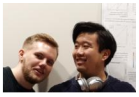
Methods and Strategies for AI Bot in StarCraft: Broodwar

Group 5

Minghua Lu
19981202
minghual@kth.se



Carl Lundqvist
19960902
carlundq@kth.se

Abstract

With e-sports getting popular over the years, there has been an incentive to create artificial intelligence that can beat humans, either to prove that AI is better than real players or for prize money. Using BWAPI, we can interact and write code for the real-time strategy game Starcraft: Brood War. We built an AI with a pre-existing bot template, StarterBot from UAlbertaBot. By adding basic but effective macro and micro tactics to an established build order with a few improvements, we created an AI capable of adapting to the current stage of the game and beating the default AI in the game. In a small tournament with other AIs, our bot achieved outstanding results, winning every game in the final round. Further work includes observing the enemy team's actions and allowing the AI to make strategic decisions based on them, which could make the AI able to defeat real players.

1 Introduction

In recent years, online games have gotten more and more popular. With the rise of e-sports, there are now tournaments with prize money of up to \$40′000000 [1]. One of the earliest popular e-sports game was Starcraft, created by Blizzard Entertainment in 1998. Starcraft is a real-time strategy game (RTS) where each player expands their base and builds military units with the goal of destroying the opponent's base. RTS games differ from abstract games such as chess and checkers in the fact that there are no turns; moves can be issued at all times, with no wait, regardless of the actions of the opponents. In addition to requiring real-time commands, the environments in Starcraft are large with many variables adding to the complexity of creating a useful AI. As such, solving this problem may bring applications to the real world, if not for the satisfaction of winning prize money in AI tournaments.

Using BWAPI [9] and the StarterBot from UAlbertaBot [3], we implemented an AI capable of keeping our team relevant in all stages of the game. Our AI followed a build order based on an early game strategy, with simple but effective tactics, both from a micro and macro perspective. The key feature to our AI's success is our local attack manager, which determined whether we should attack the enemy or fall back, based on the number of friendly units compared to the number of enemy units in a small area. By making sure that we only attack when we have more units, we always have the advantage in fights and managed to defeat the default Starcraft AI as well as several other AIs in a tournament. However, considering the complexity of Starcraft, there are still many aspects of our AI that can be improved, though the potential is there.

1.1 Simulation and rules

The ability of our AI will be tested in a tournament with four other teams. Each team will play all other teams twice in a one versus one game of Starcraft. All games will be played on the map Fortress and all teams will play the Protoss race. The first team to destroy all enemy buildings will be declared the winner of that game. Finally, the team with the most overall wins are the champions of the tournament.

1.2 Contribution

We have contributed with implementation of simple and effective strategies in Starcraft, notably checking for local advantages and acting upon them.

Our tactics are also applicable to other RTS games and even in real battles it is beneficial to take advantage of local strengths as our algorithm does.

1.3 Outline

The rest of this report is organized as follows: in section 2, we discuss relevant research and material that we thought was most interesting in the field of Starcraft AI. In section 3 our method is presented in detail and we focus our discussion on the parts of our method that is most unique and interesting. In this project we played games with our bot versus the bots of other groups, the results of this is presented in section 4 along with an analysis of the results. Finally, in section 5 we summarize the project and include conclusions as well as possible improvements.

2 Related work

In Incorporating Search Algorithms into RTS Game Agents [5], a general overview of Starcraft AI programming is given. The authors created the UAlbertaBot and StarterBot which is the starting point of our AI. Since there is no source code for Starcraft available, the paper also introduces a way to translate code to actions in-game with the use of BWAPI. Also, due to the dynamic nature of Starcraft and RTS games in general imposing computationally heavy problems [7], many AIs implement hard-coded strategies with predefined conditions. A hard-coded AI can still be very effective and able to react to a wide set of scenarios.

A good economy is key to winning Starcraft games. Build Order Optimization in Starcraft [4] discusses the importance of creating a certain number of units and different buildings in the shortest time possible, while taking into account the prerequisites of those units and buildings as well as available resources to the player. A search-based algorithm is used to find the best possible build order, which considers the end goal, the current game state, and a time limit. We will not over-complicate things and instead achieve this by simply following an established build order and adjusting it if needed.

Since the available resources in a base are limited, it is important to expand the number of controlled bases by building new buildings close to other uncontested resources. This adds another layer to optimizing build orders because expanding too early will delay other important buildings, whereas expanding too late will lead to resource starvation [8]. The timing of an expansion can be learned through deep learning [8] or hard-coded to follow a general rule of thumb for ease of implementation.

In addition to building a good economy, micro-managing the player's unit is also crucial. Sometimes, having a superior army does not guarantee victory. Figuring out the smartest and most efficient use of the army is important to ensure positive outcomes in smaller-scale battles. We will explore simple concepts discussed in AI System Designs for the First RTS-Game AI Competition [2], such as engaging in fights where we have a numerical advantage and focus firing on a single enemy.

3 Proposed method

Our solution to this problem was inspired by behavior trees, where we had our bot follow specific instructions that could change if something special happened in the game. In this section, we will give more information about the build order that we used, our expansion method, how we choose advantageous fights, and how we control our military units in combat as these are the methods that we worked the most on and are most unique to us.

3.1 Build order

In Starcraft the early parts of each game are composed of building workers, units, and buildings. The order in which units and buildings are produced is called a build order. There are many different build orders to choose from and after some experimentation, we decided to use the *3 Gate Speedzeal* build order [10]. The build order is shown in figure 1. The goal of this build order is to quickly get an upgrade for the unit Zealot which increases its movement speed to be faster than the very popular unit Dragoon.

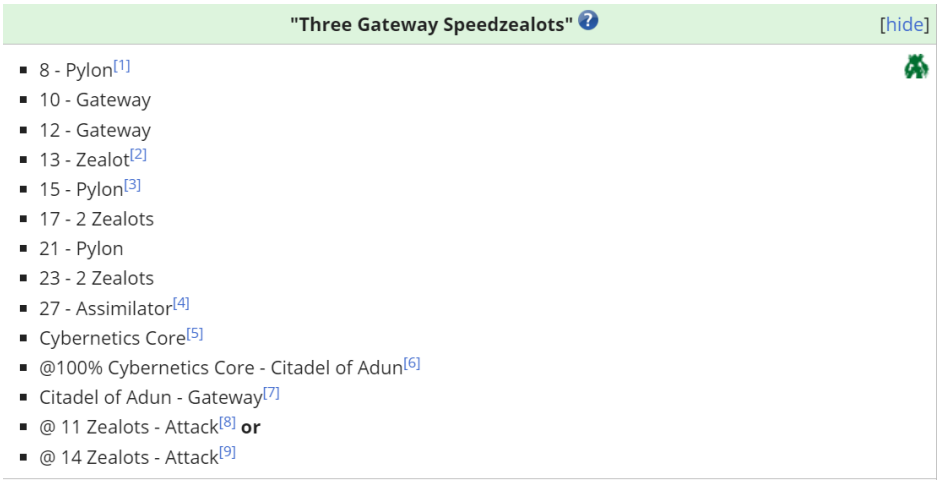


Figure 1: The build order that we used. The number on the left side (for example 8 on row one) represents the supply (total number of units) that we want before we build the unit specified on the right side (Pylon on row one). Note that we are continuously producing workers while following the build order.

3.2 Building new bases to improve economy

We implemented a method that allowed us to build new bases in order to gather more resources and another method that made sure that we had the correct number of workers at each of our bases. A crucial part of RTS games is the economy. In order to win you need an army capable of destroying the enemy's army, in order to get an army you need resources gathered by your workers, and in order to maximize how much resources your workers can gather you need to continuously expand.

There are two crucial reasons for why we need more bases:

1. Each base has one gas resource and 9 mineral resources. The optimal gathering is achieved with 3 workers per resource (meaning that $3+9*3=30$ is the maximum number of workers for an optimal base).

2. There is a limited amount of total resources to gather from each mineral or gas before they are depleted and can no longer be gathered from.

To maximize your economy you want to constantly build workers, and then add bases as the needs arise. We can find all possible base expansion positions with the BWEM library [6].

The way we do this is explained here, our current number of workers is denoted as *num_workers* and our current number of bases with non-depleted

resources is denoted *num_bases*, each time we build a worker we do the following:

1. If $num\_workers/num\_bases > 25$, continue
2. Find the closest base position from our starting position that has not yet been expanded to.
3. Construct a base (Nexus) and gas collector (Assimilator) at the new base position. This can be seen in figure 2.

This method is then combined with another method that makes sure that we have the correct number of workers at each base, every time we build a worker we do this:

1. For each of our bases:
2. Count the number of non-depleted mineral patches at this base, save this as *num_minerals*.
3. Count the number of non-depleted gas resources at this base, save this as *num_gases*.
4. The total number of workers that we want at this base is $max\_workers = 3 * num\_minerals + 3 * num\_gases$.
5. Count our workers that are close to the base, save this as *num_workers*
6. If $num\_workers > max\_workers$, send remaining workers to one of our other bases.

Figure 2: A new base being built. On the bottom left is the mini-map, here we see our first base as the green area on the upper middle part of the right edge, our new base is the smaller green area right below our first base.

3.3 Choosing advantageous fights

We implemented a method where our military units only fought battles where they were in a numerical advantage compared to the enemy's military units, an example of this is shown in figure 3. This to ensure that every fight we take will result in a net gain for us. This method was mainly used by the unit Zealot, which is the first military unit that we are able to build. The method counts all enemy military units, including the structure Photon Cannon, which is a non-moving building that is able to attack enemies within a specific range.

It works in the following way:

1. On each frame, do:
2. For each military unit:
3. Attack-move towards the enemy base, attack-move means that the unit walks towards a target position and attacks any enemies it finds along the way, the unit will also prioritize enemy units that can attack (enemy military units or cannons).
4. Find and count all enemy military units, including photon cannons, within a range of 250 pixels, save this in integer *num_enemies*.

5. Find and count all allied military units within a range of 250 pixels, save this in integer *num_allies*.

6. If *num_enemies* >= *num_allies*, retreat towards our base.

(a) Zealots retreating from a fight because of disadvantage, 3 allied units versus 3 enemy units.

(b) Zealots attacking because of numerical advantage, 4 allied units versus 3 enemy units.

Figure 3: Example of our method that ensures only advantageous fights are taken.

3.4 Dragoon kiting

We also implemented a method that allowed our ranged unit Dragoon to kite, which means that it will use its higher movement speed compared to many other units to attack with its ranged attack and then run away before it is damaged by the enemy, this will be explained in more detail below.

The dragoon is a unit that has a ranged attack and it has a high movement speed. It is mainly used in fights against other Dragoons but also against Zealots which is a melee unit that moves slower than the Dragoons. The impact of this method was noticed when our Dragoons fought against the enemy Zealots. Figure 4 illustrates how this method works.

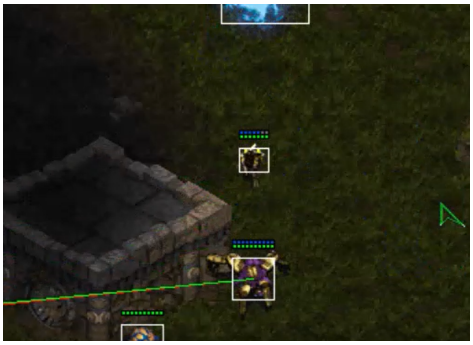
Our kiting method works like this:

1. For each of our Dragoons:

2. Attack-move towards the enemy base.

3. Search for enemy units within a range of 100 pixels, which is approximately how far the Dragoon can attack. If there is an enemy military unit within this range, run towards our base.

(a) Dragoon attacking an enemy zealot.



(b) Dragoon running away from the zealot that is now to close.

Figure 4: Dragoon kiting.

4 Experimental results

The tournament was divided into two parts, pre-finals and finals. The results are shown in figure 5 and 6.

Prefinals												Results	
		G2	**G3**	**G5**	**G6**	**G10**			Wins	Losses		Result	Wins
	G2	x	G2	G5	G6	G2		**G2**	4	4		**G5**	7
	G3	G2	x	G5	NAP	G3		**G3**	3	5		**G2**	4
	G5	G5	G5	x	G5	G5		**G5**	7	1		**G3**	3
	G6	*NAP	G3	G5	x	G10		**G6**	1	7		**G10**	3
	G10	G2	G3	G10	G10	x		**G10**	3	5		**G6**	1

Figure 5: Results from the pre-final tournament, we are group 5.

Finals												Results	
		G2	**G3**	**G5**	**G6**	**G10**			Wins	Losses		Result	Wins
	G2	x	G2	G5	G2	G2		**G2**	6	2		**G5**	8
	G3	G2	x	G5	G3	G3		**G3**	4	4		**G2**	6
	G5	G5	G5	x	G5	G5		**G5**	8	0		**G3**	4
	G6	G2	G3	G5	x	G10		**G6**	0	8		**G10**	2
	G10	G2	G3	G5	G10	x		**G10**	2	6		**G6**	0

Figure 6: Results from the final tournament, we are group 5.

4.1 Analysis

In the pre-finals, we won all games except for one, and in the finals, we managed to win all of our games. We will divide the analysis into two parts, one for pre-finals and one for finals.

4.1.1 Pre-finals

We saw great results in the pre-finals, seen in figure 5, winning 7 games and losing only 1. The biggest reasons for our success was our superior economy compared to the enemy as well as our very well-functioning dragoon micro, explained in section 3.4, we also noticed that many of the other groups still had problems with bugs that we had noticed and solved earlier.

During the pre-finals, we also had an opportunity to see what strategies the other groups used. The main thing we noticed was that all groups seemed to prioritize the Dragoon unit and because of this, we decided to change our strategy to something that countered dragoons. We looked around for good build-orders and which unit to get versus Dragoons and eventually decided to go for a lot of Zealots with the available speed upgrade. This speed upgrade makes the Zealot move faster than Dragoons. See section 3.1 for more information on the build order we decided to use.

Our superior economy was clear to see once we attacked, we had a lot more units than all of our opponents, for example in the game against group 2 we see the armies in figure 7. The reason for us having so many more units is that we constantly built workers the entire game, expanded when we needed, and built as many military units as possible. Another explanation is that most teams decided to go for late-game strategies, focusing on buildings instead of units early on. This is something we wanted to exploit in the finals by attacking very early.

Figure 7: Battle between our army (blue) and opponent army(yellow). It is clear that we have many more units than the opponents.

The game that we lost was because of a small bug in our bot where we would not build buildings properly, we managed to solve this before the finals.

4.1.2 Finals

In the finals we had amazing results, seen in figure 6, winning every single game.

The main reasons for our success were our new build order, explained in section 3.1, and our method that ensures that we only fight battles that we can win, explained in section 3.3.

Every game played out in a similar fashion, once we had produced our first couple of Zealots we attacked while still producing and sending more Zealots towards the enemy base. Similar to the pre-finals, most groups still stuck with a late-game strategy, only producing a few Zealots in the beginning of the game and those were quickly and safely defeated thanks to our methods. After only a couple of minutes, the difference in Zealot numbers between us and the opponents was too large for them to handle and our victory was secured.

5 Summary and Conclusions

This report defines the problem of creating an AI for Starcraft, discusses the choice of methods, and compares the results with other solutions. We managed to implement methods that worked extremely well, winning all

but one game in the pre-finals and winning every game in the finals. Our macro plan included a good build order to ensure that we built a strong base in the shortest time possible, and expanding to a new location with enough workers, which gave us a great economy coupled with a large army. With the advantage from our macro and smart micro strategies, where we only took advantageous fights and kited with ranged units, our AI managed to outmaneuver our opponents with great success. We also observed that attacking our enemies early yielded great results, as most of our enemies opted for a late-game strategy and did not have enough units to withstand us during the early-game.

With a majority of our focus being on the early game, leading to quick games, our AI's late-game strategies were not tested and may be lacking. Future improvements include implementing micro strategies for more advanced units which will make our bot translate better to drawn-out games. Also, to make our AI more flexible, it should be able to play other races, on other maps.

References

[1] Webster Andrew. Dota 2's the international returns in august with $40 million prize pool. *The Verge*, 2021.

[2] Michael Buro, James Bergsma, David Deutscher, Timothy Furtak, Frantisek Sailer, David Tom, and Nick Wiebe. Ai system designs for the first rts-game ai competition. In *Proceedings of the GameOn Conference*, pages 13–17, 2006.

[3] Dave Churchill. Ualbertabot. `https://github.com/davechurchill/ualbertabot`, 2021.

[4] David Churchill and Michael Buro. Build order optimization in starcraft. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 6, 2011.

[5] David Churchill and Michael Buro. Incorporating search algorithms into rts game agents. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 8, 2012.

[6] Igor Dimitrijevic. The bwem library. `http://bwem.sourceforge.net/index.html`, 2017.

[7] Timothy Furtak and Michael Buro. On the complexity of two-player attrition games played on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 5, 2010.

[8] Niels Justesen and Sebastian Risi. Learning macromanagement in starcraft from replays using deep learning. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 162–169. IEEE, 2017.

[9] Brood War API Team. Brood war application programming interface. `https://github.com/bwapi/bwapi`, 2019.

[10] Liquidpedia Team. 3 gate speedzeal (vs. protoss). `https://liquipedia.net/starcraft/3_Gate_Speedzeal_(vs._Protoss)`, 2019.