Battlecode 2023 Postmortem

Don't @ Me

George Zhang
Henry Liao
Parum Misri
Ray Guo

Table of Contents

1) Introduction

1.1) About Us

We are a team of four second-year college students, with Ray, George, and Parum from the University of Washington - Seattle, and Henry from Georgia Tech. George and Henry participated in BC 2022 - Mutation while BC 2023 - Tempest is Ray and Parum's first time doing Battlecode.

Because of how much we enjoyed Battlecode as a team, we hope to make this postmortem as much a celebration of the competition as it is a strategy report. We hope you enjoy it!

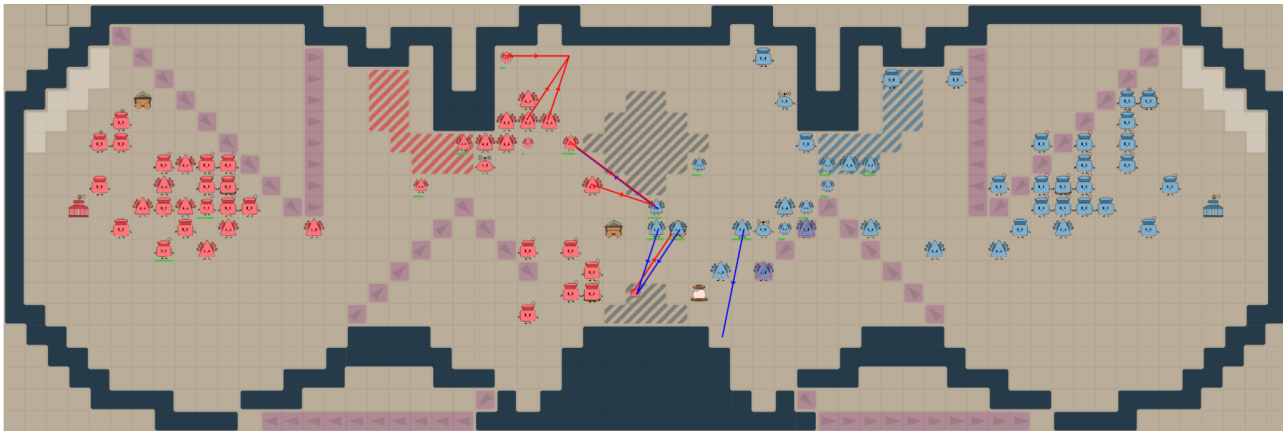
BC2023 Github Repo for *don't @ me*

2) Game Overview

2.1) About Battlecode

Battlecode is the longest standing annual AI programming competition hosted by the Massachusetts Institute of Technology (MIT). This AI competition, which changes its main theme every year, generally lasts around 1 month and is fought between teams composed of up to four people. During the competition period, teams can freely submit their own bot to the Battlecode website. From here, one team's bot will duke it out against another team's bot on a virtual battlefield built by **Teh Devs** in a ELO rating system. There are four main competitions during the season, with Sprint 1 hosted during the first week, Sprint 2 hosted during the second week, Qualifying tournaments hosted during the third week, and the final tournament hosted the final week. All tournaments are held online, with the exception being the final tournament, where the top 12 US teams and the top 4 International teams are flown out to the MIT in Boston to compete for over $20,000 worth of prizes.

2.2) Perspective on BC 2023 - Tempest



Battlecode 2023 is based upon a team controlling a faction of robots to capture as many "sky islands" as they possibly can. The map is based upon a 2-D grid, with the size ranging from 20 x 20 to 60 x 60.

Some form of symmetry is guaranteed on the map. On the map, there are certain tiles that are unique and represent a hazard, a resource, or a boost. Robots are produced with resources collected from wells, as well as anchors which can be used to capture sky islands.

Robots:

HQ. Headquarters are immovable and indestructible bastions that house resources, spawn robots, and create anchors. They are also able to write into the comms array (see 4.5) Communications), and can also generate small amounts of resources every turn. Carriers are able to deposit resources into the HQ. HQs can also utilize the most bytecode of any unit type, which means the vast majority of comms interfacing takes place here. We cycled through several strategies involving HQ defense, such as having launchers regroup at HQ when enemy units are sighted, but ultimately we realized that falling back to defend HQs wasted far too much time, and rarely if ever gave us a competitive advantage. We eventually streamlined our philosophy to be simply this: the HQ exists to receive resources, produce robots, and share information.

Carriers. Carriers harvest Ad and Mn from wells and drop it off at HQs, can throw the contents of their inventory to deal damage, and can carry anchors to capture islands. There are two main questions we faced in our implementation of the carriers: whether we should spawn them before launchers, and what resource type to go for first. The first question was mostly resolved after the sprint 1 updates which allowed HQs to spawn units much faster. The second was resolved when we devised a well-assigning system that guaranteed a balanced resource harvesting split, allowing us to properly grow our economy in the early game. As for the carriers' ability to throw the contents of their inventory to deal damage, we decided that if a carrier was damaged, they would immediately throw the contents of their inventory at their attackers and run away, as they would otherwise usually just die and lose their resources. Finally, regarding anchoring islands, in the first couple weeks it didn't have much impact, but as the competition progressed and updates made it more viable, we discovered that it was extraordinarily useful for establishing communications, healing units, and generally winning the game. Carriers have proven themselves to be an indispensable part of any strategy since day 1, but innovation and iteration have helped us make the most of their diverse skillset.

Launchers. Launchers are Mn units and represent the basic attacking units of the game. They are able to deal damage to an enemy robot within its action radius, even if the enemy robot is obscured. When devising our strategy for utilizing launchers, we focused on three key features - their micromovements, their pathfinding, and their grouping. Kiting, where a launcher steps away from the target after firing a shot to avoid taking damage, was an essential part of our combat strategy, allowing us to win otherwise unfavorable matchups. Pathfinding and grouping went hand in hand - the launchers needed to efficiently find their way to their destination, and were most effective when working together as a group. We were able to create "squadrons" of launchers, where small groups would traverse the map together and attack targets together. Improving the cohesion and maneuvering of the launchers proved to be an extremely important part of the game, and mastering these aspects most likely meant mastering Battlecode as a whole. We found that launcher rushing, where we sent a large amount of launchers early in the

game to attack the enemy HQ, was a highly effective strategy, often deciding the fate of the game in less than 200 rounds.

Signal Amplifiers. Signal Amplifiers are Mn and Ad units that allow for robots within its vision range to write to the comms array. We believe that they should be produced in moderation, as too many signal amplifiers means alot of overlapping in signal, but to little signal amplifiers one may find a lack of coordination within the team. Amplifiers' main use was for exploration and mapping out points of interest, as they are the only unit (aside from HQs, which cannot move) that can both read from and write to comms at any point. By gathering more points of information and building a more complete picture of the map, in combination with verifying symmetries, amplifiers allow us to predict things like the location of enemy HQs and resource wells. Due to the intense limitations on available information in the game of battlecode, the freedom of communication provided by amplifiers can be remarkably powerful.

Temporal Destablizers. Destabilizers are Ex units and represent late game attacking units. For their attack, they are able to select a tile within its action radius to attack. This attack slows enemy bot action and movement cooldowns for a large, circular area for 5 turns and deals damage to all enemy robots caught within the attack. Elixir was far too slow to successfully utilize in the rush-heavy metagame, so we ended up not ever using destabilizers. However, there are some theoretical use cases in which it could be strategically advantageous to have a destabilizer. Launchers are most effective when grouped up, so the area-of-effect damage provided by destabilizers would ostensibly be a powerful counter to the dominant force in the metagame. Carriers are also almost always grouped around wells, so a single well-aimed destabilizer could swiftly cause significant disruption to the opposing player's economy.

Temporal Boosters. Boosters are Ex units and represent late game support units. For their action, they are able to boost robots within their action radius, reducing the movement and action cooldown for nearby robots. This boost can stack up to three times. Much like destabilizers, thanks to the unusability of Elixir, we never ended up implementing and using boosters. However, there are still some interesting use cases that we considered. Using boosters in combination with carriers could significantly increase the economic output of a single well. Including a booster in a launcher squadron could increase the effectiveness of their micro, allowing them to both shoot faster and kite more quickly.

Anchors:

Standard Anchor. Costing an equal amount of Ad and Mn, a single anchor canbe carried by a carrier. An anchor can be used to capture a neutral sky island. Once used, a standard anchor allows for a certain amount of healing to allied robots. We always knew that since islands were a major part of the game that the standard anchors would be critically important to winning the game.

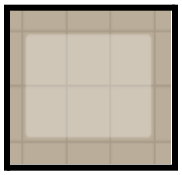
Accelerating Anchor. Costing a large amount of Ex, accelerating anchors share the same traits with standard anchors except they have higher HP and any island captured by an accelerating anchor heals allied robots for more HP, as well as decreasing their cooldowns. Much like

destabilizers and boosters, we did not use this anchor due to the unavailability of Elixir. While in theory they are useful for holding down islands, when an enemy snowballs over an island it's very difficult to play to retake it, regardless of the health of the anchor on the island. Even if you did have elixir, you would be better off with the offensive power of a Destabilizer.

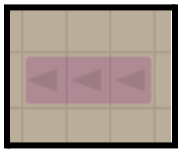
Map Tiles:

Standard Map Tile. Usually covering most of the map, these are tiles that have no special trait attached to them and are standard passable tiles.

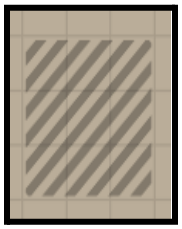
Walls. These tiles are impassable and cannot be removed. Unlike previous years, there were tiles that could not be passed, which made BugNav a reliable pathing strategy.



Clouds. These tiles obscure vision. For every robot that enters a cloud, their vision is decreased, but they are also obscured to others not within the cloud tile. Clouds also slow both movement and action cooldowns for any robot on the cloud tile. Unfortunately, vision being obscured meant that pathing and decision making would be extraordinarily limited while fighting into clouds and while in clouds.;



Currents. These tiles boost robots in the direction the current arrow is pointing in at the end of the robots turn. Current and cloud tiles cannot overlap. These tiles could ostensibly be used for more efficient pathfinding, as bots could "ride" currents over longer distances more quickly than walking there, but since so little information is initially available about what direction the currents flow and which way they lead, the only optimization we made for them was to treat the currents as impassable when you were trying to move against them (or rotate around a block of currents using bugnavs..



Sky Islands. These tiles represent an island that can be captured by any team with an anchor. There are anywhere from 4 to 35 islands, with each island having a maximum area of 20 units. A captured island is changed to the color of the team that captured the island and may heal allied robots that are 4 units within an ally island's boundary. A captured island may be uncaptured by the opposing team. Sky islands are one of the most important map features in the game, since maintaining control of 75% of the existing islands is an automatic win. The buffed healing provided by anchored islands allowed for defensive play to become slightly more viable - damaged launchers could retreat to a friendly island and heal up rather than fruitlessly push forward and end up dying.

Wells. These tiles represent indestructible resource wells that can be used by any team. Each well represents a single resource and contains an unlimited amount of that resource. Wells can only be accessed by carrier robots adjacent to its location. Wells can also accept resources to upgrade it. If a well gets its own resource pumped back into it, it becomes an upgraded well for that resource and the draw rate is higher. If the well gets an opposite resource pumped into it, it becomes an elixir well. Wells are considered passable terrain. Upgraded wells and elixir wells proved to be competitively unviable due to the tremendous loss of tempo that resulted from spending 600 of any given resource on producing more resources rather than producing more

units. Unupgraded wells, when fully utilized, provided more than enough resources to fully occupy our carriers in the early and midgame, and by the late game a winner has almost always been declared.

Resources:

Adamantium. Otherwise known as Ad, Ad is solely responsible for the production of more carriers, and is a contributing resource to creating anchors and amplifiers. Ad is found in Ad wells, and is also passively produced by each HQ. As the season progressed, we realized that going for more Ad-heavy build orders was only useful in the meta insofar as it helped you generate more mana and by extension more launchers.

Mana. Otherwise known as Mn, Mn is solely responsible for the production of more launchers, and is a contributing resource to creating anchors and amplifiers. Mn is found in Mn wells, and is also passively produced by each HQ. Due to the fast-paced, rush-heavy earlygame dictated by Mn-based launcher skirmishes, Mn proved to be significantly more important than Ad in early rounds, to the point where we had a ratio approaching 3 Mn carriers for every 1 Ad carrier as we filled out our initial wells.

Elixir. Otherwise known as Ex, Ex is solely responsible for the production of accelerated anchors, destabilizers, and boosters. Elixir is only obtainable from elixir wells. As stated previously, the loss of tempo from attempting to create an Elixir well prevented us from utilizing Ex in any real fashion.

Win Conditions:

The primary goal for each team is to gain control of 75% or more of the sky islands on the map. If this occurs, the game automatically ends and the team with control of the islands wins the game. If no team achieves this condition within 2000 rounds, tie-breakers based on anchors produced and resource count take place. In early rounds of development, such as sprint 1, most games were settled by tie-breakers, due both to the fact that most teams neglected to implement island capturing mechanics, and the fact that anchoring islands was not as powerful as it would become through later patches. Once anchoring was buffed, however, it became a dominant strategy in the metagame. Teams that didn't properly implement island capturing could lose a match even if they had a dominating economy and unit advantage. Because of this, we made sure to implement mechanics for capturing islands even if we were falling behind economically - it could help us eke out a win in otherwise hopeless scenarios.

3) Development Timeline

In this section, we give a brief overview of what occurred during our team's journey through this season of Battlecode. We had a habit of changing our team quote daily to a line from *The Art of War* as a joke to lighten the otherwise (often) stressful process of development.

3.1) Sprint 1: "*There is no instance of a nation benefitting from prolonged warfare.*"

- *The basics.* As with any competition, it always helps to start with the basics. For the first day, our team primarily spent the day going over the spec to understand the game. At this time in development, we had the idea that we would first start off with basic carrier resource gathering, basic launcher micro, and basic island capturing, while following a pre-laid out robot control system (see Robot Control Flow). Within the first 48 hours of release, our team was able to get this version of the bot finalized and uploaded to the ladder. For this early bot, our carriers would move in random directions until they found a well, in which they would extract the resource from it and return back to the HQ. Our launchers would explore randomly until they saw an enemy, in which they would attack the enemy. Our HQ would eventually produce anchors, where a carrier would then take the anchor and move randomly until it sees an uncapped island. Although not anything impressive, given the lack of obstacles on the few maps that the ranked system were using, we were able to climb to a formidable amount of rating to jump start our journey the right way.

- *Rush, rush, rush*. As the "getting to know the game" period for every team ended, it soon became apparent that whichever team won the beginning launcher duel would go on to win the whole match. This was also bolstered due to the small map sizes that the ranked matches were held on. Because of this, our team shifted our attention completely to launcher micro and carrier resource gathering optimization. We abandoned the idea of capturing islands because by the point one was able to produce the resources to build an anchor, the game was already won or lost. However, given Henry and George's experience with battlecode the previous season, we knew that it wouldn't be like this for the whole season. We made sure to generalize and make our code modular, alongside working on island capping in case it ever became relevant ever again. On top of all this, we implemented pathfinding that was more general and didn't hardcode any values for specific maps.

- *Sprint 1 Tournament.* Heading into the Sprint 1 tournament, we were placed 25th seed on the ladder. Because of this placement, the team did not have high hopes during the coming Sprint 1 tournament. However, because the tournament matches are played on new maps, we soon realized that a lot of teams had hardcoded values according to the three original game maps. Because of this, some teams' robots did not function as intended and were subsequently upset by other teams. Given the introduction of new maps, our more generalized code (especially in regard to exploring map symmetries) meant that our bot could compete and even beat upper level teams, despite our seemingly poor standing compared to them. We were able to upset the 8th seed **Kryptonite** in round of 32 and lost to the 9th seed **noBFSplz** 2-3 in round of 16.

3.2) Sprint 2: "*Ponder and deliberate before you make a move.*"

With our newfound motivation to improve on our bot after the results of Sprint 1, our team got right back to work. Alongside Sprint 1, the new maps from the Sprint were added to the pool of maps. These maps had more variance than the original three maps, allowing our bot to climb rating without any adjustments. As Sprint 2 came along, the newbies of the team also gained a greater understanding of the game and the code behind it, allowing for faster bot development.

With the start of Sprint 2 also came a plethora of changes. The first major change to the game was that all robots got a durability buff, since health values were increased while damaging abilities had their damages reduced. Alongside these changes, captured islands could now heal allied robots. Given these updates, we made the following developments:

- *Communication…* Even with the durability update, it quickly became apparent that the updates would have minimal impact on the meta. Spamming launchers the moment an HQ could build a launcher was uncounterable. Any thought of trying to build anchors or trying to dump resources into wells to upgrade them against a launcher rush team meant almost certain loss, even on larger maps. Because of this growing trend, we decided that the best teams were the ones that could maximize launcher production. We decided to dedicate most of our carriers to gather from Mn wells while sending few carriers to gather from Ad wells in order to get more launchers up faster than opposing teams in an attempt to out blitz them. Alongside these changes, our team also believed that our launcher micro and carrier micro were in a good enough state that we could start shifting our focus to setting up the comms system. With a comms system, we enabled macro based decision making. We were able to determine which wells carriers should go to, and where launchers should rush. Given our newly created comms system, we also implemented amplifier code to allow for more robots to communicate with each other.

- *…and Deliberation.* With the increase in communicative powers of our robots, more complex decisions were enabled on an individual and team-wide basis. Our bots were able to determine and communicate map symmetries, island locations, well locations, give attack commands, and more. It allowed for carriers to be given well locations on spawn, anchoring carriers to find islands to anchor quickly, and allowed launchers to coordinate attacks. It added a whole new layer of gameplay - this change would be seen throughout the ranked later as well. Games were quicker and the robots moved and coordinated in a more organized fashion. Several teams would adapt well to the change in pace of implementing comms, while others didn't. Thankfully, our team would navigate well through building up and implementing the comms system and we soon found ourselves constantly hovering the 1st and 2nd page of the leaderboards. Our robots were now able to deliberate with each other, but our team also had to deliberate what was important and what wasn't. With the new island healing feature, our team needed to decide if we were to implement the feature or not. We had tested some "fallback" code which allowed launchers to fallback to a captured island to heal if it was low health. However, after deliberation, we decided that the value we were getting from the island heal was too low and we would rather have the map presence that the launcher presente, even if it was low health. Unlucky for our launchers, but the best for the team.

- *Sprint 2 Tournament.* We entered the Sprint 2 tournament with a much better standing than Sprint 1, coming in as 10th seed. We were confident in making it to the round of 16, but didn't expect to get much further. However, just like in Sprint 1, we pulled an upset and managed to beat the 8th seed **Bruteforcer** out 3-2 in a set of very small maps to make it to the top 8, losing to the 2nd seed **Xenoblade GOTY** in a 2-3 loss. Many of the maps were more advantageous to us than the ones on the ladder due to a high density of wells, which enabled us to determine map symmetry and assign carriers using communications efficiently.

3.3) US Qualifiers: "*The greatest victory is that which requires no battle.*"

From the beginning, our team always had the goal of qualifying to the final tournament. What seemed like an impossible task to accomplish before the Sprint 1 tournament suddenly seemed realistic after our performance in the Sprint 2 tournament. Given our string of good performances, we decided to put our best efforts forward to our US qualifying bot.

With the conclusion of Sprint 2, the final major patch notes for the season were released. The major changes included reduced resource costs to transform wells into upgraded/elixir wells, cheaper anchor costs, reduced launcher damage, decreased launcher cost, reduced amplifier cost, and a big increase in how much a captured island heals allies for.

- *Optimization.* Like heading into Sprint 2, we were confident about our carrier and launcher micro. However, there was still good room for improvement. For this week, we fleshed out our carrier micro, making sure that every carrier was doing its job as intended,and that our launchers were engaging in the right fights. We also fixed a heap of bugs that became apparent after thorough testing, such as issues with anchoring code, errors in the comms array, and more. Anchor spawn rate, amp spawn rate, and how many carrier well assignments were also values that were also subject to optimization. Upon reflection, we realized that we were likely only able to optimize every small detail about our robot because the meta still mainly revolved around launcher-rushing on small to medium sized maps.

- *Playing the objective.* In the latter days of the week leading up to the US Qualifying Tournament, we had noticed matches where other teams would capture early islands in an attempt to get value from island healing. This wasn't a common trend, however, as many teams still seemingly only produced anchors once the game was already in a winning state and just wanted to guarantee the win. Despite this, through several rounds of testing, our team decided that we would implement early island capturing and reimplement the launcher fallback code that we had decided against during Sprint 2. With the updated healing values, we found that there were very few scenarios where early anchor production would cause us to lose a match, but found that there were several special scenarios where getting value from island healing would enable us to win a match. On top of this, our optimized amp code allowed for frontline launchers to get information about potential fallback islands. We also had a hunch that with the updates, **teh devs** would create island heavy maps for the US Qualifiers in order to reduce launcher rushing and promote island capping instead.

- *US Qualifiers*. Going into US Qualifiers, we were 8th seed. We were fairly confident that we would qualify to the final tournament, since even if we lost the winners bracket match to qualify, we would, in the losers bracket, eventually face the loser of the 1st seed and the 16th seed to qualify, in essence having the easiest losers bracket matchup if no upsets were to occur.. During the actual qualifier, however, the 1st seed **4 Musketeers** got upset by the 16th seed **Baby Ducks**, meaning that we had to win our winners bracket matchup - otherwise we'd eventually face the wrath of the 1st seed in the losers bracket. In our matchup against the 9th seed **John Silver** to qualify, the maps were relatively large, with numerous large islands to capture. Because of this, our early island capture and launcher fallback mechanism, alongside comms and pathfinding were able to demonstrate their full power - even drawing remarks from the commentators due to how apparent the strategy was. Launchers on these maps were able to generate significant value from island healing, soaking 5-9 hits before healing back to full. We would win 4-1, qualifying us to the final tournament!

3.4) Finals: "*In the midst of chaos, there is also opportunity*."

After three weeks of grueling theory-crafting, coding, and optimizing, our team was offered the opportunity to go through this process one last week. Unfortunately, reality hit as most of our team was backlogged on several weeks of unfinished homework, exams, and projects, which made it difficult to respond to changes in meta on the leaderboards. We would have loved to do a revamp of launcher micro and carrier optimization, but unfortunately we were unable to.

For this final week of competition, no changes to the game were made.

- *It's a numbers game.* Our strategy for the final competition doesn't differ any much from our US qualifying strategy. We want to get the most value we can out of launchers, carriers, and islands. For this, we made several value changes regarding spawn rates regarding anchors, amplifiers, and we hope that this can help us gain a more optimal strategy compared to our opponents during the final competition. We also fleshed out our island health system in hopes that it can provide us with more value.

- *Bug fixes.* During the competition, we noticed that aspects of our pathfinding and carrier code seemed to be unable to handle the significant amount of clouds, currents, and islands on the new maps. After some analysis of the issue, we discovered numerous pathfinding and carrier bugs that were subsequently fixed.

- *Pre-finals.* As of writing this, our team has just landed in Boston. We arrived one day early since us West Coast kids have never visited the East Coast before. We're equally as excited at touring the city of Boston as we are to compete in the final competition! We're happy to have made it to the final competition, and we hope that we can make a good run during the competition. We're currently looking at going in as 8th-12th seed at the tournament.

4) In-Depth on Strategy

4.1) Robot Control Flow

Each turn, each robot will run a series of functions based off the sense-think-act paradigm from robotics theory.

- *Sense.* Every turn, a robot first needs to sense its environment and changes in communications. This includes sensing nearby enemies, allies, and map elements and storing them so that if we wanted to iterate over them several times we would not have to re-sense anything, wasting bytecode.

- *Think.* Based off of sensor data, comms, and the robot's current control state, we would decide the robot's next control state. Control states included general "plans" that a robot would follow, such as pathing to an attack command, gathering at a specific well, exploring etc. Based on the robots' selected control state, they would then attempt to carry out a series of actions.

- *Act.* In this part, robots would run algorithms to carry out the objectives of their control state, which generally comprises of finding the best action and movement based off of algorithms. Robots would also write information to the communications array if they could at this stage.

4.2) Robot Micro

<u>Launchers:</u>

- *Attack priority.* Every single robot can run **rc.senseNearbyEnemies()** to return an array of enemy robots that are within their vision. If there is only one enemy, it makes sense to attack the one enemy. However, if there are multiple enemies, we need to be able to effectively determine which one wass of the "highest value target". For this, we had a had a calculation that determined the attack value of a certain robot: it was (Robot Type Value) + (Amount of HP Attacked) + (Enemy Missing Health)/2 + (100 on kill). This strategy enabled us to focus more important enemy robots, maximize the damage dealt, and focus low-health targets, especially if they could be killed. We always prioritized robots already in action range.

- *Kiting.* Kiting is an important concept that is applicable to this game. The concept of kiting revolves around attacking a target and moving away from it. This is extremely effective in scenarios where our launcher has the first move over the opponents launchers during a round. By using kiting, our launchers are often able to attack the opposings launcher first. Afterwards, it can step away from that launcher, potentially even leaving its vision radius, essentially getting a free hit. Because of this, we do it every single time movement is available. In general it also allows for other allies to close the gap - if the lead launcher makes first contact, it can kite back to allow other launchers around it to step in to gain space.

- *Step up.* If a launcher sees another ally is already in attack range of an enemy, it will allow itself to step towards the enemy rather than kite away from it, as if the unit is not a destabilizer, it can only hit one enemy in a given turn, meaning that you might as well move to attack.

- *Falling back.* A launcher's state is set to fallback if it is not in combat, if they are less than half health, and if they have a fallback island.

- *Fallback island.* A fallback island is the closest captured island to the given launcher. Once an island has been captured, information regarding the island is added to the comms array. This means any launcher within comms range will have a fallback island - if there are multiple captured islands, it will iterate through all of them every turn to find the closest one.

Island healing. The moment the launcher arrives onto a captured island tile, it will stop until it heals to full, where updateState() will reassign it to another state. It will only leave this state before reaching full health if it sees an enemy.

Follow first. When we first started making launchers, we were losing early 3v3 duels left and right, with the opponents launcher seemingly attacking as if they had comms while our launchers were getting picked off. After analysis, we soon realized this was because the higher level team's launchers would stick together, staying constantly adjacent and making optimal moves to take grouped trades against isolated launchers. With this knowledge in mind, we implemented a feature where launchers would follow the first person near them that moved. The basis of the feature revolves around **rc.senseNearbyAllies().** Each individual launcher would use this function to sense its nearby allies for the turn. On a movement turn for the launchers, it will iterate through the new positions of nearby allies. If a nearby ally's location had changed, this meant the robot must have moved, so the robot follows that one. If none of the locations changed, this likely means that the current launcher is the "first" robot to move in the pack. This way, for every group of launchers, launchers will follow the "first" launcher that moves, making sure that they stick together.

- *Explore on evens.* Because launchers can only move once per every two turns, we decided that launchers all only move on even turns, unless they are in combat. This helps launchers stay together and helps launcher stay safe during exploration.

- *HQ camping*. Camping only occurs when the robot state is not set to Combat, Pursuing, or Fallback, and detects a HQ within its vision range. Camping code rarely triggers because of how rare all of the conditions are met, but the code is best described as "win more" code, since by the time one can camp the HQ they have likely already won the game. Camping code simply directs the launcher to wait outside of the HQ's passive attack range. If an enemy is detected, then the state is switched to combat. Once there are no more enemies in range, it simply returns back to Camping code barring any other state update.

- *Cloud attacks.* If a launcher still has its attack after their move, and it will also have its attack next move (calculated by sensing the cooldown of its current tile) and it sees a cloud, they will randomly attack it if they can't sense inside. There is no harm in performing this move, since you can't damage allied units, and there is a small chance one can get a free hit on a hidden enemy inside the cloud.

Carriers:

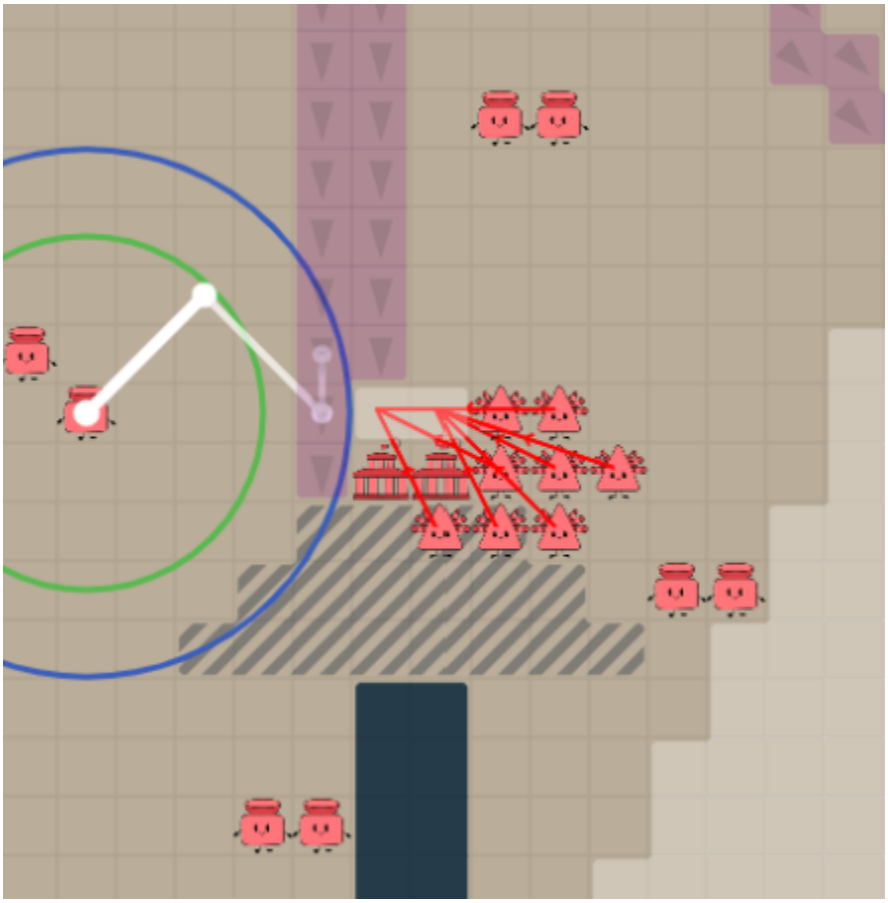
- *Anchoring.* When a carrier returns to HQ, if it detects that an HQ is storing an anchor, it will take that anchor. Once it takes the anchor, it will make an attempt to go to the closest neutral island and cap it. This island is usually communicated to the carrier through the comms array. However, if it cannot access comms, it will move to explore islands. If it detects a neutral island, it will move towards it to cap it. Once a carrier has placed its anchor, it writes into the comms array a new captured island. This is possible because captured islands grant nearby allies access to the communications array. The carrier is now set back to a gathering state.

- *Initial Gathering.* Initially, we assigned carriers to wells by reading from communications the location in the command slot of the HQ that spawned the carrier. The carriers would accept the command by wiping it from the shared array, after which the HQ would write a new command for the next carrier. This worked well enough for Sprint 1 but ran into several issues that we looked to improve for Sprint 2. First, since carriers harvesting at a well by themselves cannot write to comms, the HQ would have no idea when carriers are killed and therefore cannot replenish carriers. We lost some maps during the Sprint 1 tournament because carriers chose to gather from far away wells instead of closer ones, leading to lower resource outputs. Second, the update that allowed HQs to spawn 5 units in one turn made our well assignment procedure unviable, unless carriers waited near the HQ until they received their command.

- *Gathering Revamped*. For Sprint 2, we revamped our carrier gathering system completely. Our key insight came when we realized that it was almost always more efficient to gather from wells closer to HQs until they became full. We ended up removing the entire HQ assignment system, detached well assignment decisions from the HQ, and placed it entirely onto the carriers. The carriers would fill known wells saved in comms from closest to furthest alternating between Mn and Ad, moving on to the next closest well if the current one is full or has reached a set limit of carriers. If all of the wells are full, the carriers enter the exploration state. We were also able to simplify our assignment decision process at the start of the game after seeing the importance of Mn over Ad. Instead of creating numerous cases for whether an Ad and/or a Mn well is within vision of the starting HQ, we decided to either assign the first four carriers to Mn if there is a Mn well within vision range or choose to explore for Mn if no Mn wells are in range. Because wells are saved globally and not specific to any HQs, we gradually increase how far away carriers are allowed to be assigned to its first Mn well such that they would only gather at a Mn well near another HQ after they had sufficient time to explore near their own HQ. As more carriers are produced, we gradually increase the limit of carriers on Mn and Ad wells, beginning with roughly a 3 Mn to 1 Ad ratio, as to not overcommit to one resource type. The Mn limit in the early game is increased for smaller maps and when the number of known Mn wells is less than the number of HQs.

- *Exploring.* Ad wells are guaranteed to spawn within 100 units of the HQ, but Mn wells are only guaranteed to spawn within 100 units of the guaranteed Ad well. Since Mn wells are crucial to earlygame strategy, we needed to devise a method that ensured we found the first Mn well as quickly as possible. To this end, we utilized a strategy that we called “Pinwheel Rotation.” We

only utilize this strategy if we do not see a mana well upon initialization. Because if we do, we can simply begin assigning carriers to it. HQs can spawn 4 carriers on the first turn, so we spawn the four carriers as far away from the HQ as possible in each of the four diagonal directions. Each of those carriers then moves 2 steps diagonally away from the HQ, up to the edge of the HQ's vision radius, and then turns left 90 degrees and moves 4 more steps. This essentially makes each carrier sweep out ¼ of the area around the HQ while rotating counterclockwise around it - named the Pinwheel Rotation because it mimics the rotation of a pinwheel. By sweeping out a wide radius around the HQ, we greatly increase our chances of discovering a Mn well and the guaranteed Ad well. In the event that we do not find it, we begin randomly exploring away from the HQ. We also have our carriers randomly explore when the well they're assigned to is full (or artificially capped), allowing us to continue finding more resources to exploit. Once a well not stored in comms is found, the carrier returns to the HQ in order to write the well into the shared array.



An example of the pinwheel rotation strategy in action - there are no visible wells.

<u>Amplifiers:</u>

- *At arm's length.* For combat, amplifiers simply just path away from the nearest enemy military unit that is detected. This simple but powerful move allows our amps to stay alive for a long time, since amps are able to detect threats from far away given its vision radius advantage from attacking units.

- *Military attachment.* The following state is activated whenever there is no attackCommand found and when there is a nearby military unit. The amp simply paths towards the furthest visible allied military unit.

- *Maximum coverage.* With the exception of when an amp is combat, movement will be overridden if there is another allied amp in sight. If this is the case, the amp will path away from the detected allied amp. With this, we can get the best coverage with amps by making sure amp radiuses don't overlap.

Elixir Units:

- *Too expensive*. Elixir units didn't see much play, and because of this, we only implemented very basic micro for these units that never got to see the light of day. Our boosters would use the launchers' follow first mechanism and boost on cooldown, while our destabilizers had a follow first and a combat state similar to the launchers. However, again, because of how the game was, we performed minimal testing on these units.

4.3) Managing Spawns

Because carriers only require Ad to build and launchers only require Mn to build, each HQ will spawn these units whenever they can. Failure to do so will likely lead to a huge tempo disadvantage VS other teams that spawn these units whenever they can. However, we will occasionally decide to make the HQ turn into anchor or amp spawning mode (cannot be in both modes at the same time), meaning that they will hold resources in order to spawn an anchor or amp. Note that because anchors and amps require both Ad and Mn, if one Ad/Mn requirement is met plus additional excess to be able to build a launcher/carrier while the other Ad/Mn requirement is not met, we will build the launcher/carrier for the met requirement as to maintain tempo on the map.

Anchor Spawn Rates:

- *Scale in moderation.* For our anchors, we determined that the best way to spawn them was in set intervals if certain conditions were met. Through watching matches, we found that in general it was better to start the anchor building process for each HQ at round 250 if there were enough robots on the field. This is because regardless of map size, if there is no evident winner at round 250, then it means that the value from capturing an island and getting its healing is worth it. After all, island healing is only as strong as the number of units that use the island to heal, so capturing early islands just for no robot to use it would be a large waste of resources and tempo in the early game. From here, we do a check on rounds 500 and 750 and build anchors on these rounds under the same conditions to slowly gain more island control. From round 1000 and beyond, we then will spawn anchors every 100 rounds as by round 1000, since at this point if the game has gotten to this stage, it likely means that we have the resources and launchers to afford mass anchor spawning.

Amp Spawn Rates:

- *Command and control.* For our amps, we knew that we needed to get them out early for early game exploration and early usage for comms, but we also knew that we didn't want to spawn too many of them. This is because as described in 4.2) Robot Micro, amplifiers are able to get

out of harm's way relatively easily and are able to stay alive for several rounds. If we were to spawn amplifiers at a constant rate like we did with anchors, then there would be a lot of useless amplifiers that likely will have overlapping coverage. Because of this, amplifier spawning for each HQ was based upon how many launchers were spawned from the HQ. Once the threshold was reached, it would then reset the launchers spawned counter and increase the launchers spawned requirement in order to spawn new amplifiers. By using this system for amplifier spawn rates, we were able to spawn early amplifiers while making sure that we didn't spawn more amplifiers than actually needed after the early game.

4.4) Pathfinding

We implemented three different kinds of pathing, in order of greatest to lowest bytecode intensity.

- *Bellman-Ford Pathing.* Bellman-ford pathing is a dynamic-programming approach. Canonically, it takes O(n^3) iterations, however through a strategy called loop unrolling, it is much easier to save bytecode compared to a heap implementation for Dijkstra's. This is because each loop of Bellman-Ford is a simple cost minimization function across neighboring nodes, making the comparison easier to write and generate. There are also certain heuristics that can be used to optimize Bellman-Ford such that you don't have to relax all edges N times: rather, you can do it once or twice by simply relaxing edges in the correct order (from the center and radially outward was our heuristic), which allows you to find paths moving away from your current location reasonably well without taking up all your compute power.

- *Greedy Pathing.* While greedy pathing might just seem worse than Bellman-Ford, in practice it often turned out to be more reliable. Some situations in which greedy pathing is arguably more competitive is when a robot doesn't have the bytecode to spare, or a robot is concerned with 1-2 turn optimality of movement (such as in combat), or a robots vision radius has been lowered (if it's in clouds). The greedy search is a very simple recursive cost minimization. On the first recursion it would only search in the direction of the target location and its two adjacent directions (since this guarantees that you move towards the target). Then, on subsequent recursions, it would enable searching on directions perpendicular to the target direction as well.

- *Tangent Bug.* Bugnav is a well-documented pathing algorithm: it switches between two states: pathing towards the goal in a straight line, and circumnavigating an obstacle if it can't continue moving in a straight line, until its distance to goal is lower than the initial blocking distance of the obstacle. Circumnavigation of obstacles had several edge cases to consider, most of which involved it attempting to circumnavigate around other robots. We used this algorithm whenever other algorithms were unable to return a viable movement.

4.5) Communications

Part of our communications array is pictured below:

Index	6 bits (1-6)	6 bits (7-12)	4 bits (13-16)		
0	HQ Count	AC Even Count	AC Odd Count		COUNT_OFFSET_1
1	Well Command Count	AD Well Count	Mana Well Count		COUNT_OFFSET_2
2	Enemy HQ Count	Team Island Count	Island Report Count		COUNT_OFFSET_3
3	Anchor Command Coun	Anchor Command Coun	Array Wiped/Symmetry Bi	First bit is array wiped,	COUNT_OFFSET_4
4				rest is symmetry	
5					
6	HQ 1 **X**	HQ 1 **Y**		Managed By HQ	ALLY_HQ_OFFSET
7	HQ 2 **X**	HQ 2 **Y**			
8	HQ 3 **X**	HQ 3 **Y**			
9	HQ 4 **X**	HQ 4 **Y**			
10	Enemy HQ 1 **X**	Enemy HQ 1 **Y**	HQ1 **Contested**	Done Through Database	ENEMY_HQ_OFFSET
11	Enemy HQ 2 **X**	Enemy HQ 2 **Y**	HQ2 **Contested**		
12	Enemy HQ 3 **X**	Enemy HQ 3 **Y**	HQ3 **Contested**		
13	Enemy HQ 4 **X**	Enemy HQ 4 **Y**	HQ4 **Contested**		
14	HQ Command **X**	HQ Command **Y**	HQ Command **Type**	Managed By HQ	HQ_COMM_OFFSET
15	HQ Command **X**	HQ Command **Y**	HQ Command **Type**		

Our communications system is extremely modular, and is based off of four types of communications that can be added in to it: *HQ*, *Permanent, Command,* and *Report* communications. Each type of communication had its on control structure, and each time a communication was added to the array its count was adjusted accordingly. If we wanted to wipe a set of communications from the list, we simply reset the count of that communication to 0 (not spending the extra bytecode on cleaning out the rest of the array).

Each index in the array was split into two 6-bit blocks and one 4-bit block for ease of storing MapLocations.

- *HQ Comms.* HQ comms included all communications that HQs were responsible for setting and maintaining the lifetime status of. That includes 4 ally HQ locations, 4 global HQ commands, and 2 Well Construction commands.

- *Permanent Comms.* Permanent comms included all information that would be stored in the communications array permanently. This was in order to have all units be able to read the information on communications. These communications included locations of Enemy HQs, Adamantium Wells, Mana Wells, and symmetry bits.

- *Command Comms.* Command comms were temporary commands that were split into even rounds and odd rounds. On even rounds, robots would read from the odd section of the array allocation and write into the even section of the array, and the opposite was true on odd rounds. This was because if a robot published a command and it was just wiped out of the array at the beginning of the next round by HQs, then there was a chance that half of the robots might not see it (e.g. if you were the last in terms of turn order and published a command, then it was wiped, nobody else would have the chance to see your comms). Thus, by writing to the evens part of the array on an even round, you could read all commands from the last round (the odd

section of the array) without having to be afraid of overwriting a command that somebody hadn't seen yet. The communications that we used this control strategy for included Attack Commands (where launchers within radius (mapdiagonal)/4 could respond to it, and would path towards the highest priority Attack Command near them) and Anchor Commands (which would be sent to direct carriers with anchors towards uncontested islands so that we could capture them.

- *Report Comms.* These communications were used where a robot would report a map location and the state of that map location, and the HQ at the beginning of the next round would read the report and then modify a permanent set of information based off of it. We mostly used this for carriers and launchers reporting that an island had been captured or lost.

Local database. Another implementation detail that is worth mentioning is an interface that we made for robots to store information locally. Since communication was not global this year, robots had to be able to store information and report them when they had a chance. In order to do this, we had robots store a list of wells and enemy HQs in local arrays, then "upload" to the global array when they were in range of HQ or amplifier communications. They would also "download" any new permanent information when they saw the global array had been updated (by checking the communication counts). This made for a system that enabled us to do basic symmetry checking of the map (for if it was rotational, vertical, or horizontal) and well assignments.

Reflection and rotation. Symmetry checking especially made significant use of this global/local information storage strategy since by using the rotationally, vertically, and horizontally reflected map locations of wells and HQs, you could rule out certain symmetries. Once symmetry was known, you could reflect known well locations and ally HQ locations to find the HQs and wells on your opponents' side of the map, and set those as priority exploration locations for your launchers.

4.6) Bytecode Hacking

Battlecode robots are constrained by the amount of compiled code that they can run, represented by lines of bytecode. For those unfamiliar with bytecode, Java bytecode's relationship to Java is as Assembly is to C. Due to the nature of this constraint some actions are especially punishing: long loops (due to increments and conditional checking), Java STL usage, and deep recursive functions.

However, due to how useful some of these features are, the Battlecode community has come up with some ingenious strategies to resolve these issues.

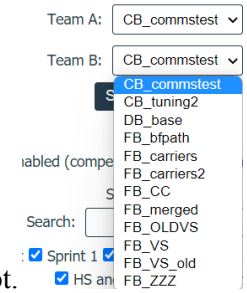
Last year, we focused a lot on bytecode hacking, but this year we found that focusing on basic strategy helped us more. However, there were a few areas that we felt were used so often in our code that they deserved to be optimized:

- *Bellman-Ford search of tiles within vision.* The loop-unrolling strategy writes out every minimization step in the series of loops, enabling us to skip the iteration steps of initializing, conditional checking, and increments, which chew through bytecode in cases such as long loops or recursion.

- *Custom Hashset*. Since HashSets were useful for determining whether an ID or MapLocation had been seen/processed before, we definitely found the need for a HashSet that didn't chew through as much bytecode as the one in Java.util (which has a lot of unnecessary stuff like exception checks and binary trees for managing collisions). Instead, we implemented our own linear-probing hashset without any extraneous parts, significantly decreasing the costs of the add() and contains() functions.

4.7) Version Control

We went through a 5 archetypal bots throughout the season, each with a set of 6-10 packages slowly developing new features until we thought we had finished the objectives that we wanted to for that bot.



bot.

- *DB (Dumb Bot).* This simple archetype was completed within the first 2-3 days of the tournament releasing. It consisted of exploration every turn in random directions, running at the nearest interesting thing in vision range, and acting on cooldown.

- *BB (Better Bot).* This was the natural continuation of development, where we implemented random targeted exploration rather than random movement, more optimal micro, better pathing, and carrier assignment to wells. This is the archetype we submitted for the Sprint 1 Tournament.

- *CB (Comms Bot).* Here is when our comms system came into full effect, with us maxing out our Communications capacity such that we could determine map symmetry, report well locations back to HQs, and send commands through the array. This is the archetype we submitted for the Sprint 2 Tournament.

- *EB (Elixir Bot).* This archetype, ironically, did not end up using any elixir. We tested converting wells, but found that without the Adamantium upgrade exploit that we were losing too many maps due to losing map pressure when farming elixir since the units built using it were too expensive for most of the game. Instead, this archetype implemented better island reporting and launchers falling back to islands, which enabled us to leverage the active use of island healing to our advantage.

- *FB (Final Bot).* At this point, most of our strategies were pretty set in stone, so we decided to just do extensive spawn ratio tuning and bugfixes. This archetype was the bot that we submitted to both the Qualifying and Final tournament.

5) Final Thoughts

5.1) Areas for Improvement

- *Bytecode Hacking.* In early season, bytecode wasn't much of an issue since most of our strategies were simple, or complicated implementations were usually the only bytecode-intensive part for their specific bot. However, as our bot behavior became more complex over time, our lack of bytecode optimization started to hurt us more and more as our bots started to run out of compute power later on into the season. Some of the higher-level strategies we conceptualized but never implemented would definitely have required a significant degree of bytecode hacking (e.g. parallelizing through integer operations, using more bitshifts, code generation for loop unrolling, using 1-cost stringbuffer ops to store and modify info), so if we start looking into advanced strategies earlier in future competitions it will be important to optimize bytecode usage to a higher degree than we already have.

- *Advanced Communications.* Right now, our communications system is well-implemented, but it lacks an order of power that can be gained from changing the type of information that is reported every turn. This problem is twofold: first, the way the current communications system is designed, there is empty space in the array almost every turn that could likely be used to communicate something more important going on on the map at a given moment. Second, some information can be broadcast on even turns, or odd turns, or every four turns, such that bots can gain a slower, but more in-depth understanding about the state of the map.

- *Be more adventurous.* Last year, our team suffered from conceptualizing difficult strategies early on without actively figuring out implementation details, leading to us having some well-thought-out strategies but with glaring weaknesses that were exposed during competitions. This year, in order to qualify, we focused almost entirely on generality and robustness, which got us towards the finals. However, against the top teams, what we've realized that at the highest elo, generality and robustness starts to take a back seat to legitimate higher-level strategy. We didn't have enough time to pivot our designs out towards having more specific strategies for specific map states. In the future, with our increased experience of competing against better teams, we hope to develop into these more specific strategies earlier on into the season.

5.2) Advice for New Teams

- *Be prepared.* Read the specs and previous years' postmortems: you never know what might be changing from last year, so it's useful to amass all the knowledge that you can. Study strategies that remain relevant throughout the years: previous pathing, communications, and development processes are especially important. If you are extremely unfamiliar with robotics theory, bit manipulation, or search algorithms, pick up some material on those as well . If you have new members, make sure to spend the time to give them a thorough onboarding, and make sure they understand the huge time commitment that Battlecode is.

- *You may have lost the battle, but you have not lost the war*. It's a long season which means that mistakes early won't amount to anything large. Being consistent in the work that you put in is more important.

- *Make it work, make it right, make it fast.* While this mantra is often an overused cliché in software development, it holds significant merit in a high-pressure, low-resource environment like Battlecode. More often than not, it is important to get the fundamentals of your core strategies down first. Don't try to invent Battlecode Stockfish without having the basic dumb "run straight at the nearest thing" bot done. This isn't to say that grand ideas and designs aren't possible: rather that having something working first is more important. Due to the distributed, high-dimensional, bytecode-intensive nature of battlecode, optimality is often not so important as robustness, at least when you're just getting started.

- *Push every advantage.* This might seem contrary to the above, it doesn't have to be. While you're getting something simple working, it's a good idea to plan ahead for ideas that you'd like to try out, and list them in terms of priority. If you're not sure about a strategy, there's no harm in just trying to implement it in a copy of your current best bot and testing if it makes the bot better. Create a copious number of new packages and new versions, each implementing or testing a new feature that you think will help you win. You'll need every edge to win come the tournament.

- *Modular code.* This is more for teams of more than two members, where not everybody understands all the code in the project. Making the code readable and also flexible greatly improves the efficiency of how fast a team is able to adapt to changes. By splitting an overall strategy up into multiple parts (such as communications, pathing algos, bot-specific micro, and interfaces between each of these) it becomes much easier to swap out easier-to-implement strategies written earlier with advanced strategies developed later in a cycle of iterative development. It also helps in others' understanding of and ability to modify code that you have written. You don't want to be heading into week 3 with a mountain of technical debt due to an unparseable codebase.

- *Copying strategies.* Understanding another top team's strategy and applying it yourself is never a bad idea in Battlecode. Some of these competitors have been doing Battlecode for 5-6 years, so you'll have to spend time just to catch up, then more to imitate, implement, and understand some of their strategies. But remember: you have to actively attempt to improve upon the "inspired" algorithm, or else it'll never be as good or better than the team that you took it from.

- *Full-set scrimmaging.* Make sure to get a comprehensive overview of how well you're doing against other teams, and use the ladder to examine their strategies. Just three matches might not be enough to highlight the strength and expose the weaknesses of one's robot. Luckily, this season **teh devs** decided to enable 10-match scrimmages online, which greatly helped in this regard. We paired online full-set scrimmages against other teams on the ladder with all-map scrimmages against old versions of our bots on our local clients to ensure that we were making progress consistently throughout the season and to track the state of the meta.

- *Code > Elo.* Especially during early season, we found that by focusing on the quality, modularity, and adaptability of our code rather than on how to win matches on the existing maps brought us greater success during tournaments. Generalizing solutions in your code rather than coding for specific cases that appear on the ladder often helped us find wins in tournaments where opponents might have hard-coded heuristics that beat us on the ladder. Focus on making the bot better, and the elo rise will come naturally.

- *Find the fun.* It's way easier to focus on the competition if you can find the enjoyment in it, whether that's climbing elo, developing a specific part of the bot that you're responsible for, interacting with the community, or the competitive atmosphere of the tournaments.

- *Strong mental.* Finally, we'd like to offer a few words of encouragement. Battlecode wasn't always a good time. Sometimes we despaired as our bot started losing to an older version across a set of 30 maps. Sometimes we went close to 0-12 in ranked scrimmaging. But being willing to go through lost game after lost game, examining your bot's weaknesses so that you might improve them, is par for the course in this competition. With everybody on our team at less than 2 years of experience, at times it was difficult to see the point in continuing to compete against all the absurdly good teams that participate in Battlecode annually, but it's important to remember that even the best teams weren't always ranked highly. If you truly believe in your team's ability to pull out the win, and can break down your problems into manageable objectives, then odds that at first glance seem insurmountable can eventually become just another hurdle to crossed.

5.2) Acknowledgements

Again, thanks so much to **teh devs** for all the work they put into organizing the game, ladder, and tournaments, and especially **serenali** for helping us with the trip. We know how much time in advance preparation begins for Battlecode and greatly appreciate them for spending their time to give us such a wonderful opportunity. Also huge shout out to the teams **Super Cow Powers** and **XSquare** for actively exploring well beyond the meta throughout the season and often openly sharing strategies generated from their wealth of experience. They greatly helped us and other competitors improve our bots, and continue to contribute to the Battlecode community even as grad students.

5.3) Funny Memes

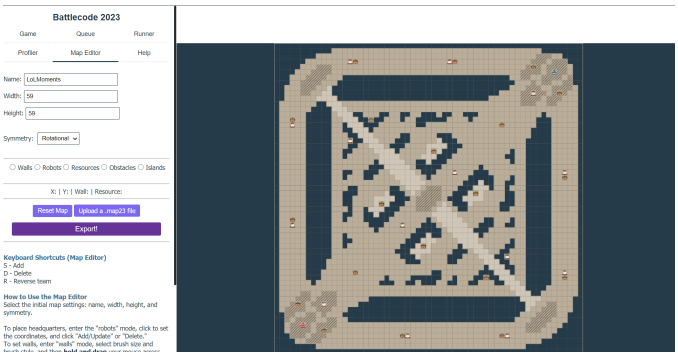
We wanted to spotlight some exceptional memes made by the Battlecode community that we think exemplified the spirit of the competition this year.

Credit: **A214**

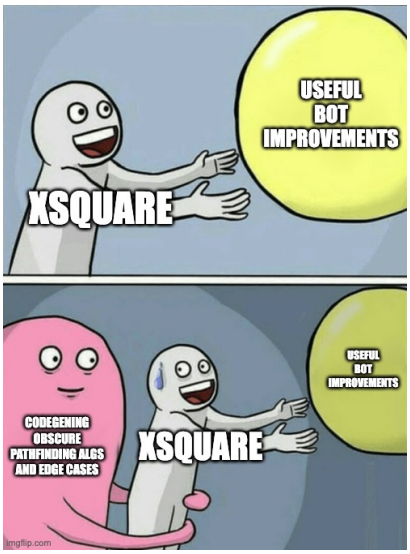
Credit: **Your Anchor is On Another Island**

"Never Gonna Give You Up" by Rick Astley, but it's in BattleCode

Credit: **odo**



Credit : **Pudding1015**



Credit: **4 Musketeers**