Efficient armies in StarCraft II

Connor Wise[1] and Murtaza Jafry[1]

[1]Department of Mathematics
University of Washington

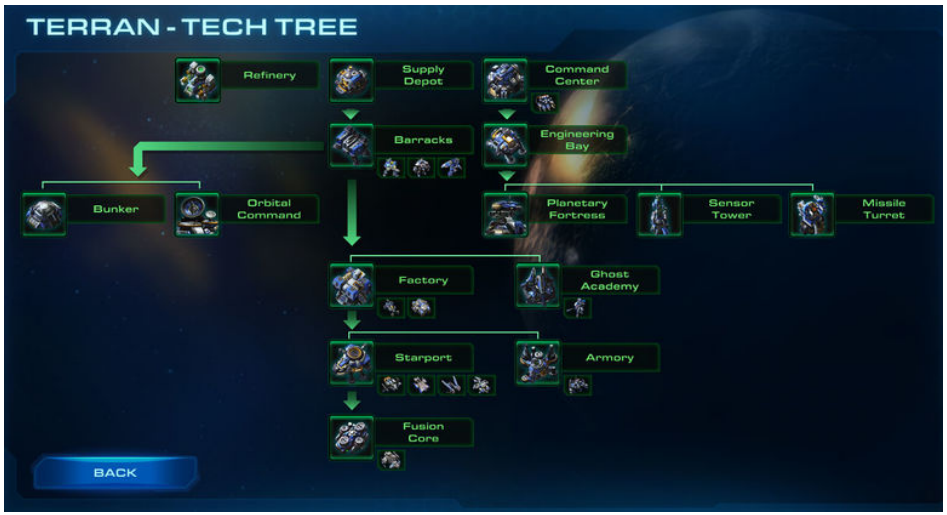
August 2021

Star Craft II

- Star Craft II is a real time strategy game in which players construct armies based on combination of units to battle. The game is meant to simulate the strategy that is developed in real battle.
- The game further allows users to make choices between certain army units based on resource constraints. Thus, the objective of the game becomes how to create optimal armies with a certain amount of finite resources.

Research Goals

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



- In this presentation, 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.

Assumptions/Simplifications

Given the complexity of the Star Craft II game, we will have to make some inherent assumptions about our linear program.

- Given an enemy army composition, what is a viable counter army with
- Two opposing armies
- Each army begins just out of sight from each other
- Each army is grouped together in a relatively tight, random cluster
- Every unit in both army has perfect accuracy
- Each army receives a single “attack-move” command at the same time

Linear Program

Given our assumptions and simplifications, we can now formulate our linear program. In our linear program we will have two main variables defined as

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

and

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

Linear Program Defined

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

$$\text{such that } \sum_{army} picked_{army} = 1.$$

$$picked_{army} \cdot supply_{army} \leq supply\ cap$$

$$picked_{army} \leq viability_{army}.$$

Simulation

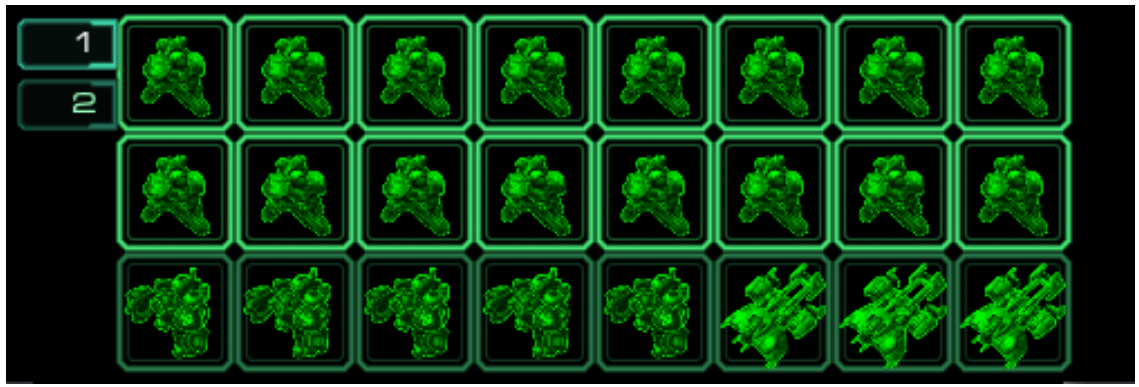
- **Health:** Integer representing the amount of health plus shields this unit has. If the unit has armor, then we updated health to be $1.5 \cdot armor \cdot (health + shields)$. The unit is considered dead when their health reaches zero.
- **DPS:** Decimal representing the total damage per second the unit can deal.
- **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.
- **Attributes:** String attributes of the unit, such as Biological, Mechanical, Light, Armored, etc.
- **Type:** String determining if that unit is an Air or Ground unit.
- **Targetable:** Strings determining if the unit can attack Air and/or Ground units.
- **Bonuses:** Strings determining what Attributes this unit deals bonus damage against.
- **Bonus DPS:** Decimal representing the additional damage per second this unit deals with bonus damage.

Combat

```
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)
```

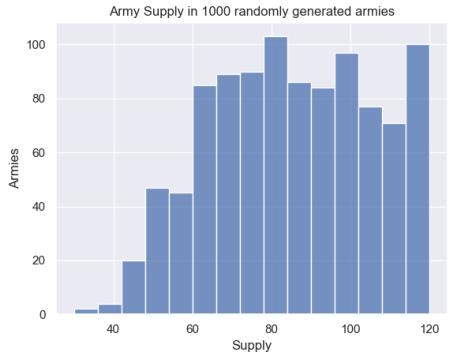
Solution of the Mathematical Model and Monte Carlo

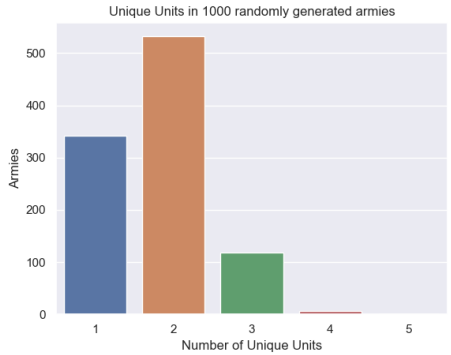
Our combat simulator is a close approximation of a real in-game battle, but due to the randomness of damage and other limitations our model for determining combat viability must include more than just a single run of the simulator. For this reason army viability is determined by running the same combat 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.



Results and Analysis

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.





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.

Demonstration

We ran our program 100 times to find an optimal counter to a Terran army composed of 45 Marines, 15 Marauders, and 6 Medivacs.

- Most outputs only contained 1 or 2 unique units
- Some outputs were very unreasonable
- As number of outputs increase, chances of optimal solution increase
- Rerun program using 100 outputs as new inputs. Final outcome was good, but still not optimal

Our program gave 32 Cyclones and 6 Banshees as most optimal, but an army of just 29 Cyclones is cheaper and still viable.

Improvements and Future Work

- 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.
- 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.

Acknowledgments

- We would like to Junaid Hasan for his help with the Monte Carlo simulation. We would also like to thank our parents and recommenders.

- The full code for the project can be found at https://github.com/MurtazaJafryPrime/Math381FinalProject
- Questions?