.2 Defending

Sometimes the enemy attack first, break through our attack or somehow avoid our army. In this case the base must be defended. At other time, the opponent might have sent an early Zerg rush or a worker as scout and harasser. In these cases we need to pull workers to fend off the attackers.

Although they have not been implemented, turrets are effective when playing defensively. Usually cost-effective compared to mobile units, they can easily repel early assaults. The problem however is finding proper locations, as the bot has to predict the point of entry to defend it. This is coupled with the logistical problem of blocking the entrance or bottlenecking it.

The `Defender` manager handles the enlistment and commanding of defenders.

### Scrambling Defenders

The problem with designating defenders is related to retreating. We must avoid drawing troops away, that would be better used attacking. This is difficult to

asses. It is obvious however, that everyone in a region under attack, should defend it. This will not interrupt desired combat or cause undesirable behavior. If they are already defending, nothing is interrupted, and otherwise they are better served defending.

As it turns out from testing and because of the straight-line assumption, we very rarely have any units left beyond our base if the opponent is attacking, since they would already have fought and died. Those that are beyond however, continue to assault the opponents' now undefended base, which is desirable since they are too far away to actually help the defense. The current simple implementation therefore proved to be sufficient.

## Militia

Sometimes there are not enough troops in the base. This usually happens before an army has been aquired or after it has died. In this case, a common strategy is scrambling all the workers for defense in hopes of surviving. Worst case all workers die and the player loses, but that would have been certain without the militia anyways.

Workers are tasked to defend until `CombatJudge` predicts they win. If its an early attack, usually a few workers, but later in the game it is almost all workers.

Chapter 10

# Macro Strategy

This chapter concerns itself with how we implement the overall strategic reasoning in our bot. We have already gone through the basics of strategies in StarCraft, and all the basic functionalities has been implemented in the bot. Now it must all be put together in a cohesive form to achieve a strong AI.

Initially the bot functioned purely on a greedy principle. Concerning a specific set of units, we attempt to produce them if possible every frame. This is possibly the fastest and simplest AI to implement, but it has proven to be an effective approximation in some algorithmic fields. As the bot was developed additional systems was implemented, but the core auto-pilot remained greedy.

The overall strategy of the bot is rushing. It needs to be effective in economy early on and build troops as fast as possible. It sacrifices long term advantages for an early game attack. If the initial attacks are successful, the bot would gain even stronger long term advantages by hindering the opponent, and would be able to transcend into mid-game tactics.

While one could focus on a niche strategy such as proxies or such, as explained earlier these strategies are strongly countered by some specific strategies. They are high-risk, high-reward, but only work in a small subset of match-ups. So even though they would be effective in these match-ups, we would need to implement other solutions for the rest, where a not insignificant amount of them

would require an advanced bot beyond these 'cheese' strategies. Therefore, we could just as well implement an advanced bot and cover all match-ups. All-in would be such a niche-strategy against humans, but bots have proved to be very ineffective against this strategy. Additionally, the all-in strategy is a more aggressive version of rushing, so it could easily be transformed into the latter without any large scale implementations.

All combat decisions are made by the `Strategist` manager, while economy decisions are made by the `Economist` manager. These are disabled by the `Despot` manager when executing a build-order. The `Despot` module is the highest in the hierarchy, beyond the core AI module receiving call-backs from StarCraft.

## 10.1 Economy

TODO intro

The `Economist` manager moderates worker production and expansions, unless it has been disabled by the `Despot`.

### Workers

Generally it is advised that a Protoss player keeps producing workers constantly until late-game, but some openings might temporarily stop worker production. Therefore, if we are not at the desired worker amount in a base, we attempt to produce a new worker with regards to resources and whether a worker is already in production.

### Expanding

An expansion secures a more permanent economic advantage, as a player secures both more resources and gathers them faster. More expansions also secures map-control, as the player will have more presence in the map and control more resources.

Deciding when to expand is both difficult and pivotal. It takes time and resources and leaves the new base pretty vulnerable unless there is a standing

army. A player with a stronger army can safely expand while guarding it, securing his advantage or at least keeping up with the opponents expansions. Without the stronger army its a risky maneuver, but might still be viable or even imperative. At some point the resources in ones current bases will be depleted, leading to a need to expand. If a player expands before having saturated his current mineral fields, its called a *fast expand*, sometimes used as an opening. This is very risky but with a very immediate payoff in economy. The contrary formula, keeping a single base until mid-game or alike is called *one base play*.

#### Implementation

It is only clear that an expansion is viable when all mineral fields have been saturated. Before this, it is both risky as we sacrifice current resources for later economy, which requires both a strategic reason and foresight whether it is viable. By expanding when we have saturated our mineral fields, we risk expanding too late, but ensure a consistent worker production throughout the game. Expanding too late is better than expanding too early; an early expansion will waste resources or possibly result in an immediate loss, where a late expansion simply wastes an opportunity.

Since expanding is a costly affair, the bot needs to save up resources. Therefore, the expansion is queued in the build-order, thereby pausing other productions until the expansion is ready.

TBD Implement better expanding, avoid opponent regions and avoid if in combat.

The bot can perform a fast expand opening, but it has not been very successful compared to its usual one base play. The difficulty lies in detecting whether a fast expand is viable, which is very much Dependant on enemy openings. It was decided to focus on the safer opening instead as it is a more versatile strategy.

TBD Evaluation of the auto-expander vs. one-base bot.

## 10.2 Combat

TODO Combat intro.

TODO Building troops.

TODO Counter Strategies.

`Strategist` order the production of troops and troop facilities.

### Troops

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.

Currently the bot just mass produces zealots whenever it can in greedy fashion. As producing zealots is the ultimate goal of the bot, this is sufficient.

TBD Additions: Dragoons, upgrades.

### Facilities

It was found through testing that a single saturated start base can handle around four gateways constantly producing zealots. This gets more difficult to asses with expansions, and it was unfortunately also found through testing that greedily building limitless gateways is not even close to optimal.

The current limited solution is building seven gateways at max. Usually if the bot reaches a point where it needs more, it has probably already lost due to lacking in technology. A scalable solution has not been implemented, but to be optimal it would require some estimation of resource input, along with estimated need for profit for further development of base and technology.

## 10.3 Build-orders

Openings in StarCraft are usually detailed like a *build-order*, which is very much like a chess opening. Because of the huge decision space, even humans need

some guidelines weathered by experience to perform well in the beginning of the game. The optimal build-orders have changed a lot across the years, shaping the meta-game of StarCraft. As new strategies were found, new openings had to accommodate new possible opponent strategies.

A build-order is a list of units that must be produced in a specific order, sometimes after certain events. There is no timer in vanilla StarCraft and using one is considered cheating in tournaments, so items in the build-order are to be constructed when specific supply costs has been reached. Between each item in the build-order, the player must produce workers until then next supply limit has been reached unless specifically otherwise instructed.

This is akin to executing a queue of orders. One could make a queue where the next item is executed if a specific supply is reached while automatically building workers like a human player. The easier solution for bot however is of course manually filling in the gaps with worker orders, such that the supply limits will be reached at the correct times (assuming no workers are lost). Therefore, the solution in the bot is simply having a queue of units. While the queue is not empty, we do not execute any higher-order manager AI that would otherwise automate production. When the queue becomes empty, we resume automatic production.

The benefit of this solution is the ease of adding new openings to the bot. Although the general autopilot AI is the same, different openings will vastly change the strategy and outcome of the game. Proper openings are 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. What if the bot loses a worker or a structure? It is possible that the current build-order is no longer optimal, viable even. We avoid this, by making sure the build orders are short, such that the opponent could in no way interrupt with any units other than harassing workers.

Additionally, this solution could be used to pause other production in mid-game. Since the bot operates in a greedy fashion, it can be difficult to build high-cost units. The solution is to enqueue the unit, pausing all other production until the unit has been created. This is used in the agent when expanding, where the depot is queued. This solution is risky however, as the bot will not consider removing the depot from the queue under any circumstance. It does not seem to cause any problems however, as the bot will reach the amount of resources required very quickly, resuming normal production again.

Since the build-order is a queue, executing it is constant time, and enqueuing the build-order is linear. The build-order is never changed during gameplay.

The `Planner` module handles executing build-orders. The main module `Despot` handles queuing build-orders and disables other managers while `Planner` is executing a build-order.

Chapter 11

# Results

Across 780 games the one-base version garnered 527 wins, netting a 79.23% win-rate in a mixed-division on the SSCAIT tournament page. Unfortunately there is not a separate statistic for student-division only, but it was usually contesting the UAlberta bot for top student bot. The UAlberta bot usually wins against this bot however.

TBD expansion bot results.

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 produce more advance units and win through technology. The final additions to the bot were attempts to combat the turtle strategy, however it was never effective enough. It is the natural counter to the rush strategy.

The bot did not reach a very complex higher-level AI. The focus was creating a competitive bot, so development focused on areas that improved performance the best. This is expectedly in low-level technical features such as using the command interface and handling units. The opening-moves and attack commanding is the most advanced AI the bot can boast. While as a whole it performs somewhat intelligent, the lack of adaption means it is really not.

Even though the bot has now grasped the basics of StarCraft and RTS gameplay, there is probably a long way yet before any high-level AI implementations would be effective. Its still does not use most of the Protoss units and buildings and does not research upgrades. The next steps to take would be improving its combat predictions and map awareness with regards to enemy unit placement, such that it could understand separate bases and armies. From there, counter strategies would be necessary, especially against flying and cloaked.

As an additional note, the bot has problems pathfinding in the maps *Circuit Breakers (4)* and *Heartbreak Ridge (2)*, where neutral structures block terrain passages. The bot uses the built-in pathfinding, which cannot path around these kinds of obstacles. In Circuit Breakers, this will result in a loss every fourth game, as it is only an issue in one of the four start locations. In Heartbreak Ridge, it prohibits the bot from expanding to a certain base location. This problem was never fixed as it would require the implementation of a custom pathfinder which would be too time consuming. Interestingly, the built-in bots suffer the same issue.

The bot will be applied to at least to the 2015 AIIDE tournament with a submission deadline the 15th of august. There is however a limit to 30 submissions, so it is not certain the bot will appear in the tournament.

Chapter 12

# Conclusion

The bot has been a success with a 79.23% win-rate as Protoss in mixed-division on SSCAIT. It handles basic StarCraft maneuvers such as harvesting both types of resources on multiple bases, building structures and executing a simple rush strategy. It can execute build-orders, including early expansions. When in combat, it can do basic tactical decisions such as gathering troops and retreating. With a very simple but somewhat effective combat prediction heuristic, it will attack only when ready. The project has achieved its goals of developing a competitive bot for StarCraft. While the bot is simple compared to humans and older bots, it is still above in complexity and effectiveness than the average.

The bot will be submitted to the 2015 AIIDE competition, and will probably be further developed at least in this year in preparation for later competitions.

# Bibliography