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.

The thesis deals with ...

The thesis consists of ...

Lyngby, 30-June-2015

Carsten Nielsen

Acknowledgements

I would like to thank my....

Contents

Chapter 1

Introduction

StarCraft, AI in RTS.

Goals, challenges, programming language.

Other projects and agents.

1.1 The "separate document"

Original project plan.

Revised project plan.

Brief self-evaluation.

1.2 Starcraft: Brood War

Game description: premise, goals, races, units, challenges.

Imperfect information.

For this bot I focused on Protoss. The main AI challenges between races are shared, however specific gameplay elements differ quite a lot.

BWAPI

BWAPI description.

Loading modules.

Imperfect information interface.

Regions

BWTA and SparCraft description

AI Competitions

Different competitions and their descriptions.

Competition rules: Setup, limits, goals.

This bots involvement/predicted involvement.

Short desc. of Starcraft 2 situation.

1.3 Process

Agile development.

Short iterations.

Testing in SCCAI.

Opensource competitions.

Documentations, strategy sources.

Chapter 2

Datastructure Design

TODO Intro. The imperative when designing the bot is to lessen the decision space as greatly as possible.

Before all that, we need a gameplay based goal as well. Since this bot will not reach a great level of complexity due to time restrictions, we can design the data structures with the overall limits of strategy in mind, to speed up the development process.

2.1 Overall Strategy

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. In most RTS games, the trinity of strategies are *rush*, *boom* and *turtle*. Rush 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. Closely related, a player could instead focus on developing economy and upgrades to quickly reach late-game units. If successful, the player will have superior units and easily beat the opponent. The problem is early

game defense will have to be sacrificed, and puts the boom player at risk for enemy rushes. Finally, turtling players focus on early defenses, repelling enemy rushes. Usually in RTS games, defensive structures are more cost-effective than mobile troops, putting the defending player at an advantage. The goal is to use less resources than a rushing player to counter the rush, and then use the economic advantage to win. Turtling players will lose to boom however, as their defensive structures are wasted against non-rushing strategies, and will be at a disadvantage against the boom players improved units.

It is also possible to instead implement some niche strategy, like proxy gateways and the like. The problem is these strategies are hard countered by some general strategies. This means it would not have a chance against some specific other strategies, making it somewhat unsustainable and not very time effective to work on.

This bot will focus on being strong at early game. The logic here is that if it loses early on, there will be no mid- or late game. Therefore, the bot must master early game first before others. Simplest solution is implementing one strategy, which can be expanded into mid-game if the round drags on. Both boom and turtle strategies seek to push the game 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 infantry and send them to attack. Additional features would be upgrading, building more advanced units or expanding.

2.2 Overall Design Paradigm

The problems a bot faces are easily divided into smaller, isolated problems. Its therefore easy to build the bot out of individual modules, each solving some non-overlapping subset of problems. In this paper, managers are modules which has some sort of AI behavior, and is updated every frame of the game. Contrary, a non-manager is passive, only active when other modules need it or on event callbacks from BWAPI. The benefit is modules are individually easily replaced or refactored while the whole still works, as they are non-overlapping.

Behind every module there is an additional design rule. We would rather want to do the right things ineffectively, than doing the wrong things effectively. It is a trivial axiom, but important when implementing new strategic resolutions. We would like to prioritize the minimization of times the bot applies strategies that are not appropriate, rather than possibly applying strategies at appropriate times. Slow and steady wins the race, and this is a safe approach to developing

the bot. The idea is that the bot is constantly added to with features, rather than current features being ironed out and odd behavior removed. It streamlines the development process, and eases evaluation of different features.

The UAlberta bot by Dave Churchill used an interesting design paradigm. The modules in the bot are in a arborescence graph structure. There is a root node, the primary AI module loaded into StarCraft, which has exactly one path to each other node in the hierarchy. In other words, there are no cyclic dependencies. 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 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 just evolve upwards with newer modules easily integrated to the older. This structure can easily contain multiple sibling managers, separated by information and controlled by some higher order manager. Much like the military, the lower managers make smaller decisions with fewer resource such as building and gathering, and the higher managers 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 top of the hierarchy, where the root module is the final word on what the bot does. At some point all the choices have to be made in the final module which has all the information available.

An alternative design would be a single manager which makes all decisions based on some subsidiary modules. This structure can easily get confusing in priorities and the order of decisions, but it does allow greater freedom and even a dynamic order in which choices are made.

Chapter 3

Resources and Bases

In early- and mid-game, *economic advantage* is the most important aspect of StarCraft. This is when you have a higher resource input than your opponent, and happens when you focus more on resources than them, or they sacrifice long term advantage over short term. The point of this is if you eclipse your opponent in resources, you can perform worse in combat and still end up on top. You can send more units, recover them faster and expand economy quicker, retaining your advantage.

There are three resources in StarCraft. Minerals are mined from mineral fields which usually are in clusters of 8-12 by each base location. Everything you produce costs minerals, so it is the most important resource. They are harvested by your workers, which bring it back to a nearby resource depot in batches of six. Usually there are 2-3 workers on each field depending on the distance from the depot. *Vespene Gas* or simply gas is harvested from *refineries*. These must be built upon *Vespene Geysers* of which there are one or two by placed by each base location. 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, and a decision with major impact is when a player starts harvesting gas. The immediate cost of the refinery and lost mineral gathering is a liability against fast rushing strategies. The last resource, supply, is reserved by units. Every unit reserves some amount of supply which is released upon their destruction. The only way to secure more is building *supply depots*,

pylons or *overlords* respectively for Terran, Protoss and Zerg. Each of these add some amount of supply, which is removed if they are destroyed. A player can happen to use more supply than they have, but will be unable to build more units until more supply is acquired.

Only one worker can be active at a mineral field or refinery.

StarCraft has a built in worker gathering AI. Workers will automatically return cargo from resources (unless they are interrupted). When gathering minerals, the workers will move to another mineral field if the current one is occupied. When gathering gas, they will wait until the refinery is unoccupied and then gather.

TODO TaskMaster module

3.1 Mining Minerals

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.

TODO alternatives.

3.2 Harvesting Gas

Gas harvest is very much like mineral mining, however it is simpler. It is agreed upon that three workers are the optimal amount per refinery. The implementation reflects this, and is otherwise identical to mineral mining.

Bases have between zero to two geysers, but in no AI tournament maps are there more than one per base location.

3.3 Building Supply

A player can have a maximum of 200 supply. The simplest supply implementation is to build supply units whenever the supply limit has been reached while it is below the maximum threshold. This will however throttle production while the supply structure is being constructed, which is inefficient. The optimal solution would be predicting supply needed in the near future and interlace supply production with unit production such that all orders are completed the earliest. 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, and even then resource gathering rates must be predicted for optimal solutions.

A simple solution which is implemented in the current bot is to build supply units when released supply is below a set threshold. This threshold is somewhat arbitrarily set according to testing. A more dynamic solution would be to set the threshold to the amount of supply used if all production facilities built a unit, assuming the unit type can be predicted. This solution will build supply units that are completed before they are needed, but might not result the optimal amount of units completed at any given time (impacting economy with relation to workers). The worst problem, production throttling, is avoided however.

3.4 Managing gathering

Usually a Protoss player should not stop building workers until late game, but some openings require a temporary pause. Therefore the solution is just greedily building workers while no build order is being executed.

Two workers per mineral is the optimal amount. The third worker will have

diminishing returns and the fourth would have no impact. A third worker is optimal if the mineral is far away from the depot. Three workers on every refinery is the standard, regulated only according to need. Usually a refinery will be at max capacity, unless the player has lost too many workers.

TODO Multiple bases.

The Production chapter will cover how and where we build these expansions.

TBD Mineral to gas ratio.

TODO Maynard Slide

Chapter 4

Production and Expansion

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. This is allegedly because of time constraints, and when the term unit is used it usually does not include these. 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. Officially producing a building is called *building*; *training* if it is a non-building. The term production has been devised for this paper to refer both building and training, whichever is appropriate. Note that the act of building is creating a unit while building a structure usually refers to the entire process of making structures.

When a production has been ordered, there is a duration where the unit is *constructing* before being complete and usable. The event of a unit being created and starting construction is separate from ending it and being completed. An incomplete unit cannot execute any commands and a player will not receive any benefits like additional supply before completion. Buildings are placed in the world prior to construction while non-buildings are hidden, appearing at their constructor upon completion. A unit consumes resources upon production,

including supply requirements.

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 at higher rates than with only one base. Every tournament legal map has a specific expansion called the *natural expansion*. This is the closest base location to a players start and is usually easily defended compared to other expansions. It usually has less resources than other expansions to compensate.

4.1 Training Units

Producing non-building units is simple compared to building structures. The act of producing a unit is called *training*, however confusingly an incomplete unit is considered *constructing* as if it were a building. The steps can be described as firstly acquiring a relevant and available facility, and secondly monitoring the training.

Implementation

StarCraft has a simple system of who-constructs-who, as there only exists one possible constructor for each unit. All buildings refer to the relevant worker and all units refer to at most one structure. BWAPI has already implemented this functionality, so given an arbitrary unit type we can easily determine if it is a non-building, and if so who trains it.

TODO implementation.

The agents manager for training and monitoring units is called the *Recruiter*. It is very low in the module dependency hierarchy, while accessed by many other managers. It is not an independent AI, updated only through events. It contains all units capable of training other units. The implementation is generic enough to allow training any train-able unit in the game, even those trained by non-buildings.

4.2 Building Structures

Like training units, building structures is in two stages. The structure is built and then it is constructed. Contrary to training, structures require workers to be built, where the worker must first arrive at the desired structure location. All player structures are built by one of the three workers (except Terran extensions, but they are handled more like upgrades).

To build a structure we must first find proper placement, find an available builder and then monitor the structure until completion.

Structure Placement

There can be a lot of strategy behind the placement of buildings, and in special cases it is imperative.

Terran can build structures anywhere on the map. Zerg can only build on *creep* which terrain that has been transformed by specific nearby Zerg structures. Protoss can build the Nexus (depot) and Pylon (supply structure) anywhere on the map like Terran. All other structures must be *powered*, which is satisfied when they are within a certain distance of a pylon. A structure can lose power if all pylons nearby are destroyed, and will become inactive until new pylons are constructed.

The pylons are therefore important when placing new structures. The Protoss player must place at least one pylon in any area he wishes to build in and should distance pylons in a base to allow more structures. Fortunately, Protoss bases are usually compact enough such that one can place pylons arbitrarily and still manage to fit other structures in the region. It is important however to place at least one pylon in an expansion before other structures can be built.

Unless a specific building location is specified, the agent attempts to place the structure as close as possible to the main depot of the desired region. This is easily implemented, avoids some pitfalls and accomplishes some goals. All the workers in the region are usually at the mineral fields by the depot, and therefore will have a short distance to travel. Additionally the workers and depot are shielded since they are the furthest point in the base from all angles. The opponents' troops are forced to either destroy the other structures first or move through the bottlenecks of the base.

Even if a region only has one exit, it might be desirable to spread out all struc-

tures, as flying units can attack from any angle. Depots will always be placed in new regions at the location BWTA has marked. Refineries can only be built on-top of vespene geysers and are exempt as well.

Implementation

The implementation to place buildings is simple but expensive. The agent attempts all locations in the map in a spiral pattern around the depot, and returns the first location that is available. There are multiple factors when deciding availability. Obviously there must not be any units in the area and the terrain must allow the structure placement.

Placing buildings too close can block passage between them, especially for larger units. The structure hitbox is somewhat arbitrary in StarCraft, as different combinations of buildings next to each other will allow different units to pass through, even though it is tile based. This is because the structure hitbox usually does not cover its location completely. By placing buildings at least one tile from each other, we ensure all units can easily pass through. The agent does this by keeping a tile map of all structure locations inflated by one tile in all directions. If a build location overlaps any of the occupied tiles it is not available. Registering new buildings and querying this is constant time operations, but the space used is polynomial to the map size. It could be improved by only keeping a map for regions we have bases in, although not an asymptotic improvement.

We would also like to avoid placing structures between resources and depots to avoid blocking workers. This is simply done by drawing the smallest rectangle including the depot location, geyser(s) and mineral fields, and avoiding 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).

Aquiring and Commanding Builders

The naive solution in getting a builder is picking one arbitrarily from all controlled workers. This however might interrupt other jobs, pick a worker that is carrying resources, or pick someone far away from the build location.

Instead, recall the `TaskMaster` module from resource gathering. 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 Construction and Completion

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.

Implementation

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.

4.3 Building Expansions

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.

Locating Expansion

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 5

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.

5.1 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.

5.2 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 6

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.

6.1 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.

6.2 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.

TODO Edge of region problem.

TODO Alternatives: Vicinity check, Standing army, Pullback.

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.

TODO Anti-scouts / anti-harassment

TODO Alternatives: Worker relocate.

Chapter 7

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.

7.1 Economy

TODO intro, importance of economic advantage.

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.

7.2 Combat

TODO Combat intro.

Building troops.

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.

7.3 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 8

Results

Results from SSCAI sparring with different bot builds.

Overall appraisal of relative bot quality.

Thoughts on further development, where to go from here.

Tournaments to apply to.

Chapter 9

Conclusion

Agent status, focus on different areas of AI challenges that has been solved

Results.

Competitions.

Further study.