BattleCode 2023 Strategy Report / Postmortem

Carl Guo, Ray Wang & Yuxuan Chen, for **Gone Fishin'**

March 7, 2023

1 Introduction

Yuxuan Chen (Yale '25), Ray Wang (MIT '26), Temo Toloraia (MIT '26), and I (Carl Guo, MIT '26) competed in BattleCode 2023 under the team name **Gone Fishin'**. This is our first time competing in BattleCode, and I'm honestly so amazed by how far we have came since kickoff. Our team name directly copied that of Yuxuan and I's high school robotics team (we're not very creative), and I always love the analogy of "fishing for a win." Our code is made open-sourced here. It was quite impressive how we managed to start off only as only the Top 16 and Top 32 in sprints but ended up as the 1st seed going into the finals and placing 2nd. Here is to wrap up our journey.

2 Game Overview

According to BattleCode's website, BattleCode is a game where "two teams of virtual robots roam the screen, managing resources and executing different offensive strategies against each other."

BattleCode 2023: Tempest is exactly that. Each team must collect resources (Adamantium, Mana, Elixir), navigate confusing terrains (Clouds, Currents, Walls), and place down anchors to capture islands. Whoever captures 75% of all the islands first wins the game.

2.1 Robot Types

There are 6 different robot types, listed below.

1. **Headquarters (HQs)**. Each team has 1-4 HQs per game, and HQs are where teams spawn their robots and the main source of communication. Different from many other games where a team wins by destroying HQs, HQs in this game cannot be destroyed.

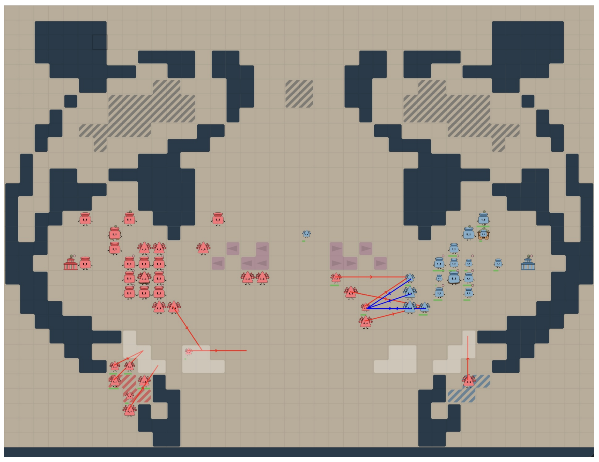

Figure 1: **Gone Fishin'**'s Theme Map, "SomethingFishy."

2. **Carriers**. Carriers go to wells to mine resources and bring resources back to HQs to spawn other minions. Carriers are usually very fast but would get slower as they collect more resources. They can also throw their resources at an enemy to deal a small amount of damage, but the resources would be lost forever. Carriers also carry the anchors necessary to capture islands.

3. **Launchers**. Launchers are the main attackers. They have a vision radius larger than the attack radius, which makes the micro-level attack strategy very interesting.

4. **Amplifiers, Boosters & Destabilizers**. These are more advanced but costly units that, in the end, do not seem to matter much in the meta-game for all the teams. We tried implementing amplifiers and destabilizers before finals, but they did not win us any games decisively.

2.2 Terrains

There are 5 different terrains, listed below.

1. **Wells**. Wells contain an infinite amount of resources (Mana Adamantium). They can be upgraded to accelerated wells and elixir wells by dumping resources into them. The latter ended up not mattering for the metagame throughout this game.

2. **Clouds**. Clouds slow robots down, obscure their vision, but also obscure the robot from robots (friends & enemies) outside of the cloud. Clouds can be on top of other terrains except for impassible tiles.

3. **Currents**. Currents move robots in different directions. Some maps can be particularly nasty where it traps robot with currents into a pit, and they would never be able to come back out again (at least with our pathfinding algo ;-;).

4. **Islands** Islands are the shaded regions depicted in this Map 1. A team needs to build anchors to capture an island, which is the main method of victory. Launchers can also stand on islands so that the other team loses possession of the island.

5. **Impassibles (Walls)**. Robots can't walk past them. They are annoying for pathfinding!

2.3 Resources

There are three resources, Adamantium, Mana, and Elixir.

1. **Adamantium (AD)** are used to build carriers, anchors, and amplifiers. Adamantium is the less prioritized resource because carriers are fairly cheap and don't die as frequently as launchers, meaning that we have smaller demand for carriers. In fact, spawning too many carriers would clog up the space near HQs and decrease production because other robots can't find a path back to dump resources into HQs.

2. **Mana (MN)** are used to build launchers, among other things. Mana is much more prioritized because having more launchers gives a team a decisive lead. For small maps, we have our launchers only mine mana, not worrying about adamantium, as the first few launcher skirmishes would determine the game.

3. **Elixir** are used to build boosters and destabilizers. Elixir wells are not generated on the maps; rather, a team has to upgrade it from a mana or adamantium well by having carriers dump resources into the well. Despite **Teh Devs** constantly lowering the cost of upgrades, they are still too costly to be put into actual use. We, like most other teams, eventually did not touch elixir-related strategies.

2.4 Tiebreaking

Once 2000 rounds hit and no team wins by capturing islands, the game goes into a tiebreak. The tiebreaking is by 1. Islands captured. 2. More anchors placed in total throughout the round. 3. Elixir. 4. Mana. 5. Adamantium. 6. Coin flip.

3 Communications Strategy

3.1 Shared array

We have an array of 64 integers of 2^{16} bits. Because the map size is between 20x20 and 60x60 tiles, each location is represented by 12 bits. This is what we stored in the shared array (a total of 821 bits, with 203 bits unused):

1. 48 bits: 4 locations of friendly HQs
2. 3 bits: Whether each of the three (rotational, horizontal vertical) symmetries is eliminated.
3. 4 bits: Whether each of the 4 HQs is congested
4. 8 bits: Carrier counts.
5. 120 bits: 5 locations of the closest wells for each of the 2 resources. 10 locations in total.
6. 140 bits: a total of 10 enemies, each containing 12 bits for location, and 2 bits for the round this enemy is reported.
7. 8 bits: 2 bits for checking whether each of the 4 amplifiers is alive (but we only spawn at most 2 in the end).
8. 490 bits: 14 bits for each of the 35 islands (12 bits for location and 2 bits for the status of their occupation).

3.2 Symmetry Detection / MapRecorder

Before Sprint 1, we assumed a rotational symmetry, as all three maps up for scrimmages were rotationally symmetrical. This decision was due to time constraints and eventually backfired quite heavily in Sprint 1. We immediately implemented a MapRecorder after Sprint 1.

Each map must fit either of 3 symmetry types: rotational, vertical, or horizontal. Each robot (including HQ) has a MapRecorder that records all the tiles that this robot has seen. Each location is represented by a char (8 bits):

- SEEN_BIT: whether this tile has been recorded before. This is also useful to avoid repeatedly scouting the same tile.
- PASSIBILE_BIT: Whether this tile is possible, which is also useful for pathing.
- WELL_BIT: Whether this tile contains a well.
- CLOUD_BIT: Whether this tile is a cloud tile.
- Extra 4 bits to store the current direction.

The symmetry possibilities are recorded in the shared array, waiting for scouts to eliminate them. Each unit also has a local copy of those possibilities to update if it has extra information. If the symmetry is not confirmed, all units will check if a newly seen tile eliminates an existing symmetry. All units report symmetry if they can write to the shared array by returning to the base.

This feature was added between Sprint 1 and Sprint 2. We first had a carrier that goes to the center of the map to scout for symmetry and then comes back to report. After Sprint 2, even though carriers can run faster, we realized that having carriers scout is a waste of their potential to mine resources, which are especially important in the early game. Having launchers scout, on the other hand, is less wasteful. We then changed to having all launchers scout symmetry on the fly as they guess a base location and move to the base.

4 Carriers & Resource Gathering Strategy

The shared array stores the closest 5 wells of Mana (MN) and Adamantium (AD). Each carrier also has a local memory of all the wells seen. If a carrier can write to the shared array, it tries to update the shared array with its local knowledge.

4.1 Mana/Adamantium (MN/AD) distribution

Mana produces launchers that kill off enemy units, so they are the most important resources in the game. We only want adamantium to build carriers so we can produce more Mana. We decide the MN/AD split based on the size of the map: the smaller the map, the less adamantium we mine.

To achieve this MN/AD distribution, we have a count of carriers in the shared array that gets reset to 0 every 100 turns. Each carrier spawned will increase this count every turn, which is its index for these 100 turns. We decide what the carrier mines, given this index.

4.2 Congestion Control

It's not always better to have more carriers as they can get stuck and slow the process of other carriers delivering resources back to the HQs. Therefore, if an HQ senses too many units around it and the carrier count is too high, the HQ will pop the congest flag in the shared array, stopping carrier production in all HQs.

Before the final tournament, we mine at well capacity minus one and make sure units can still pass through the mines without getting stuck. However, we deleted this feature to allow for a higher mining capacity of Mana (More testing could be done to decide whether it was worth it).

We also tried self-disintegrating useless carriers to avoid congestion. One issue with this strategy is that carriers would never be able to get out of a pit if there are currents blowing into the pit. Some other teams without congestion control would overflow out of such pits because of unit collisions and win some maps this way. We eventually decided only to do congestion control with carriers but not launchers.

4.3 Carrier

From the MapRecorder, a carrier calculates minable tiles of each mine: some wells have impassible tiles around them or currents blowing them away from the well. At the same time, when an Adamantium well is next to a Mana well, we mark the tiles adjacent to both wells as only available to the Mana well because Mana is more important.

If a carrier senses that all minable tiles of a well are occupied, it marks the well as congested and moves to another well. Such congestion marking is cleared each time the carrier returns to HQs to deliver resources. If the carrier cannot find a well, or if no wells are within the vision range of any HQ at the start of the game, the carrier starts a search pattern along the curve $r = \theta$ or $r = -\theta$ originating from its home HQ. It relies on the MapRecorder to avoid seen tiles and impassible tiles.

Carriers also mark wells as dangerous when they encounter enemy launchers there. They try to avoid as dangerous wells if possible when choosing where to mine, in a short period of time before that dangerous flag gets removed.

5 Launcher Strategy

5.1 Macro

The macro strategy is simple: going towards the enemy HQs. If we see a base in sensing range and no enemy units, we wait a while before going to another base. We also implemented a check to have launchers wait outside of the enemy base's attack range.

5.2 Micro: Kiting, or moving in and out of enemy vision range

Micro strategy is the most annoying part because it's very hard to tell whether a strategy works, but this is probably the most important part of the strategy. A slightly improved macro strategy might increase our win rate by 5%, but micro can do a 30%-50% increase. We learned the hard way, tuning hyperparameters and playing against an older version of the bots a million times.

General Idea. Each launcher can attack once every turn but can move only once every two turns. Therefore, we should strive to have the launcher attack every turn and make sure the enemy can't. To do so, after attacking, we move to a tile that minimizes the number of enemy launchers that can see us. And we store the enemy position when we move out of range, so we can move into range again and attack the opponent.

Here is a scenario versus a simple implementation, assuming our launchers act first.

- Turn 0: Two launchers meet. I attack you. I pull back.
- Turn 1: Both can't move.
- Turn 2: I move into range. I attack you. You attack me
- Turn 3: Both can't move. I attack you. You attack me.
- Turn 4: I attack and move back.

Assuming we act first, we can see that in every 4 turns, we attack 3 times while the enemy attack 2 times. We tried to evaluate the skirmish to only kite back in certain scenarios but eventually realized that **ALWAYS** kiting back after an attack is better. This is corroborated by **Super Cow Powers**, the Sprint Tournaments winner.

Per **Super Cow Powers**'s suggestion, we would prefer horizontal or vertical movements over diagonal ones as a tiebreaker if the absolute distance between the launcher and the enemy is the same. This is because diagonal distances are further in euclidean distance and are more likely to mess up formation.

Moreover, since going into clouds increases both movement and action cooldown and reduces our vision, we avoid going into clouds if possible when kiting and chasing. However, it could also be argued that going into cloud makes it harder for enemies to find friendly units while our launchers still know where the enemy is (based on a cached location) to blind attack and can then pop back out of the cloud. We did not fully test this implementation and did not go with it.

5.3 Micro: Chasing

We only chase a target out of attack range but in vision range when there are no other immediately attackable targets, when the target is not a launcher, or when the strength of our team is high (have 2+ more launchers than the other side).

5.4 Micro: Initial Spawning

Having the first 4 launchers spawn in the correct order actually has a significant impact on gameplay. We tried to come up with ways to have them grouped together on their way to the enemy base without communication. The most effective method we saw was to spawn the first 4 launchers in a square shape with the earliest spawned launchers furthest away from the base. Since units act based on spawn orders, outer launchers will move first. If the inside launchers move first, they will consider the outside launchers as walls and turn outward and mess the formation up. This spawning other wins about 2/3 of the game when we play against a version of our bots without it.

Some other spawning optimizations: HQs should not spawn units on currents, as that would also mess up formation. To prevent HQs from spawning units in Jail-like spaces shown in Map 2, HQs check if the nearby locations have a connected path to the location of the HQs themselves.

5.5 Micro: Target prioritization

We record an array of friendly launchers and an array of enemy launchers. For each of the enemy launchers, we enumerate how many friendly launchers can instantly attack it and calculate a ratio between their health and friendly attacks to know how many turns we need to kill the launcher. We prioritize launchers that we can "one-shot" over ones we have to kill over multiple turns.

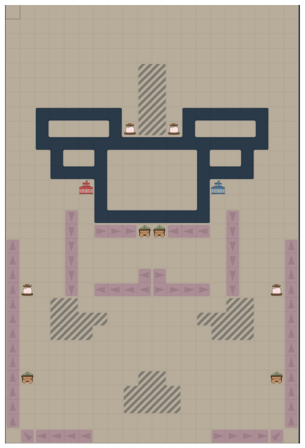

Figure 2: The infamous "Jail" where teams can spawn bots in the literal jail.

5.6 Micro: Grouping

Having launchers grouped together means that they can target one enemy and kill them off quickly, but as each robot is loaded with its own code, it's very easy to have launchers wander off alone.

Before Sprint 1, we had launchers choose the friendly launcher with the lowest ID within its vision range as the leader. The leader moves to the enemy HQ, and everyone else moves toward the leader. However, the followers often get in the way of the leader and block the leader. Therefore, we eventually removed this feature in the middle of the week between Sprint 1 and Sprint 2.

Before the final tournament, we attempted again to group the launchers together. For 10 turns after a launcher finishes a fight, we determine a friendly launcher as a grouping target to go to if it has 1. the closest distance to the enemy HQ and 2. the highest health. The launcher in the vision range closest to the enemy HQ is most likely to be on the correct path and not get disturbed by other micro strategies like kiting. If the launcher finds a grouping target, it moves toward it until the two launchers are adjacent. When they are adjacent, if Launcher A has lower health, it stops; if it has higher health, it continues the macro strategy of going to the enemy base. This would make sure that the launchers with lower health are behind and don't get one-shotted by enemy launchers coming out of the enemy HQs.

5.7 Micro: Cloud-related Guess Attacks

If a launcher can still attack at the end of a turn, it should attack somewhere because the cooldown allows the launcher to attack every round, so why not. This is especially helpful when the launcher is in the cloud or near clouds and the clouds obscure the launcher's vision. If there are tiles within our attack range that the launcher can't see (due to clouds), the launcher should try attacking those tiles. We used to have them randomly attack a tile for a whole week but then changed it to the following heuristics that we found more success with:

1. If an enemy is reported in the shared Communications array (through islands or amplifiers), the launcher should attack it.
2. We record the location of the last enemy seen. If the launcher can attack it and the location is an adjacent tile, we attack that tile.
3. We randomly sample 8 tiles and choose the one tile closest to the enemy base.

6 Island Occupation Strategy

Islands gained importance after Sprint 2 due to the buffed healing. Before that, we only produced anchors in the late game and had carriers carry them making random moves until an island was seen.

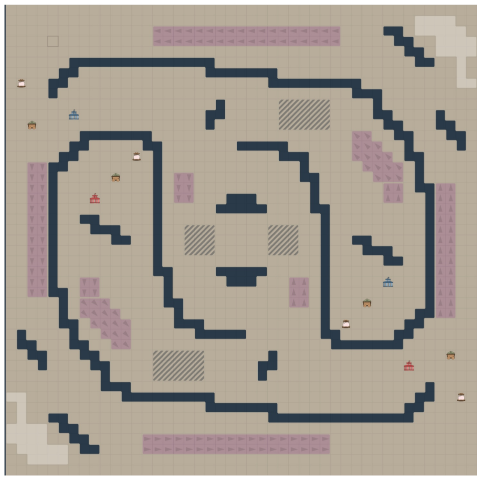

Figure 3: Our pathfinding nightmare: "Pathfind."

After Sprint 2, we have HQs start to produce anchors mid-game if we reach a certain threshold for the number of robots. This threshold increases with more HQs and decreases with more islands.

For a while, we have all our launchers follow the closest anchor-carrying carrier, which goes to the closest island to place the anchor. Testing shows that this strategy didn't work so well. The launchers either cluster too much around the carrier and clog up, or lose track of the carrier's location because of difficult terrain or cloud.

One day before the final deadline, we changed our whole strategy: instead of having launchers following anchoring carriers, we have carriers anchor islands on more important positions. More specifically, our carriers place anchors at an island slightly closer to our HQ than the enemy HQ. If the carrier finds that island already anchored by our team or encounters an enemy, it marks the island as unavailable and moves to find another island.

We have some launchers standing on enemy islands to destroy the islands. Injured carriers (≤ 120 health) go to the closest friendly island for healing. This feature, combined with the priority to deliver anchors to central islands, will most likely create a favorable situation where our launchers fight and heal around friendly islands.

7 Path Finding

7.1 Pre-Sprint 1

Before sprint 1, our approach to pathfinding was a very straightforward bug-nav that **Teh Devs** covered in the BattleCode lectures. The general idea was simple: if the robot can move in the target's direction, it moves in that direction; otherwise, it turns left and follows the boundary of the obstacle that is blocking the move until it can move in the direction of the target again. It's called bug-nav, as the robot looks like a bug crawling on the side of a wall, trying to get past the wall.

However, this algorithm was too simple to find its path out of impassibles of a "C" shape, such as the one demonstrated in this Map 3, where the robot needs to climb to the direction opposing the target to get out of the trap. To deal with this problem, we added a stack to each robot to store all past turning directions of the robot. At every turn, the robot removes movable directions from the top of the stack one by one until either the stack is emptied or the direction is not movable. If the stack is emptied, then the robot knows that it has successfully gone around the obstacle and can proceed directly to the target. If the stack is not emptied, the robot will then add all subsequent directions (rotating to the left) that are not movable at its current location to the stack, then move to the first movable direction. This solution ended up being a major part of our bug-nav solution that we kept to the finals but had a lot of room for optimization at that point.

The simple stack-enhanced bug-nav worked on most maps, and we were pretty happy about it given the time constraints. However, we found a few problems that we immediately tried to deal with after Sprint 1.

The first is with currents and clouds. Because of the time limitation and the lack of a detailed enough MapRecorder before Sprint 1, the algorithm simply treats currents and clouds as impassable tiles. We didn't advance to later matches of Sprint 1 to see our own robots play on maps where the only passible tiles are currents. We thought we were going to be so screwed on those maps.

The second big issue is friendly unit collisions. The pre-Sprint 1 algorithm treats friendly robots as impassable tiles (similar to currents and clouds), and once there are a lot of them clogged up, there is no way for them to get out.

7.2 Pre-Sprint 2

After Sprint 1, we first attempted to improve the bug-nav algorithm. The initial attempts were A* and BFS, but none of them could finish within the bytecode limit. We decided to focus on improving our existing bug-nav algorithm.

The next attempt was based on a line-sweeping bug-nav algorithm, where the robot first looks for obstacles on the line connecting it and the target and then performs a quick BFS on the obstacle tiles and determines the left-most and right-most points of the obstacle. Afterward, it determines whether it should go left or right based on distance. It recursively performs the above process until there are no obstacles on the line connecting it to the target. Unfortunately, this approach does not perform well due to bytecode limit and excessive edge cases that we couldn't all accommodate.

In the end, the major optimization we add to pathfinding is to turn the other way if the robot follows an obstacle for too long and still couldn't find a path out, which helps to prevent the robot from going a long way around the obstacle when there is a much shorter path on the other side.

In addition, support for current and clouds was also implemented. We allow robots to go on currents that travel to the target direction, the left of the target direction, or the right of the target direction. Tests are also performed on different approaches to treat friendly unit collisions. We found out that it is best to treat friendly units as walls to prevent robots from being stuck.

7.3 US Qualifier

Based on what happened before Sprint 2, it was quite obvious that over-complicating things did not help. Therefore, the primary focus was on optimizing the existing bug-nav. The major improvement before US Qualifier is the addition of a simulation process. The robot would perform a simulation for turning left or right to go around the obstacle and then pick the optimal direction.

The simulation will run for a certain number of rounds, and if it does not finish by then, the robot would assume that that direction is too far to go.

Due to the patch after Sprint 2 that allows carriers to travel at most twice per round, we removed the current constraint for carriers with a sufficiently low cooldown and let them treat the current as empty tiles. There were also other trivial optimizations done for edge cases and super-parameters.

Finally, bytecode cutoffs were added to the simulation process to make sure that the robot terminates its calculation before the end of its turn. This led to the possibility of the robot not finishing one or both of the simulations, under which case it would turn to the finished direction. If none of the simulations finish, it would simply turn left.

7.4 Finals

Before finals, we decided to remove the turning-around logic. Instead, we hard-coded into the pathfinding algorithm to prohibit robots from following map edges.

The most significant change we have is the addition of randomness. In order to improve overall performance and reduce the likelihood of robots being stuck in a loop, the robots now treat friendly units as walls for 75% of the case and as empty tiles for 25% of the case. The robots also will move randomly left or right if none of the two simulations finish in time. We also optimized the stack of directions and checked for more possibilities when circumventing obstacles. More edge case optimizations were added.

8 Amplifiers

For us, amplifiers are only useful to allow nearby launchers to report enemy positions around clouds and update island status. Therefore, we only produce them in the late game and have at most two of each alive at any time.

We also have tried having defensive strategies and amplifiers as the moving command centers. However, such strategies didn't beat our rush strategy in self-play, as it often yielded important central islands and mine locations to the enemy, and still struggled to defend against unintentional base attacks.

9 Abandoned Elixir Units

1 launcher = 45 Mana = 200 Health = 10 damage per turn

1 booster = 150 Elixir = 400 Health = 10 percent boost to friendly each 10 out of 15 turns

1 destabilizer = 200 Elixir = 300 Health = 5 turns of 10 percent slowdown and 50 AOE damage every 7 turns

With some calculations, it was clear that booster is never worth it.

Destablizer might be worth it in some cases, especially with groups of 4 that can almost instant AOE kill launchers. However, they can only be produced in small numbers and don't have enough health. Moreover, it's pretty hard to write the micro-level logic for correct grouping and coordination, and we have already written a lot of complicated micro-strategy for launchers. In our self plays, it has lost more games than won.

I know some teams make elixir units only if they have more than one mana mine and can keep up launcher production, such as **4 Musketeers**.

10 Development Timeline

10.1 Pre-Sprint 1

Being the newbies we were, we started off knowing nothing about how to approach the game. We started off trying to implement the simplest ideas: having carriers only mine Mana, having launchers rush enemy bases, assuming a rotational symmetry so we don't need to scout, and implementing a simple bug-nav. We didn't try to capture islands and could only win based on resource tiebreakers. Despite the rush, some of the features worked surprisingly well and got us to Top 16 of Sprint 1 Tournament.

10.2 Spring 1-2

We learned a lot more about what other teams were doing and became more familiar with the game itself. We implemented a successful symmetry checker, implemented congestion control, optimized launcher micros like target prioritization, removed the launcher's leader-following logic that caused us so much trouble, added a rudimentary island logic, and tried a bunch of different pathfinding algorithms that didn't work in the end. Unfortunately, with some seeding issues, we were only Top 32 for the Spring 2 Tournament.

10.3 US Qualifications

We tried out more launcher micro strategy optimizations, mostly kiting and cloud-related guess attacks, which finally worked after so many hours of self-testing and honestly made so much difference. We finalized improving the rudimentary bug-nav algorithm. We tried some defensive launcher and amplifier stuff that didn't work as well as intended. With an ELO burst and a good seeding due to our launcher micro improvements, we became a finalist.

10.4 Finals

We implemented launcher grouping, found better spawn sequences, changed our island logic to capture central islands, a risky but high-reward strategy given the recent map-designing trends to have islands on more strategically important places, adjusted our MN/AD distribution to be more Mana heavy, and further optimized pathfinding.

10.5 Final Project Line Count (excluding utils)

143 src/bot1/Amplifier.java

664 src/bot1/Carrier.java

516 src/bot1/Comm.java

35 src/bot1/Constants.java

253 src/bot1/Headquarter.java

553 src/bot1/Launcher.java

248 src/bot1/MapRecorder.java

87 src/bot1/RobotPlayer.java

383 src/bot1/Unit.java

2882 total

11 Thoughts After Finals & Conclusion

It was great meeting all the finalists in person, including those who flew into Boston in such miserable weather. Congratulations to **Producing Perfection** for winning; they absolutely deserved the spot, and we have learned a lot from them. The finals were so close and were definitely some of the best BattleCode matches I've watched.

Some thoughts on the finals match: we know the clear reasons why we lost the last two maps right after we lost. On the second to last map, our scouting algorithm could not find the further away wells promptly. This was due to a designed spiral pattern that our carriers scout in, as the pattern helps maximize the scouting position if we have our base closer to the center of the map. As **Producing Perfection** found their wells earlier than we did, we knew the game was going to a Round 5.

On the final map with the long islands, our strategy to capture the center islands came back to haunt us. As **Producing Perfection** captured a long island near their base early and allowed launchers to heal, we could not take it down because **Producing Perfection** had their units standing on the island. **Producing Perfection** said their island capturing is randomized. Our anchoring strategy made us wait to capture a center island, but that was quickly destroyed by the opposing launchers as no friendly launchers were actively defending it. We predicted that maps with center islands became a trend going into qualifications and finals because islands and the corresponding healing became more strategic. We did not foresee a map like the last one at all. I would argue that while our center island strategy won us a majority of the mainstream, "normal" maps, it made us worse on strange maps like the last one than a more evenly distributed random algorithm. Whether we should do map-specific optimizations like capturing center islands and bet on what kind of maps **Teh Devs** would draw up became a question of much debate recently in our team (and we'll see if similar strategies can still work out in BattleCode 2024?).

Finally, shoutout to **4 Musketeers** for open-sourcing their utils library from 2022, everyone who engaged in the Discord channel, and everyone who wrote postmortems from past years.

BattleCode was a blast. We had a lot of fun, learned a lot, and would definitely compete again or recommend our friends to compete!