Efficent Armies in StarCraft II

Connor Wise and Murtaza Jafry
Department of Mathematics, University of Washington

August 16, 2021

Abstract

In this paper we will construct a linear program for optimal army configurations within the the real time strategy game StarCraft II. We will provide a detailed analysis regarding the simplifications and assumptions made in our model, and then assess the respective output of the model. From this output, we will run simulated battles against different permuted army configurations. The desired result will provide a quantitative metric from which the success of the linear program can be assessed. Ultimately, we will conclude with future directions of the linear program and potential modifications that can be made to provide a more complicated model.

1 Introduction

In the real-time strategy game StarCraft II, one must gather and use resources to build up their base and army in an attempt to destroy the other player's base and army. The resources used to create one's base and army are limited, so one needs to be careful about how much they are spending. The two main resources, minerals and gas, are gathered over time at one's base and are used to generate new structures, research upgrades, and produce new units. Though not explicitly a resource spent in the game, time is also valuable as each unit takes time to produce. When building an army, a typical player wants to minimize both the amount of resources and the time spent producing their units.

Army composition is a huge aspect of the game. Each army is composed of a variety of different units, each with their own health, damage, and resource cost. There is a population cap of 200 that players cannot exceed, and different units can contribute different amounts to this cap. In game, this is called supply, and as an example one Marine contributes 1 supply whereas a Siege Tank contributes 3 supply. Each unit behaves differently, for example some units can only attack flying units, some units deal bonus damage against organic units, some units are ranged. Additionally, there are three races that one can play as, Terran, Protoss, and Zerg, each with different units and play styles. How a player builds their army is one of the biggest determining factors for whether or not they might win a battle. For example, a line of Siege Tanks might do well against a group of Marines, but will get absolutely destroyed by just a single Liberator air unit.

One key aspect of the game is scouting out an enemy's army and building one's own army to counter that. The faster one can decide on a counter strategy, the better that player has a chance of winning. Our goal is to create a few different models that can determine optimal army composition with minimal resource cost and build time given some enemy army composition.

1.1 Research Goals

The main goal of this project is to create a tool where a player can input an enemy army composition that they are struggling with and the output will be an efficient counter army composition[1]. This leads us to our two main research questions:

1. Given an enemy army composition, what is a viable counter army with minimal build time?
2. Given an enemy army composition, what is a viable counter army with the least resource cost?

[1]The full code for the project can be found at https://github.com/MurtazaJafryPrime/Math381FinalProject.

Our tool can measure efficiency two ways, build time and resource cost, and the user can choose whichever they value more. Fast build time is important if a player must react quickly to an impending enemy army approaching their base, whereas minimal resource cost might be more important if one is trying to spend more resources on expanding their base or researching upgrades instead of purely building armies. This tool is most useful outside of game matches. A player might use this tool to analyze past matches to see instances where they were using less than ideal armies and highlight areas for improvement. This tool can also be used to input a wide range of different similar enemy armies and analyze the outputs so that one can gain a better understanding of unit interactions and further develop their game sense that way.

In this paper, we will be demonstrating one use case of the the tool we developed by finding an optimal army configuration for a Terran player playing against a standard Terran Bio army.

1.2 brief objective/description

In our problem, we are seeking to provide a solution to optimal army configurations in StarCraft II. This will be achieved through constructing a linear program with set of resource constraints such that given a finite set of resources, what kind of optimal armies can be constructed such that the probability of winning is maximized.

2 Assumptions/Simplifications

To determine if the output army is indeed a viable counter to the inputted enemy army, we must have those two armies do battle. We cannot reasonably run actual games of StarCraft II for every army we wish to test, so instead we have created a combat simulator that replicates how we would expect those two armies to interact in the real game. This simulator is a very simplified version of battle, and runs with the following assumptions:

1. Two opposing armies
2. Each army begins just out of sight from each other
3. Each army is grouped together in a relatively tight, random cluster
4. Every unit in both army has perfect accuracy
5. Each army receives a single “attack-move” command at the same time

The assumption that every unit has perfect accuracy refers to the fact that a unit will always attack (and damage) an enemy unit if it can. This disproportionately gives melee units an advantage over ranged units since in a real game ranged units tend to clump up with melee units only able to attack the surface area of that clump, whereas the simulator assumes that melee units can always attack regardless of clumping. Due to the prevalence of ranged units in the Terran race and the swarming melee units of the Zerg race, certain Terran-Zerg match-ups are not as accurately modeled by our simulator. In a real game, there are usually a small number of units that stand on the edge of the battlefield just outside of attack range, and one must carefully micromanage all of their units to ensure that every unit is always attacking. An army that has perfect accuracy would give it a huge advantage over one that does not, but since we give both armies perfect accuracy that advantage should cancel out if the two armies are of similar balance between melee and ranged units.

The assumption that each army is grouped together in a tight random cluster is due to the fact that our simulator has no concept of space or distance. In a normal game, different units move at different speeds and so similar units tend to clump together and armies tend to spread out as they move across the map. However, at sufficiently large scales the space between units at the battlefront becomes small enough that each army becomes densely packed again. Due to this density, our simulator also assumes that damage that has an area of effect will deal its damage to multiple enemy units densely packed in that area. While this leads to area damage being overpowered for small engagements, it is fairly accurate at the large, chaotic scales that a player might struggle with.

Some key elements of the game that are ignored are terrain and map restrictions (so no height advantages or bottle-necking of armies), no micromanagement (this comes form assumptions 3 and 5), and no spells or primarily spell-casting units (so no walls, invisibility, or other special abilities). There are a few exceptions to this last rule, notably the healing function of some units and the ability for special units to produce other units (such as a Carrier producing Interceptors in the middle of battle). These limitations make our simulator

extremely inaccurate at high levels of play, where very skilled players often micro-manage their units, abuse terrain advantages, and strategically use key spells to turn the tide of battle. However, this simulator accurately reproduces the way in which new or low-skilled players play the game. Since this tool is aimed at less skilled or knowledgeable players who are looking to improve, these limitations are determined to be acceptable. The limitations of the simulator also allow it to simulate multiple large battles within a second, which is necessary since we run it thousands of times every time we run our model. Similar combat simulators were demonstrated to overall be fairly accurate models of basic battles. This is also seen in an adjacent construction of the StarCraft linear program found in [1].

Another limitation of our tool is that one must enter in exact unit counts in for enemy army. While scouting is a key element of the game, it is extremely unlikely that a player will know the entire composition of their enemy's army with enough time to build a counter army from scratch. We work around this issue by running our tool against a list of enemy armies whose compositions are extremely similar. These enemy compositions differ by having slightly different unit ratios, or are of similar ratios just scaled up. In this way we can analyze the range of outputs and get an idea of the general type of army compositions we should build to counter that general type of enemy army.

3 Mathematical Model

3.1 Linear Program

In our mathematical model, we will seek to construct optimal armies in combat in StarCraft II. Each army consists of a combination of various units, and so each army composition can be described by the name and count of each unit in that army. So each army can be described as

$$army = \{unit_1 = n_1, unit_2 = n_2, ...\} \tag{1}$$

where each n_i is the count of each $unit_i$ in the army. Our model iterates through various combinations of these units to construct armies, figures out which of these armies can viably counter the given enemy army, and then outputs the army composition that is the most cost-efficient.

We will begin describing our linear program by defining our variables. First is

$$picked_{army} \tag{2}$$

which is a binary variable that is 1 if this army is chosen, and 0 if it is not. Next is

$$viability_{army} \tag{3}$$

which is a binary variable. Viability is determined by running the given army against the enemy army in a combat simulator 100 times. Viability is 1 if the given army meets our combat requirements, and 0 if it does not. The combat requirements are adjustable in our tool, but by default an army is viable if it can defeat the opposing army within 200 seconds with at least 10% of its units surviving at least 90% of the time. The 200 second time limit is to avoid situations where two armies come to a standstill (such as only ground units that cannot attack the air on one side and air units that cannot attack the ground on the other) or any other situation where the enemy army can simply walk past the opposing army with minimal losses and attack one's base directly.

Next are our integer variables.

$$supply_{army} \tag{4}$$

is the total supply the given army uses. We can think of army supply being the total population of the army, with each unit's individual supply being a weighted population value. We can therefore define $supply_{army}$ as

$$supply_{army} = \sum_{unit \in army} supply_{unit} \cdot count_{unit} \tag{5}$$

The variable $cost_{army}$ is the sum of the total resource cost of each unit in that army. By default in our tool this is defined as the finite resources of minerals and gas, though another option is the more intangible resource of build time. This value is defined as

$$cost_{army} = \sum_{unit \in army} cost_{unit} \cdot count_{unit} \tag{6}$$

Now that we have defined our variables, we may explain our objective function, which is to minimize

$$\min\left\{\sum_{army} cost_{army} \cdot picked_{army}\right\}. \tag{7}$$

This function is subject to the following constraints. The first is that we only want to have one optimal army, giving us the constraint

$$\sum_{army} picked_{army} = 1. \tag{8}$$

We only wish to pick armies that can viably counter the enemy, and so we have that for every $army$,

$$picked_{army} \leq viability_{amry}. \tag{9}$$

We must also restrict our army size, giving us

$$picked_{army} \cdot supply_{army} \leq supply\ cap. \tag{10}$$

Supply cap is an integer that determines the maximum army size. This value has a default cap of 200, though it is not always the case that the player is willing or capable of spending all of their supply on a single army (for example, such player might already have other units built or might be fighting a war on two fronts and needs multiple armies).

The resources used to construct units are finite and take time and effort to collect in the game. Since a player would presumably know how much they can spend on an army we have included constraints that caps the total resource cost per army, each in the form as

$$cost_{army} \leq resource\ cap, \tag{11}$$

where cost can be either minerals, gas, or time. By default, the resource cap for each of these is ten billion, effectively letting each of these resources be uncapped without additional user input. These constraints are not necessary to run our program, but they do aid in speeding up calculations.

3.2 Simulation

The biggest key component to our program is the use of a simulation that approximates battles in StarCraft II. Our simulator takes in two army compositions, written as the names of units and count of those units in the army, and follows a set of simplified game rules and logic to approximate the actual outcome of battle if those two armies were to fight in a real game. The limitations of this simulation are outlined in section 2, but the key thing is that our simulation approximates how two low-level players might play. There are two components to our simulation, the approximation of the individual army units and the approximation of the battle. The methods used to create these approximations follow closely from those outlined in the paper *Approximation Models of Combat in StarCraft 2*, along with some improvements and added complexity [1].

3.2.1 Units

To approximate the way in which individual units behave, we modeled each unit to have the following base combat attributes:

1. **Health:** Integer representing the amount of health plus shields this unit has. If the unit has armor, then we updated health to be $1.5 \cdot armor \cdot (health + shields)$. The unit is considered dead when their health reaches zero.
2. **DPS:** Decimal representing the total damage per second the unit can deal.
3. **Ranged:** Boolean value if that unit has a ranged attack or not. Units with ranged attacks got one extra round of damage at the beginning of combat.
4. **Attributes:** String attributes of the unit, such as Biological, Mechanical, Light, Armored, etc.
5. **Type:** String determining if that unit is an Air or Ground unit.

6. **Targetable:** Strings determining if the unit can attack Air and/or Ground units.

7. **Bonuses:** Strings determining what Attributes this unit deals bonus damage against.

8. **Bonus DPS:** Decimal representing the additional damage per second this unit deals with bonus damage.

In addition to these base attributes, some special units have secondary attributes to model specific interactions unique to those units. For example, the Medivac and SCV units can both heal certain types of allies, but the healing per second of the Medivac is constant whereas the healing per second of the SCV depends on the specific allied unit being healed. Some units can produce other units, such as the Carrier that creates Interceptors over time or the Brood Lord that produces Broodlings with every attack.

Our combat model runs in one-second rounds, hence our use of damage per second instead of damage per attack. However, this leads to some inaccuracies in how we modeled the Armor attribute. In the game, armor provides flat damage reduction against every attack and so armor is more useful against low damage, fast attack speed enemies than those with high damage, low attack speed. For example, a unit that deals 3 damage with 10 attacks per second (for 30 dps) would only deal a total of 20 damage against a unit with 1 armor in a single round. On the other hand, a unit that deals 30 damage with 1 attack per second (for 30 dps) would deal 29 damage against a unit with 1 armor in a single round. To manage this difference, we have decided to model armor as a multiplicative health bonus instead of traditional damage reduction.

For units that can transform or those with multiple attacks, we chose to only include the most common transformation/attack in our model. For example, we assume that Siege Tanks are always in siege mode, and that Vikings are always in fighter mode. Most spells were not included, and so some pure spellcaster units or other units that lacked attacks were not included either (such as the Overseer or Observer).

3.2.2 Combat

Combat was modeled in one second rounds by having each unit randomly choose an applicable target and deal damage, removing dead units, and updating special interactions until one army has no more units or our 200 second time limit is exceeded (this time limit was to included to avoid stalemates and other undesirable situations). Only ranged units may attack during the first round of combat to simulate the attacks that ranged units would deal before melee units move close enough to counter attack. An overview of our combat model is as follows:

```
combat_sim(army1, army2):
    round1 = True
    rounds = 0
    while (get_health(army1) > 0) and (get_health(army2) > 0) and (rounds <= 200):
        if round1:
            deal_ranged_dps(army1, army2)
            deal_ranged_dps(army2, army1)
            else:
                healing(army1)
                healing(army2)
                deal_damage(army1, army2)
                deal_damage(army2, army1)
        army1 = remove_dead_units(army1)
        army2 = remove_dead_units(army2)
        build_Interceptors(army1)
        build_Interceptors(army2)
        track_Locust_Broodlings(army1)
        track_Locust_Broodlings(army2)
        rounds += 1
    if rounds >= 200:
        army2.clear()
    return(army2)
```

The functions to build Interceptors serves to check for the special interaction Carriers have in building Interceptors, and the way that such Interceptors die if their parent Carrier is destroyed. We have also included

a special function to model how Locusts and Broodlings (which are special units created by Swarm Hosts and Brood Lords respectively) can only live for a set number of rounds before dying.

The way in which units deal damage is as follows. The unit checks that it still has dps remaining and that the list of attackable enemies is not empty (if a unit can only target ground units but only enemy air units remain, then that unit cannot attack). A random attackable enemy is targeted and damage is dealt until either the target is dead or the unit has dealt their maximum damage. If the target dies and the unit still has damage remaining, it selects a new target and repeats these steps. Bonus damage, if applicable, is dealt during this time following the same set of logic. The target-switching is to simulate high attack speed units (such as the Battlecruiser) that can kill multiple targets within one second. However, this also gives an extra advantage to high damage, low attack speed units such as the Thor that can realistically only attack one target per round. In the future, it might be possible to increase the complexity of our model and include considerations for attack speed, as this would benefit both the target-switching and armor damage reduction inconsistencies.

3.3 Solution of the Mathematical Model

Our combat simulator is a close approximation of a real in-game battle, but due to the randomness of damage and other limitations our model for determining combat viability must include more than just a single run of the simulator. For this reason army viability is determined by running the same combat simulator 100 times, with that army being viable only if it meets or exceeds the win percentage threshold of 90%. We chose 90% (as opposed to 100%) as the default threshold to account for any remaining limitations or imbalances in our simulator that would have otherwise given our test army an advantage. In this way, all army compositions that meet our viability requirements and are outputted by our linear program are known to have a high chance of winning.

Given our linear program and combat simulator, ideally we would be able write a python function that takes an enemy army composition as an input and runs through all possible army compositions in the linear program, outputting the most optimal army as defined by our constraints. However, the number of possible combinations of units (and therefore number of possible army compositions) is several orders of magnitude large and completely infeasible to calculate. To work around this issue, we used the Monte Carlo method and randomly generated test armies to run in our program. We wrote a function that found the optimal solution to our linear program given 100 randomly generated armies, and then repeated this process until we had 110 potentially optimal outputs. From there we ran our linear program again, this time with those 110 potentially optimal armies instead of random armies. By repeating our linear program with those better armies, we are able to simulate running a single linear program with 1100 random armies (but with much a faster runtime) and continuously refine our output towards the true optimal solution. As shown in our analysis, the output of our example run was not the true optimal solution, but it is still a strong candidate and did demonstrate how additional runs can refine the solution.

3.3.1 Randomly Generated Armies

Our model relies on the use of randomly generated armies to use in our linear program. Our algorithm is outlined as follows:

```
comp = #dictionary where keys:unit names, values:unit counts
# in this case, all unit counts are initialized to 0
names = #list of unit names
random.shuffle(names)
current_supply = supply_cap
    for name in names:
    # find the maximum count of this unit possible in this army
    max_unit = int(current_supply / Units[name]['supply']
    # Motherships are unique in that only one may be alive at a time
    if (get_army_supply(comp) <= current_supply) and (not extra_Motherships(comp):
        comp[name] = random.randit(0, max_unit)
   current_supply -= get_army_supply(comp)
    # check that random comp is not already generated and only include valid comps
    if comp not in comps:
```

```
        if (get_army_supply(comp) <= current_supply) and (not extra_Motherships(comp):
            comps.append(comp)
    return comps
```

By randomly choosing the order in which we start generating unit counts, we can ensure that no single unit is consistently ignored by the time the supply cap is filled up. By constantly tracking the current supply of our army, we can greatly reduce the number of invalid compositions and speed up the process of randomly generated large amounts of armies.

3.4 Literature Review

Our linear program will seek to provide optimal army configurations in the game of StarCraft II. The use of such a linear program is quite useful when trying to construct armies in a regimented fashion to go against opposing players. Real time strategy games have many different layers of complexity. This arises from the fact that there exist many unique configurations that can be adopted by either player when constructing an army with a finite amount of resources. Thus, trying to find optimal models that would provide a definite winning strategy against an opposing player in all circumstances is quite difficult. In addition to this, due to the variability of different kinds of real time strategy games, the construction of a linear program many times is game specific. In other words, one in principle could not adopt a similar linear program for two different real time strategy games. Due to this, the literature in creating linear programs on this topic is quite limited when specifically concerned with StarCraft II. However, there are linear programs whose general framework for approach can be quite applicable. Such kinds of programs include [2, 3, 4, 5, 6, 7].

Though these programs exist for similar games the subtleties of each game do not easily transfer between games. This is seen most explicitly from the fact that armies in the different games have different inherent abilities. These abilities would then have to be included within the constraints of the linear program as this leads to advantages and disadvantages for the game. Thus, the different models should provide motivation in how to construct alternative linear programs rather than as ideal templates.

4 Results and Analysis

To demonstrate the use of our linear program, we chose one specific army composition to optimize against playing. In this demonstration, both armies are of the Terran race. This army is a standard Terran Bio build, which revolves around biological units supported by Medivacs. The biological Terran units include the Marine, Marauder, Reaper, and Ghost. Reapers and Ghosts are mainly used for harassment and not large scale battles. A typical Terran Bio build has Marines and Marauders with a ratio around 3:1, and Medivacs with a ratio around 1 Medivac for every 10 or so infantry units. Two common ratios of Marine:Marauder:Medivac are 6:2:1 and 15:5:2. For our purposes, we will be testing the 15:5:2 ratio and optimizing against 45 Marines, 15 Marauders, and 6 Medivacs. This army has a total supply of 87, simulating an early- to mid-game composition. By default, our random army generator creates armies with total supply ranging anywhere between 0 and 200. To reduce the amount of iterations needed to hone in on the optimal solution, we will be limiting our random armies to a supply cap of 120, since we know for certain that pretty much any army with a supply greater than that will be suboptimal (such armies will likely be overkill, or contain too many support units that do not add enough value to the overall army composition).

The Terran Bio build is standard because it revolves around Marines which are cheap, one of the first units Terran players can produce, and are extremely versatile as they are fast, ranged, and can attack both ground and air units. Marauders act as meat-shields in addition to extra damage against armored units, and Medivacs provide constant healing to both Marines and Marauders. While the functionality is not included in our combat simulator, Medivacs can also carry a small group of Marines and quickly fly them around the map, making this build useful for harassment in addition to all-out battle.

A few key things stood out after analyzing the 110 outputs from the initial run of our program. First, out of the 110 output armies, 51 only contained one unique unit, 51 only contained two unique units, and 8 contained three unique unit. By unique units, we mean that if an army contained 10 Marines and 5 Siege Tanks then it would have two unique units (Marines and Siege Tanks), whereas an army only composed of Marines would have one unique unit. Examine the graph showing the distribution of unique units in our randomly generated armies as shown in the appendix A. We can see that our majority of the random armies we are testing likewise are

only composed of one or two different units. Though this might lead us away from a truly optimal solution, in a way this lack of diversity is similar to how some players go "all in" and only build a small subset of the possible units. Upon further inspection, we can see that sometimes extra units with no synergy were grouped together, such as having Medivacs in an army without any biological units to heal. This results from the randomness of the test armies generated, though ideally the number of such "wasteful" armies should go down as more tests are run. The general lack of unit diversity also reflects the type of enemy army inputed. Our enemy army is mainly composed of ground biological units with very little air units, and so it makes sense that our program would prefer armies that deal high ground damage and minimal anti-air damage.

Another thing that stood out was the spread in how unreasonable some of the outputs were. For example, output 65 was 18 Hellbats and 11 Battlecruisers. The Hellbats are very effective anti-infantry units, and so their inclusion makes a lot of sense. However, Battlecruisers are massive late game units, and are one of the most expensive units for a Terran player to build. Including 11 of them is a lot of overkill, making it surprising that our program outputed this composition as the most optimal in its run. On the other hand, outputs 26, 45, 66, and 73 only included the Banshee unit. Banshees are flying units with strong anti-ground capabilities, making them very useful against Marauders who cannot attack air units. Our outputs had 35, 40, 34, and 32 Banshees respectively, demonstrating how the possibility for a more efficient army of Banshees increased as the number of runs increased.

We re-ran our linear program using those 110 outputed armies instead of using randomly generated armies. The 75th output turned out to be the most optimal against the Terran Bio army. This army was composed of 32 Cyclones and 6 Banshees, for a total supply of 114 and a cost of 3800 minerals and 3800 gas. Comparing this to the Terran Bio army with a supply of 87 and a cost of 4250 minerals and 950 gas we can see that our "optimal" output is still far more expensive than the army it was trying to beat. A quick test also demonstrates that a cheaper army made up of just 29 Cyclones (but not 28) is still viable to counter the Terran Bio army. Our final output was still close to an optimal Cyclone-only army, but we are confident that given enough runs our program will converge to such optimal solution.

5 Improvements/Future Work

In this paper, we considered a linear program to minimize the total number of resources needed to win in a game of StarCraft II. Our linear program consisted of only a few simple constraints which were the limitation of resources and the viability of an army. Though this model provided a means of constructing simple opposing configuration in a game of StarCraft II, one could add another layer of complexity to the linear program. Specifically, one could consider the case where instead of simply have a finite set of resources, one could also constrain the linear program based on the damage impact of each unit. Our linear program did not consider the differential damage impact of each army unit on the opposing team. One could in principle include a means of including such a constraint to provide an even more optimal model. As we had seen in the output, certain army units were ideal when used in retaliation against an opposing army. For example, it was seen that a hellion unit was effective in combating a marine unit. Though we had seen that in the output, one could further apply a constraint onto the linear program where for each specified unit of the opposing army, one could include a finite set of opposing army units. More specifically, for every unit x one would include y units of a different ideal retaliatory type unit. In the case of the marine, one would introduce the hellion. This added level of complexity would provide an even stricter linear program which would further increase the the efficacy of creating an army. However, to achieve this, one would have to run many different simulations to find the exact mapping between which opposing units are ideal for retaliation. Due to this, the obvious downfall with this approach is the increased time complexity of the problem. Nevertheless, such an addition will be enlightening and provide further clarity in assessing the impact of different kinds of units in forming an army.

Some functionality suggested by community members was not fully integrated into our model. We included functionality to optimize for the total build time of units, but this stat does not reflect the speed in which a player might actually build such army. In the game, different buildings produce different types of units. For example, Marines are built in the Barracks and Battlecruisers are built from the Starports, and one may have multiple Barracks to build more Marines at the same time. Future improvements would include adding parameters to specify what production buildings already exist so that our model can account for true build time, not just absolute.

In addition, the model above can further be optimized so that the complexity can be reduced. At the moment, our current scripts converge to an idealized army for many simulations. However, due to the size of

the data sets the computation time for the full program would take a considerable amount of time. In our analysis of around 10 runs, the computation time took around 20 minutes. However, for the analysis of 100 runs, the total computation time would take around 3 hours. Though the results with 10 runs were sufficient to display the effectiveness of our model, the results of a larger set of runs would provide stronger, more optimal results. Due to this, future work on this topic could include seeking to further optimize the linear program such that the total computation time would be reduced significantly. The initial complexity of our model was reduced considerably with the implementation of a Monte Carlo simulation, however an even larger reduction of the data could provide an even larger decrease in time complexity. Our algorithm for randomly generating armies could also be improved to have a more even distribution of unit diversity.

Some of the limitations of our combat simulator are extremely impactful and our model is only really accurate when approximating very basic, low-level gameplay. The combat simulator was kept simple so as to increase the speed of computation, but in the future it might be possible to include more game mechanics, such as spells, grouping, and stealth. Increasing the accuracy of our combat approximation will increase the accuracy of our linear program, though the main worry is that the added complexity will increase the already lengthy run times.

6 Conclusions

In this paper, we sought to create a linear program for outputting ideal army configurations in the game of StarCraft II. We analyzed the case where given a set of resource constraints, what are the most ideal set of army units that one could construct to oppose the army attack. In our case, we analyzed the situation where one could use any set of units available within StarCraft II, then we sought to minimize the objective function. Here the objective function for the linear program was to minimize the total resources needed when constructing an opposing army. The opposing army with the least resources would then in principle be the most efficient for simple combat matches.

In the end, we have found that our approach is too computationally intensive to be used for anything more than a learning tool or a discussion point when examining actual gameplay in StarCraft II. The results of our model were close to an ideal solution, but still needed a bit more work to be entirely optimal.

7 Acknowledgments

This work was conducted with the help of our instructor Junaid Hasan. He provided critical feedback in our work, and also assisted in providing a route to expedite the computation of the work through a monte carlo implementation. We would also like to thank our parents and our recommenders.

A Plot Analysis

In this section we examine a set of 1000 randomly generated Terran armies, similar to the ones used in our example. We can see that our random generator has a fairly even distribution of mid- and high-supply armies, but not as many armies with low supply. This was expected, which is why we included an option to include a lower supply cap in the random army generator. Our model is still fairly accurate at lower supply numbers, but it will take longer for our model to converge to an optimal solution.

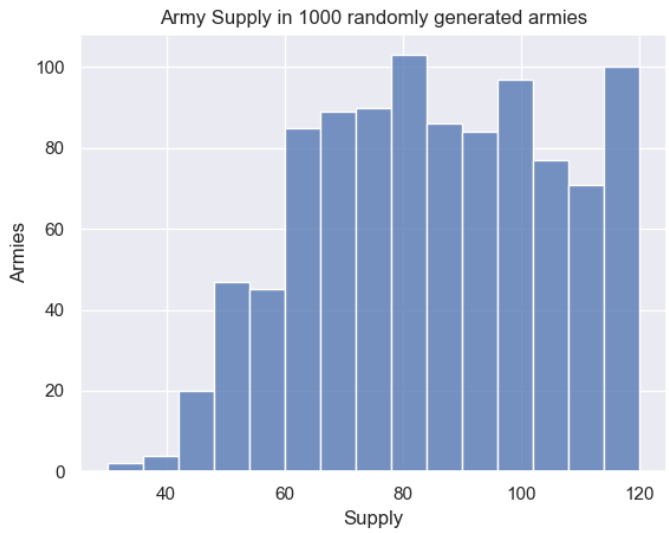

Figure 1: Armies are evenly distributed over mid- to high-supply

The next graph shows how many armies were composed of how many unique units. We included 15 different units that can be included in the Terran armies, but our random army generator skews very heavily to only included a small number of those different units. This is worrisome as we are not testing diverse army compositions at all, and so our model cannot converge to such armies even if the true optimal solution is diverse.

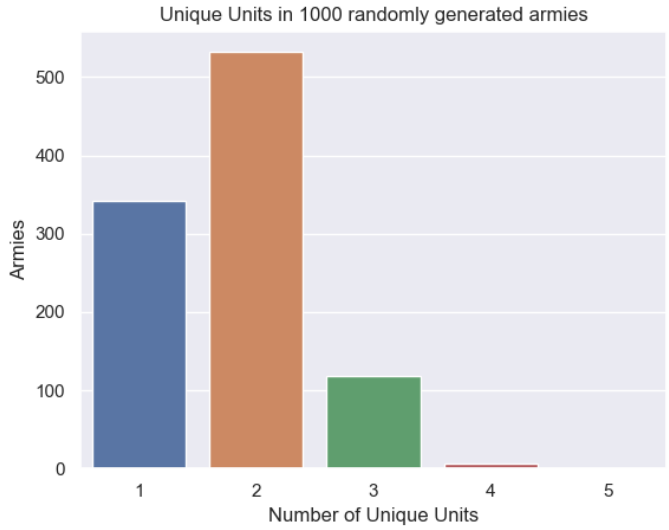

Figure 2: Most of the random armies only include one or two different units

References

[1] I. Helmke, D. Kreymer and K. Wiegand, *Approximation models of combat in starcraft 2*, 2014.

[2] B.H. Zhang and T. Sandholm, *Sparsified linear programming for zero-sum equilibrium finding*, 2020.

[3] P.-A. Andersen, M. Goodwin and O.-C. Granmo, *Deep rts: A game environment for deep reinforcement learning in real-time strategy games*, 2018.

[4] N.A. Barriga, M. Stanescu and M. Buro, *Combining strategic learning and tactical search in real-time strategy games*, 2017.

[5] A. Kadan and H. Fu, *Exponential convergence of gradient methods in concave network zero-sum games*, 2020.

[6] K. Zhang, Z. Yang and T. Başar, *Multi-agent reinforcement learning: A selective overview of theories and algorithms*, 2021.

[7] R.-J. Qin, J.-C. Pang and Y. Yu, *Improving fictitious play reinforcement learning with expanding models*, 2019.