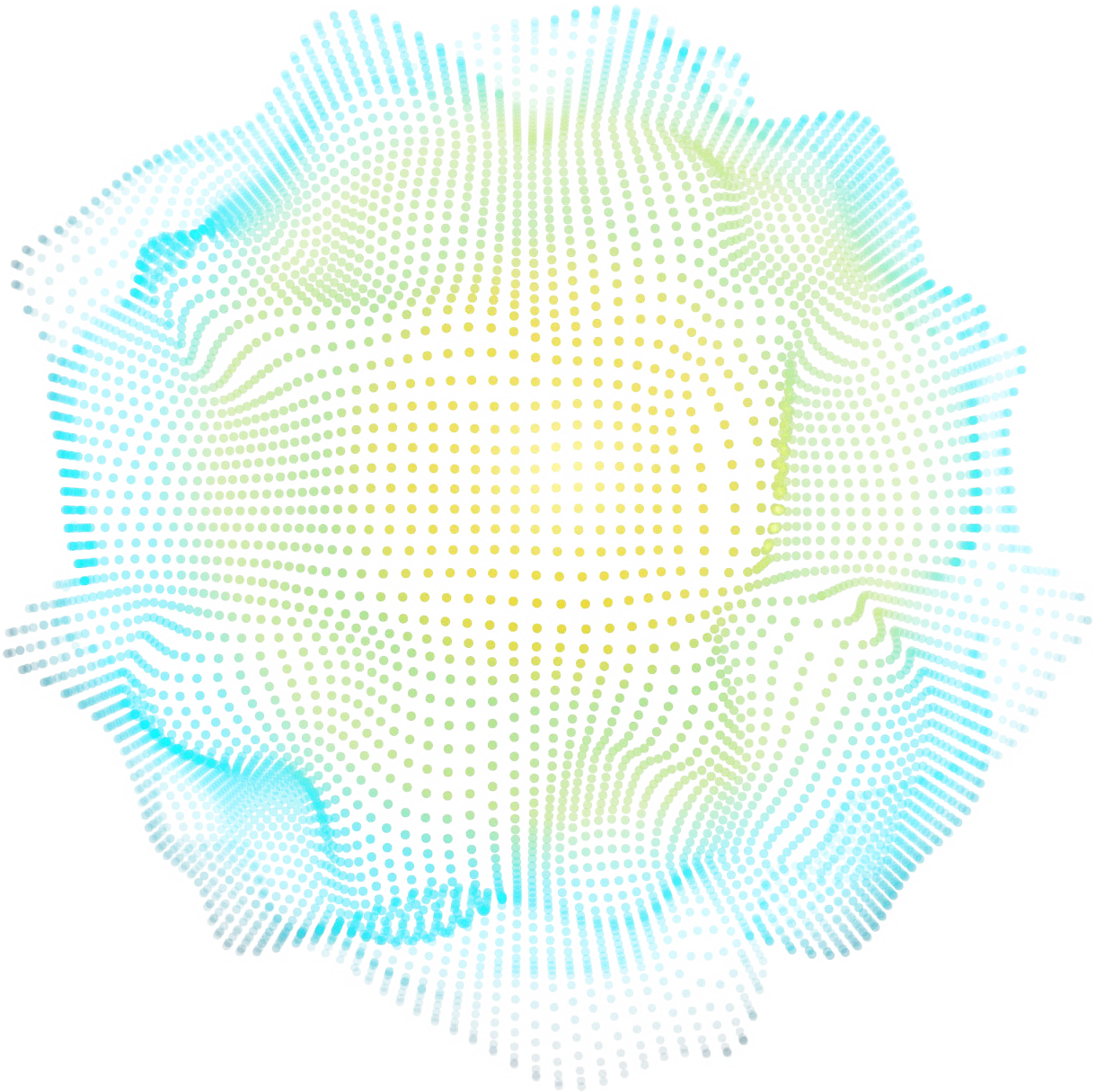
>_TERMINAL

correlation.one

PROBLEM SLOVING REPORT BY:

RISHUB JAIN, DEVENDRA CHAPLOT AND PAUL LIANG



WHAT IS TERMINAL?

Terminal is the world's largest AI programming competition, organized by Correlation One (www.correlation-one.com). In Terminal, players play a digital game programmatically, by coding their strategies into algorithms. These algorithms then face off against each other in an algo vs. algo practical coding challenge.

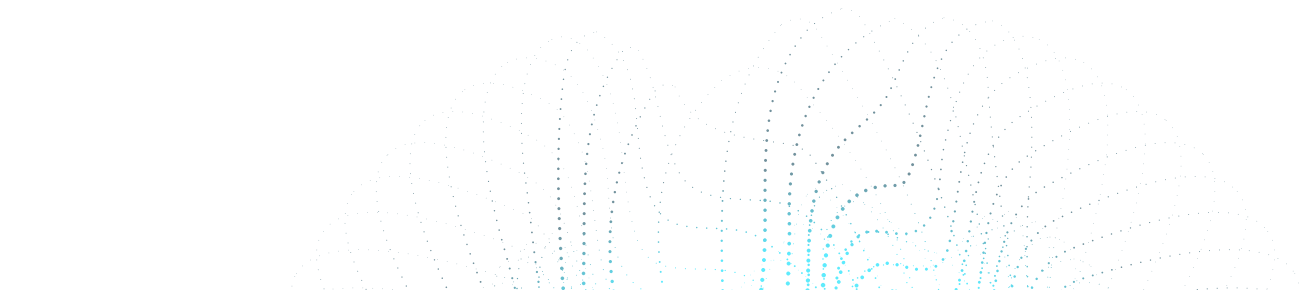
Building a strategy for Terminal requires a holistic set of engineering, problem-solving, and strategic thinking skills.

In addition to the main competition, Terminal and its sponsors host several live, in-person "game nights" in cities across the globe. During game nights, participants work in teams of 3 and have ~6 hours to build the best possible algorithm.

In this write-up, I walk through how I approached this problem during the 2019 Game Night and the skills I used. It gives you an insight into my problem-solving approach and thought process throughout the game night.

TECHNICAL FACTS ABOUT TERMINAL

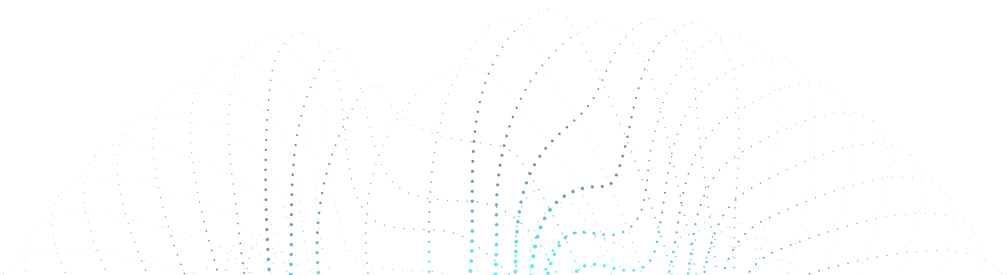
The competition page is here: **https://c1games.terminal.com**
There were 6 hours to code an algo
Bots were coded in teams of 2 or 3
Players could code algos in Python, Java, and Rust



>_TECHNICAL STACK

Q1.1. Which programming languages, environments, and tools did you use during the competition, and why? If any were new to you, how did you assimilate them into your existing knowledge and how did you leverage them in practice?

We use python to implement our algorithm, but no external tools and packages apart from those built into python such as dictionaries, heaps etc. We used learned and used the python debugger pdb for pesky bugs which was important since rerunning the script with print statements would take time. Pdb allows us to place check points and inspect the variables at each step of computation. We believe that pdb will also be an important debugging tool when we code in the future.
In addition to individual implementations of the code, we also learned how to collaborate effectively using Github.



>_STRATEGY DEVELOPMENT

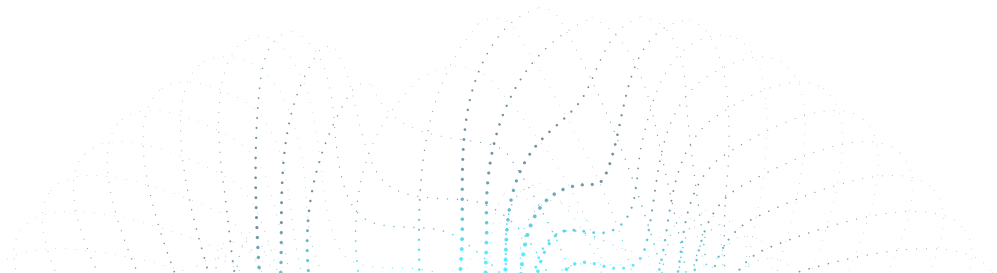
Q2.1. What was your detailed process for improving your strategies and code over time? What significant obstacles did you encounter in this process, and how specifically did you work through those obstacles?

We generally approached the problem as an optimization problem over actions and locations. We optimize over actions to determine which units to place given our fixed budget, and we optimize over the locations to place these attacking and defending units. The objective function is some measure of the expected long term reward that we gain with respect to the opponent's best response strategy. To obtain an estimate of these rewards, we perform simulation on the current state of the game to see how our actions impact the units on the board and the number of points for each player. Obviously attempting to solve this large optimization problem exactly quickly becomes intractable: there are an exponential number of {action, location} combinations to try. Therefore, the most obstacle we ran into was achieving a balance between the speed of our algorithm and the level of optimization we perform.

This calls for several heuristics that we need to perform. These assumptions are used to decrease the size of the search space thereby reducing the amount of time taken for these optimization problems.

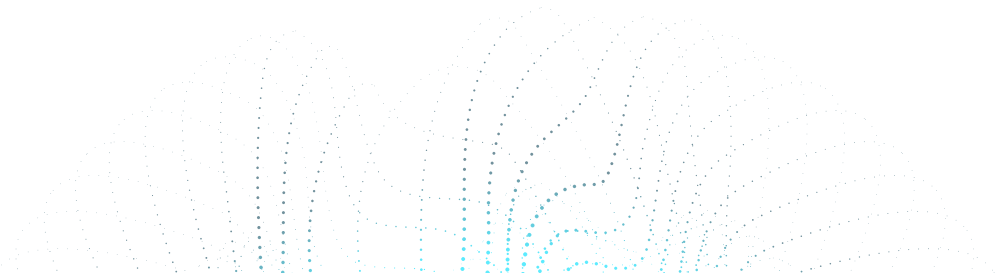
Below we list some examples of the heuristics that we chose:

1. Assuming that actions taken within each turn are independent on one another. This reduces the simulation complexity from a multiplicative combination to an additive combination across the actions taken in each turn, because we can just simulate the actions taken one after the other rather than combinatorically simulating all possible actions per turn.
2. We group locations together based on their distance. In other words, instead of searching over individual coordinates of which there are n x n, we search over blocks of coordinates each of size k x k first before recursing on the best block of size k x k. This hierarchical search strategy effectively reduces a search that scales linearly in n to one that scales logarithmically in n (similar to binary search). This was useful in practice to increase the running speed of our simulation algorithms and allows us to perform more detailed simulations in the same time.
3. We only perform simulate actions that are "reasonable", where we define



reasonable as possibly gaining good rewards. This is determined using a combination of the shortest paths, the reward of each path, as well as the opponent's best response strategy. For example, when deciding where to simulate our attacking units locations, we only optimize over the locations where the shortest paths lead to higher rewards.

Each of these simplifying assumptions and heuristics were discussed at length between the team members. We tested each assumption for their impacts on our algorithm's performance and speed. We computed the tradeoff that resulted from each heuristic and our final algorithm was generally optimized to achieve best performance while running quickly within the time limit.



Q2.2. Please walk us through your final algorithm's logic, as well as one significant non-obvious change you made to your algo along the way, as well as the rationale for the change.

Our final algorithm is composed of 3 sections: simulation, attack, and defense.

Our simulation strategy forms the basis of our algorithm. Given the current state of our board and an attacking unit at a starting location, the simulation strategy involves determining the shortest path that the unit will take towards the opponent's board and simulating the reward that this agent obtains (either back destroying opponent blockers or reaching the opponent's end of the board).

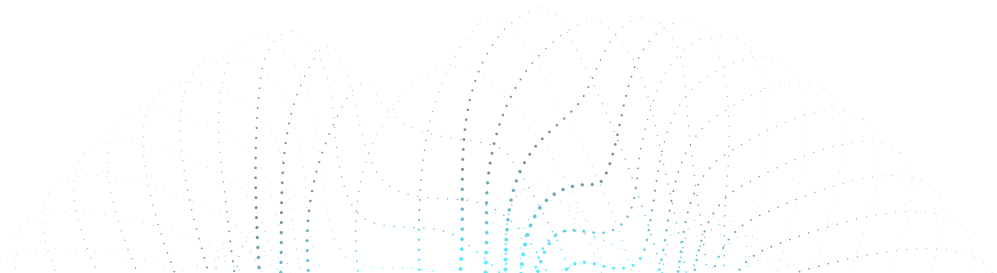
In order to determine shortest path through the board, the default algorithm uses breadth first search to determine the shortest paths for our agents to take towards the opponent's area. Below we list some important improvements we made to improve the efficiency of our search algorithm. Let n denote the number of edge vertices and m denote the number of edges.

One big change we made was to improve the efficiency of our algorithm by using more efficient search algorithms. We realized that the original algorithm provided used variants of breadth first search to determine the shortest path. Furthermore, BFS is run 3 times for a total runtime of O(m + m + m) each time. In our scenario our graph (which is actually a grid in our case) is very dense and m is approximately equal to O(n^2). Therefore this is an inefficient algorithm especially because we need to compute this multiple times from multiple possible starting locations. We first changed this to Dijkstra's algorithm which runs in time O(m + n log n) by using the Fibonacci heap data structure. We observed that this led to some practical speedup in the runtime.

Even through this improved the runtime, we still need to run this for all pairs of source and target edge locations. This means that the total runtime is O(n^2 m + n^3 log n).

For an even more efficient algorithm, we decided to use an all-pairs shortest pairs algorithm which will prevent us from having to run the same shortest path algorithm multiple times from multiple possible starting locations. To do so, we

---

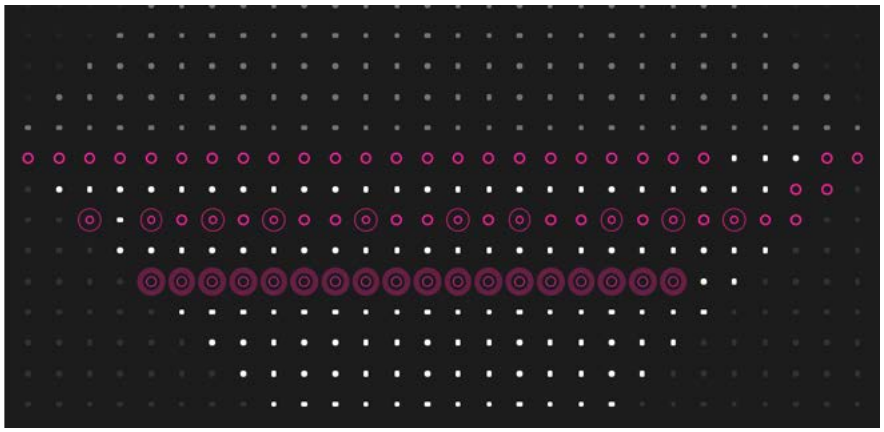


use an all-pairs shortest paths algorithm based on the Floyd-Warshall algorithm which runs in total time O(n^3). Floyd-Warshall is a dynamic programming algorithm which recurses on whether we pass through every node in the graph. In other words, starting from the same base case (the shortest path that uses no intermediate nodes), we'll then go on to considering the shortest path that's allowed to use node 1 as an intermediate node, the shortest path that's allowed to use {1,2} as intermediate nodes, and so on. This is an important speedup since recall that our graph is fully connected with m approximately equal to O(n^2). Therefore, speeding up our algorithm from O(n^2 m + n^3 log n) to O(n^3) does empirically improve the speed of our algorithm during the simulation phase.

Then, our simulation strategy is based on a function path_score() which given the game state, a number of units spawned at a particular location, simulates the movement of the units moving through the board and towards the opponent. We store the values of the path_score() outputs for all paths. Furthermore, another optimization that we made is to only recompute path_score() if some unit was destroyed, rather than every turn. This is because the score of paths will remain approximately the same unless something was destroyed along its path. This heuristic further sped up our simulations as compared to the naïve baseline.

Given the simulation and search strategies, our attacking strategy is based on finding the actions that maximize the reward. Again we use the path_score() function. By (approximately) maximizing the output of this function with respect to the number of units, their type, and a starting location on our side of the board, we are able to (approximately) determine the optimal attacking strategy for the current state of the game. We simulate the outcome of placing types and numbers of attacking units at particular locations and choose the best performing ones.

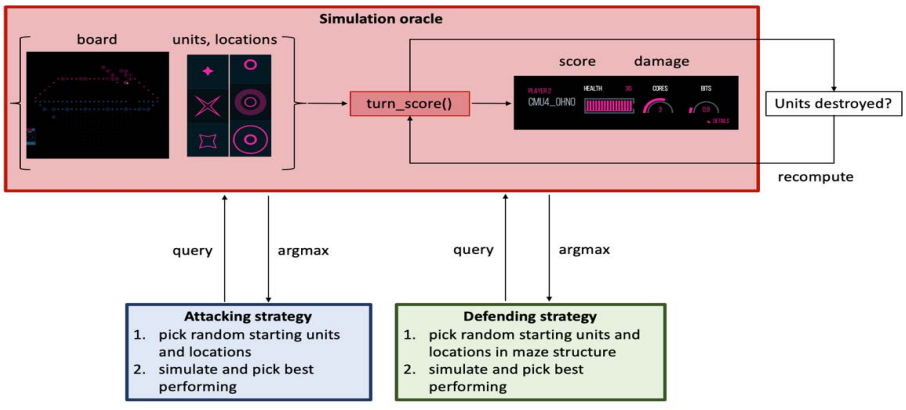
Our defensive strategy is largely based on a set of predefined locations to place the filters, encryptors, and destructors. We obtained these predefined locations by looking at experts playing the game. This is similar to how imitation learning is used to initialize reinforcement learning agents before tuning via self-play. After looking at several forums and leaderboard we decided on a maze strategy as shown below:



We found that maze structure is useful because the filter units protect us from direct paths from the opponent to our area, and also increase the distances that the opponent attacking units have to travel before they reach us. The opponent units are forced to travel through the narrow maze path and by placing destructors along this path we are able to effectively destroy opponent units. Finally the last line of encryptors provide additional stability to the defensive units in front of them.

Our maze structure is further modified dynamically by observing the attacking and defending actions taken by the opponent and choosing the areas which are particularly exploitable. For example, if we see that the opponent attacks and defends primarily from the right, we will structure our walls towards the right and leave a space on the left which our attacking units can pass through. As a result, we are able to block their attacks from the right and exploit their undefened left areas. Again, these are performed by using path_score() from the opponent's point of view to simulate their most dangerous strategy. We then simulate adding defensive units at particular predefined locations on our board and re-running path_score() to determine the how effectively these defensive units reduced the opponent's best score. Finally, we spent the last 30 minutes making a few more minor heuristic optimizations of the same sort by looking at past leaderboards and looking at how experts designed their defensive strategies as well as playing against the provided bosses.

Overall, our strategy is shown in the figure below as a flowchart across the 3 main modules: simulation oracle, attacking phase, and defending phase.



>_ENGINEERING DESIGN

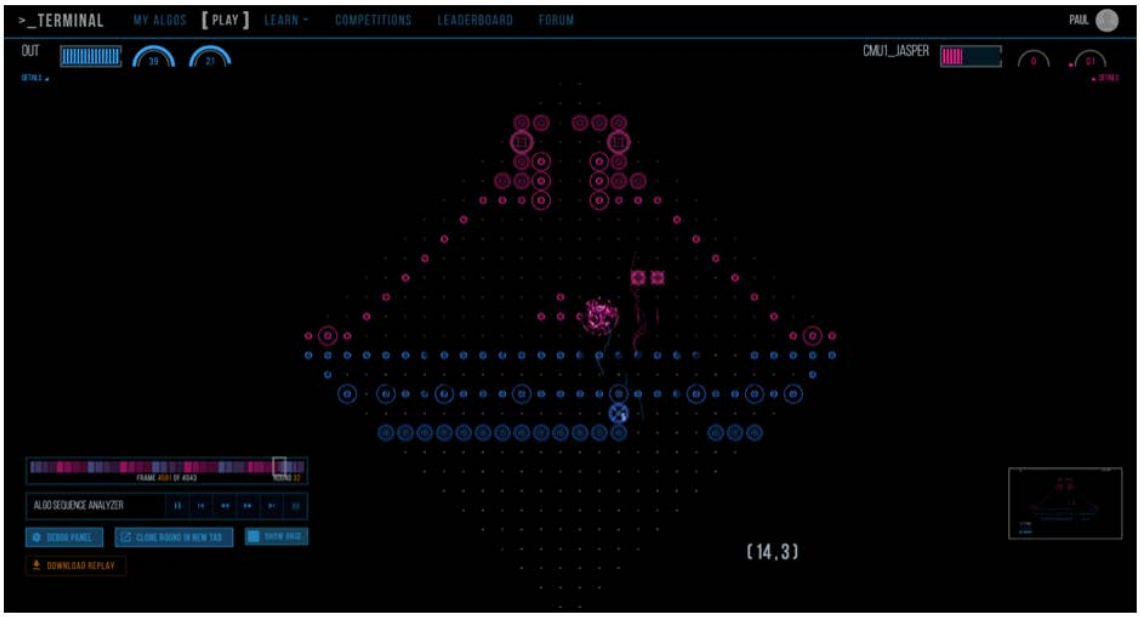
Q3.1. Please walk us through the main components of your code. How did you design and organize these components to maximize code quality and efficiency?

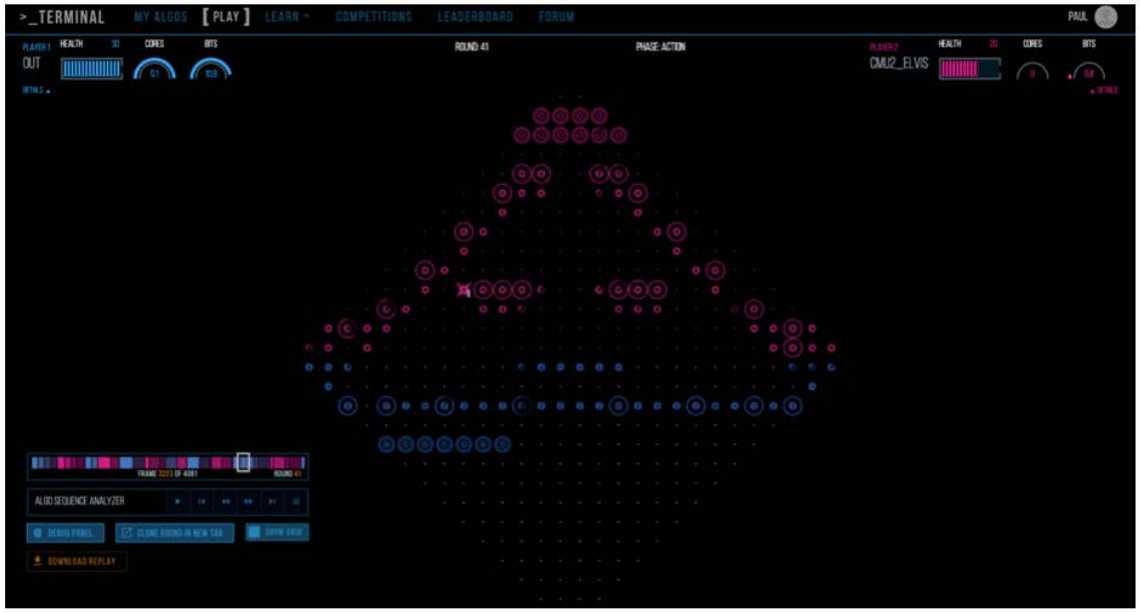
The most important helper function in our code is a function called

path_score(adv_game_state, location, unit_type, num_units)

which given the game state, a number of units spawned at a particular location, simulates the movement of the units moving through the board and towards the opponent. This function was most helpful because it served as a core component of our attacking and defensive strategies. By (approximately) maximizing the output of this function with respect to the number of units, their type, and a starting location on our side of the board, we are able to (approximately) determine the optimal attacking strategy for the current state of the game. Similarly, we can simulate the opponent's best response strategy by maximizing over the number of units, their type, and their starting location from the opponent's side of the board. This allows us to heuristically determine the regions of where the opponent's units are most dangerous so we can build defensive units in those regions. As a simple example, suppose we have relied on attacking and defending from the right, then running path_score() from the opponents side would tell us that the opponent could exploit the left side of our board. We can then simulate adding defensive units at particular locations on the left side of our board and re-running path_score() to determine the how effectively these defensive units reduced the opponent's best score. We can then approximately obtain a good defensive action (defensive units + locations) to combat the opponent's best response attack. All of these optimizations are performed approximately by using blocks of locations (e.g. left, middle, right) rather than optimizing over individual coordinates themselves.

We provide some screenshots of our algorithm playing below:





Q3.2. Which sections of your algorithm code were your most proud of, and why? Please display snippets of those sections which demonstrate the quality of your code.

We are most proud of our code that simulates possible paths and rewards. We consider this to be a central part of our algorithm that is crucial to both attack and defense. We show a snippet of this section below:

```
# First, aggregate what happens to a unit in this location
if u.move_timer == 0:

    cur_loc = [u.x, u.y]

    # gamelib.debug_write("Unit %d at (%d,%d) started shortest path" % (unit_index, u.x,u.y))
    if adv_game_state.firewall_destroyed or u.past_path is None:
        path = spf.navigate_next_move(cur_loc, u.end_points, adv_game_state, u.previous_move_direction)
    else:
        path = u.past_path[1:]

    next_loc = path[1] if (path is not None and len(path)>1) else None

    # gamelib.debug_write("Unit %d at (%d,%d) finished started shortest path" % (unit_index,u.x,u.y))

    # spf.print_map()

    # Scored a point
    if cur_loc in u.end_points:
        total_score += num_units_in_loc
        # if debug:
        #     gamelib.debug_write("Unit %d at (%d,%d) scored" % (unit_index,u.x,u.y))
    # Self Destructing causing damage
    elif next_loc is None and u.spaces_moved >= 5:
        self_destruct_targets = adv_game_state.game_map.get_locations_in_range(cur_loc, 1.5)
        for location in self_destruct_targets:
            for target in adv_game_state.game_map[location]:
                if target.player_index != u.player_index:
                    target.do_damage(u.max_stability * num_units_in_loc)
                    if attacking_unit.player_index==0:
                        total_damage += u.max_stability * num_units_in_loc
                    # Destructors that die still attack in that frame (so I'm not removing target here).
```

>_POST-MORTEM THOUGHTS

Q4.1. Compare and contrast your algorithm with other top algorithms on the leaderboard. What do you see or do that others missed, and vice versa? What would you do differently next time?

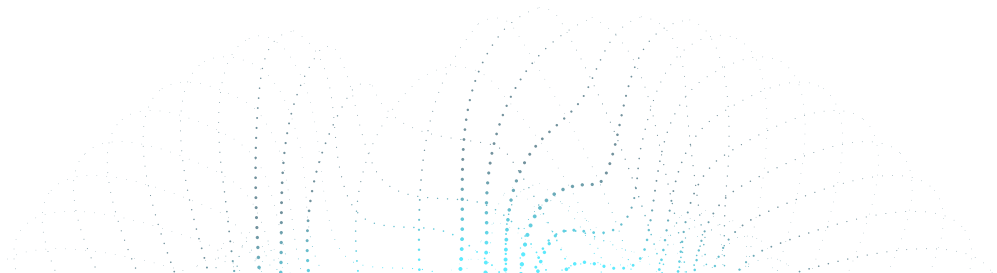
The strength of our algorithm lies in 2 areas: simulation and optimization.

For simulation, our algorithm effectively simulates possible strategies that we could take in terms of attacking and defensive strategies, as well as the opponent's best response. This is as opposed to hardcoding common algorithms for defending (e.g. maze strategy) which is commonly done even in experiences players. Our approach is a combination of both strategies since we start with a base maze structure and further optimize on top of it by incorporating dynamic adjustments to defend against opponent attacks as based on simulations.

For optimization, we experimented with various faster algorithms for shortest path computation for our simulation. We were able to improve over the BFS methods provided using Dijkstra's and faster heap structures. We also found that all pairs shortest paths help, as was storing the best scores along paths and only recomputing when units were destroyed. All of these algorithmically optimizations sped up our algorithm and allowed us to simulate possible actions taken by us and the opponent to determine good strategies.

Something that they did better than us was perhaps the use of more heuristics for both attacking and defending. We noticed that some good teams also had completely different defensive structures such as overloading defensive on one side (which also concentrates attacking units on that side) or concentrating in the center.

In the future, we would like to further experiment with heuristics that may improve performance. For example, we could change the base defensive strategy completely instead of using the maze structure. We are also thinking of using reinforcement learning (RL) to directly optimize for the optimal policy based on the inputs state space and output actions. We would likely initialize the RL method to imitate expert policies (imitation learning) before training the RL agent via self-play.



TECHNICAL APPENDIX

GAME OBJECTIVE

Terminal is a 2-player web-based strategy game played on a diamond-shaped board. Each player occupies half of the board, and the objective is to navigate units across the opponent's territory, overcoming the opponent's defenses.

RESOURCES & UNITS

Players have limited resources. Those resources are used to purchase units, which can be offensive or defensive. Offensive units move into the opponent's territory, and defensive units protect against incoming enemy units.

There are three types of defensive units and three types of offensive units. Units differ in their cost (in terms of resources), and their characteristics, like speed and range of attack. Unit names and characteristics are summarized in the table below.

Name	Cost	Description
Filter	1 Core	Cheap unit that blocks paths and soaks up damage.
Encryptor	4 Cores	Shields nearby friendly moving units making them stronger.
Destructor	3 Cores	Attacks nearby enemy moving units.
Ping	1 Bit	Fast cheap unit good for scoring. Does slight damage.
EMP	3 Bits	High range, high damage, but high cost fragile unit good for destroying enemy stationary units at a distance.
Scrambler	1 Bit	High health, high damage unit, but is slow and only attacks enemy moving units. Good for defense.

Players receive additional resources each turn, and must optimize the use of those resources given the opponent's defenses and behavior.

PROGRAMMATIC INTERFACE & STARTER KIT

Players play Terminal by writing code to automate their decision logic. They receive a starter algorithm [LINK HERE], and documentation for the game engine API [LINK HERE].

In making their strategic decisions, player algorithms are given access to detailed game state information--a matrix representation of the board positions, units deployed etc.--and can use that information along with past game state information to make decisions.

Players can also access "replay files", which contain the time series of game states from historical matches, in order to improve their algorithms.

COMPETITION STRUCTURE

Participants enter a long-duration global competition. The first global competition--"Season 1"--started in September 2018 and ended in December 2018. Approximately 12000 players entered the Season 1 competition.

Throughout the competition, players could access a global leaderboard which updated in real-time. In this way, participants got immediate feedback on the strength of the algorithms they uploaded, and they also got access to replay files which they could use to view matches they won and lost. Participants were allowed to update and re-submit algorithms as many times as they liked.

After the close of Season 1, a new global competition ("Season 2") launched in January 2019 with an end-date of April 2019.