4 Musketeers - Battlecode 2023 Strategy Guide

Winston Cheung, Maxwell Jones, David Lyons, Bharath Sreenivas

February 2023

Contents

1 Introduction

1.1 Battlecode Introduction

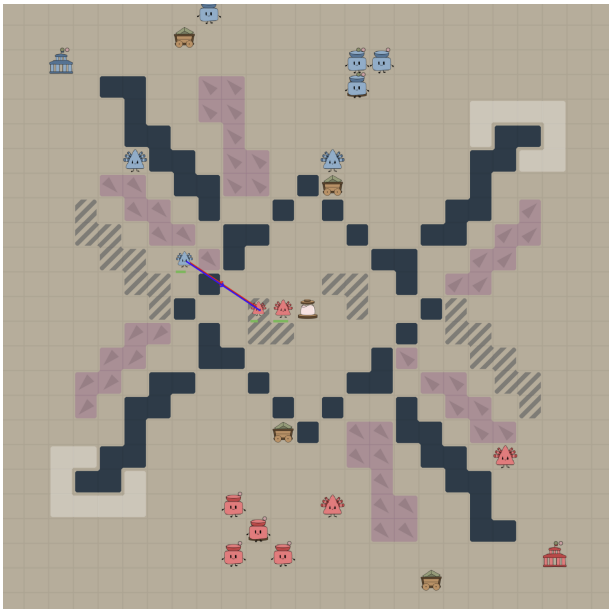
Battlecode is an AI competition run every year throughout the month of January. At the beginning of the month, MIT releases a new 2 player real-time strategy game, and teams code up a bot that plays said game. There are two sprint tournaments, a qualifier where the top 16 are chosen, and a final where the top 16 battle it out for cash prizes.

1.2 Team Introduction

All of us are computer science students in our Senior year at Carnegie Mellon University. Our first year, the original team when created didn't have Winston, so we named ourselves the 3 Musketeers. When Winston joined about a week into the competition, our name suffered from an off-by-one error. Continuing that trend the next year, we became the 5 Musketeers. This year we finally counted correctly and dubbed ourselves the 4 Musketeers. This was our third year doing Battlecode after qualifying for finals two years in a row, so we came in this time with heavy expectations. Just like last year, this year's competition coincided with school and posed a big challenge, but this year was even crazier, with ranked scrimmage requests and a mystery tournament. Nonetheless, we managed to three-peat and make finals once again, and this strategy guide will explain the road to get there. Our code is linked here, with our final player here. We link the relevant commits throughout, but it also may be useful to see the final player, as some strategies and scripts mentioned were slightly changed and improved before our last submission.

1.3 Game Overview

Battlecode is now a cinematic universe with a timeline and everything. Battlecode 2022 was about a post-apocalyptic world using mutation and alchemy, and Battlecode 2023 is about how such meddling destroyed the fabric of reality, forcing people to open portals to new universes and use their beautiful sky islands and time-bending tempests to save their universe. However, apparently other universes were also full of idiots messing with alchemy and are trying to do the exact same thing, so you have to fight for the right to claim this new territory and save your universe.



Each team gets 1-4 headquarters, and each map has a bunch of sky islands. There are three main resources: *adamantium*, *mana*, and *elixir*. There are adamantium and mana wells scattered throughout the map, but in order to create an elixir well, you need to put adamantium into a mana well, or vice versa. All three resources are useful for building robots and building anchors to seize control of the sky islands. There were two ways to win:

- Capture 75% of the sky islands on the map to win immediately, or at least more than the other team by the end of the game.
- After 2000 rounds, a tiebreaker goes to the team with elxir, then mana, and finally adamantium.

The rectangular map could be as small as 20x20 or as large as 60x60. Any square could have one of three obstacles:

- **Walls:** Unlike previous years, where rubble controlled how long it took to pass through squares, this year had walls that you can't get through at all. This made many maps a tough pathfinding challenge.
- **Currents:** If you're on a square, and that square has a current pointing to another square, you will move in that direction at the end of the turn.
- **Clouds:** The fluffiness of clouds obscures vision even though these are robots. If you're in a cloud, your vision radius is decreased. If you're not in a cloud, you can only see things inside a cloud that are within that same decreased vision radius.

There were five different units this year:

- **Launchers:** These are the main attacking units. They cost some mana and deal a decent amount of damage. Very similar to the soldiers from last year.

- **Carriers:** These are the main mining units. They cost some adamantium, can extract resources from wells, and can bring those resources back to the headquarters. They also carry anchors to sky islands and can throw their resources to deal damage.
- **Amplifiers:** Communication was different this year, and you could only communicate if you are close to your headquarters, close to an island you own, or close to an amplifier. Thus, you can use both mana and adamantium to create amplifiers that make it easier for your robots to communicate.
- **Destabilizers and Boosters:** Destabilizers cost a lot of elixir, but if you can acquire one, they can be invaluable to winning skirmishes. They deal AOE damage and slow down time, making everything in their blast range have longer cooldowns. Boosters were the exact opposite, speeding up time and decreasing cooldowns.

2 Strategy Development

2.1 Coordination

2.1.1 The Sector System

Last year, we used a clustering system where we kept track of average enemy locations and made all our choices based on how close we were to those clusters. This year, we decided to finally move on from a cluster system to a sector system, in which you divide the map into several small sectors and keep track of information for each sector. This is important because then, rather than saying an enemy HQ is some map location, which takes up 12 bits, you can say the enemy HQ is in some sector, which takes up 7. The idea is to send units to that general location, and then once they're there, they can see the sector in their vision radius and make any remaining decisions. With respect to sectors, we wanted to keep track of two kinds of information. First, for each sector, we want to keep track of some basic information about it. Second, globally, we want to keep a list of sectors that might be useful for some purpose, like a list of sectors with lots of enemies or a list of sectors that need to be explored.

2.1.2 Movement

Our movement started off with some classic unrolled greedy BFS popularized by **XSquare**, a past finalist, sprint winner, and overall genius of Battlecode. It works by a using a relaxation of general BFS that only considers paths which go strictly away from the center. The key to its efficiency is that this BFS uses a large amount of constants and if statements to perform computation in as little bytecode as possible, allowing for a larger area to searched through. This initial BFS version didn't take into account currents, but later in the tournament the bot was updated with a BFS that respected currents. We also had multiple versions of this BFS function that considered different areas to search, and we would use the largest possible BFS area on a given turn that seemed possible to complete with the amount of bytecode the robot had left.

In addition to this BFS, we used Bug Navigation (Pathfinding.java file) when we hit impassible squares, as that is the next best way to reach your goal when impeded. This naive bug navigation was later updated into a much more complex version. At first, when we haven't seen the entirety of a wall structure, we would always turn in the same direction in order to see the full wall before making a decision about how best to cross it in the future. From here, we assume the obstacles are "nicely formed" and generally have only two "thorns," called points of interest in code. Nicely formed obstacles are things like lines or L shapes, which have two clearly defined endpoints. Squares also count, which are uniform enough such that an assumption of two POIs is sufficiently accurate. We make the assumption that, in order to path around an obstacle, we have to path to at least one of these endpoints first, from which we can then continue on to the actual target. We represent the obstacle (a contiguous set of walls) as a graph and try to find its diameter. The formal double BFS algorithm only works for tree graphs, but since we're assuming "nice" obstacles (boy, did the devs love messing with this assumption), this actually gives us a good enough approximation for Battlecode. To calculate the correct direction, we save the distances of each point on the obstacle to the two POIs and choose to follow the direction which has the smaller combined total of the BFS distance to the POI and the POI dist to the target. Performing all of the BFS is often too expensive to do in one turn, so it's distributed over several instead. An obstacle of around 20 walls generally takes around 15000 bytecode to finish, so if we use our spare bytecode at the end of each turn to work on the computation, it can finish in about 3 turns. Once we've finished, we can then use our knowledge of the points of interest to always find the correct turning direction whenever we need to go around it. See here for the actual code.

2.2 Communication

2.2.1 The Database

Communication was very difficult this year. Just like last year, there's a single global array rather than a flag for each unit, but unlike last year, you can't write to the global array whenever you want. You can read from it, but you can only write to it when in range. This meant that each robot needed to keep track of much more information. Previously, whenever a robot sees something, it can forget about it next turn because it's already reported it. But now, a robot could potentially see 100 rounds worth of information across dozens of sectors without ever being able to report it. In previous years, we could get away with a robot storing a few variables and sets like a home location and a list of nearby enemies, but now, we needed a full-blown database.

We created a SectorInfo class that stores information about a sector. A well, an island, an enemy...really anything interesting. The database is a list of SectorInfos. Anytime a robot sees something in a sector, it records an entry in its database for that sector. And we keep track of the exact map location and who owns the island or what kind of well it is also. Unlike the global array, which has limited bits, individual units have plenty of space. We can store exact information and filter it later so that we don't accidentally record the same thing twice. For example, say that there's a neutral island that gets taken over by the enemy. By storing detailed information in our database, rather than thinking there are two islands, one

neutral and one enemy, we'd know that the neutral island had been taken over and replace the entry. Then, when a robot is able to finally write to the global array, it uses its database to update the information. This approach was effective, but unfortunately, it was a lot of bytecode to initialize the entire database. Thus, we opted for a lazy initialization, in which the SectorInfos are NULL unless we have something to record. However, to avoid having NULL checks literally everywhere in our code, we create a separate SectorDatabase class with .at() wrapper functions.

2.2.2 Reporting

There was one downside of switching to the sector system, which is that describing the sectors themselves takes a lot of bits, maybe 12 per sector. Each entry in the global array is 16 bits, which is enough space for a sector. Are there enough bits overall to store all the sectors? Yes. Are there enough entries in the array to store one sector per entry? No. If your sector takes up 12 bits, then you better use those remaining 4 bits to store part of the next sector. In previous years we were able to manually write Comms code, but this kind of bit packing required an automated approach. Otherwise, changing something in the middle would create days worth of side effects to fix. Thus, we adapted **smite's** autogenerated Comms code from 2022 to maintain our sector system. This led to another problem, though: write amplification. Let's say that you want to write something to sector 1, and it's on indices 3 and 4. Then you want to write to sector 2, and it's on indices 4 and 5. You've now written to index 4 twice. And what if an index contains parts of three different things? A write costs 100 bytecode, and the limit is 10000 per turn, so that's significant. Apply that to, say, 10 sectors? And suddenly you're wasting half your turn on reporting. To fix this, we applied the database systems concept of a buffer pool, where you have an intermediate set of pages that you write to and then flush to disk, rather than writing everything to disk since disk operations are expensive. Since a read is only a few bytecodes, we first read the entire global array to a local copy. Then, we write everything to our local copy, setting dirty flags. Finally, for each dirty index in the array, we flush it to the global array. Now, we could effectively communicate.

Figure 1: A carrier gets attacked by an enemy, so he goes back to his HQ to report.

2.3 The Opening Act

2.3.1 Early Strategies

As always, it pays to start small. Make carriers and launchers, and have them move basically randomly. If a carrier sees a well, mine from it. If the carrier is full of resources, go home. If we see an island, start building an anchor and take it to the island. Of course, since this is our third year, we can start a little bit more complicated than that. Since launchers are very similar to last year's soldiers, we ported the soldier code over to the launcher code, which has very well-tuned micromanagement. On a given turn, the launcher takes note of the enemies it sees and their locations and health, the allies it sees and their locations and health, and its own health. If there are more enemies than allies, we assume this is a fight we can't win and back up, attacking first if we can. We also do this if our health is critically low. If we're ready for a fight and we're in action range, we attack and then back up so that it's harder for them to attack us. If we're ready for a fight and we're out of action range, we move towards the enemy and attack. There were other basic changes we made as well beyond the basic strategy. For starters, if there's a well within vision radius of the headquarters, we build carriers towards the well so that they can gather sooner. Further, we made carriers better at surviving. Carriers are faster than launchers, but not if they're carrying resources. If a launcher is approaching, it might be game over for the carrier. If the carrier can make it home to deposit its resources before dying, it does so. Otherwise, it throw its resources at the enemy and then runs away, ensuring its survival so it can mine more later. Finally, we knew from previous years to take advantage of symmetry. Based on your HQ locations, you can guess the enemy HQ locations and move towards them to see if you can find an HQ. Launchers, in particular, are very useful here because they can crowd the HQ and kill anything that spawns, as well as block most of the spawn locations.

Figure 2: Soldiers crowd around a base and instantly kill the next spawn, and carriers throw their cargo at enemies to survive.

2.3.2 Micro (Unit Management)

After Comms were finished, we could now have a much more nuanced strategy. In particular, launchers and carriers no longer have to explore randomly. In our global array, we have a list of combat sectors. When a launcher is deciding where to go, it can go to the nearest combat sector and start battling, keeping the enemy from being able to expand their territory. If they get to the combat sector, and it's empty, they can go to the next one. Notably, since enemies move, we reset that information every 100 (this number changed later) rounds, considering it stale. The combat sectors are useful to carriers, too. If the carriers are going home to deposit materials or report, and their home is in a combat sector, then they know they'll probably die on the way there. If we have multiple HQs, and there's another one nearby that's not a combat sector, we can make that one our new home. We also have a list of mining sectors, so the carriers know where to go to find the nearest well. They can also loop through all the sectors and check to find which ones have adamantium or mana wells, meaning that we can design carriers for a specific purpose. Finally, we have explore sectors, which are the symmetry locations and any combat sectors that became stale so that units know to check out the areas and see if there's anything there.

In addition, we gave sectors a control status and claim status. We'd prefer to send our carriers to wells without enemies than to wells with a lot of enemies, and the control status makes that distinction. To facilitate this, if a carrier encounters an enemy and needs to run away, we don't just run away: we report. If a carrier encounters danger on the way to the well and back, chances are the next carrier will have the same problem. Our priority is to get home and relay this information as fast as possible. The claim status tells us that a sector is being explored and that we don't need to send anyone else there. Even without a claim status, though, we make decisions based on what our other units are doing. If a carrier reaches a well and it's crowded, they find another well, and if a launcher finds an enemy HQ and it's crowded, they explore elsewhere. To avoid an oscillation problem where launchers see that an HQ is crowded, step away, see that it's no longer crowded, and step back, we added timeouts so that we would avoid the definition of insanity and not try the same thing twice expecting different results.

Besides Comms, there are some interesting micro decisions with respect to clouds and currents. For currents, if they're away from the direction we want to go, obviously we want

to avoid them, but if they're in the direction we want to go, it's not necessarily always good to use them. If we're really close to our destination, we might as well just finish rather than use a current just in case it leads us into a ditch, as many maps did. For clouds, if we're near a cloud, and there's no one else to attack, we might as well randomly attack the cloud, as there might be an enemy hiding in there.

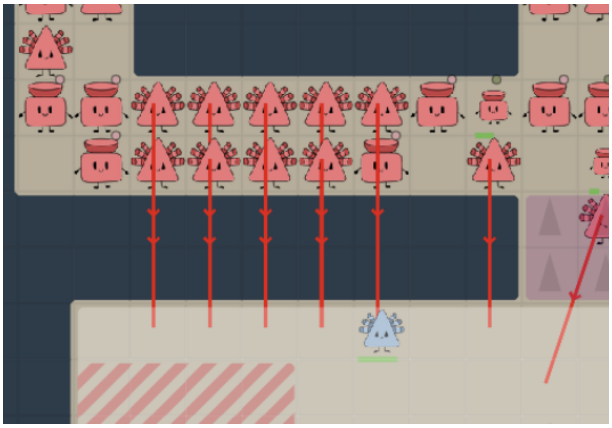


Figure 3: All of our launchers are randomly attacking clouds, and one of them manages to hit an enemy.

2.3.3 Macro (Resource Management)

In the first few rounds, we want to just build 4 carriers and 4 launchers, since we don't know anything. By default, two of the carriers hunt adamantium, and two of the carriers hunt mana. However, map size can change this. On small maps, it's advantageous to rush the enemy as quickly as possible, so all four of our carriers will go to the nearest mana well so that we can build tons of launchers. Even on bigger maps, if we see a mana well right by our HQ, we'll prefer 3-1 in favor of mana. Rushing the enemy and owning the map can be pivotal. Overall, we want about a 2-1 ratio of mana to adamantium so that our launchers can take over the map, so when we create carriers, we decide which kind of carrier to make them based on whether we're overshooting or undershooting that golden ratio. When we build the launchers, if the map is small, we want to build them towards the center of the map so that we can get map control as quickly as possible.

2.4 Winston Wonderland

Since Battlecode started a week later this year, we had to go right back to school after sprint 1. David had a compiler to implement, Bharath had dance practice, and Maxwell had conference deadlines and PhD interviews. What followed was Winston Wonderland, in which Winston completely overhauled our code for sprint 2 and mostly solo'd the competition (with a bit of help from the team), yet still performed extremely well. We got top 4 not only in the second sprint tournament, but also in the brand new mystery tournament in which the maps are extremely difficult and test the robustness of your code.

2.4.1 Balance Changes

Just like last year, the devs attempted to make major balancing changes between sprint 1 and sprint 2, but they didn't end up changing anything. There was only one change that required us to alter our strategy, and that was HQ damage. The devs were noticing that literally every team was sending launchers to charge the enemy HQ and huddle around it so that they could kill anything that spawned, so the devs made it so that anything in an HQ action radius would take passive damage. The change? Do the exact same thing, but stop when you're a step away from the action radius. The strategy was exactly the same, just with a slight tweak. We also adjusted the pathfinding a bit to avoid the HQ when possible. There was another balance change, which was that the HQ could build multiple units on the first round, so we could build all 8 of our first rounds units in less turns. There was a catch, though, at least for our strategy. In order to tell a carrier whether it's an adamantium or mana carrier, we set a flag in the global array. Well, if we build two carriers, we can't set the flag in two different ways, so how do they know which is which? We ended up building multiple launchers per round but only building one carrier per round.

2.4.2 Adding Nuance

For the most part, we've assumed that if you get to a sector, you can find anything in that sector pretty easily. That assumption starts to get rocky when you consider clouds. We added some new logic so that you start by going to the center of the sector and then explore the four corners. To avoid using a lot of bytecode, we precompute sector centers when a robot is initialized so that we don't have to do any math after turn 1. Other bytecode optimizations include hardcoding the buffer pool (i.e. rather than for i in range 0 to 64, have 64 sequential statements).

We added some more nuance to launchers blocking the enemy base. Let's say that the friendly and enemy HQs are on opposite ends of the map, but away from the edge of the map by 5 squares or so. We'd never approach the HQ from the back because there's not much open space there, but if we only approach it from the front, then the back is unguarded, and the HQ can spawn enemies. We made a change so that launchers crowding an enemy base will rotate around it, attacking any enemies they find along the way. Not only does this better guard the base, but it creates openings in the direction we originally came from, allowing more launchers to join our circle and creating a stronger defense. We also made it so that they invalidate symmetry locations, allowing them to find enemy HQs faster. If we have two HQs, then there are two possible HQ locations if the map has vertical symmetry, two more if it has horizontal symmetry, and two more if it has rotational symmetry, give or take. Well, if we explore the first vertical, and it's not there, then the second vertical isn't either! This makes us better hunters.

Our macro got more nuanced as well. Before, we were just maintaining a constant ratio of adamantium to mana. Now we make a more complicated calculation. We wait until we've taken a first guess at where the enemy HQ is. If it's close, then we need to build a lot of launchers, or else we'll die. If it's not close, then we have time to build adamantium carriers and build up our supply before we need to start getting mana for launchers. In this case, we can prefer adamantium 3-1, which never happened before. In addition, we can make

decisions based on which wells we've seen. If we see a mana well but no adamantium well, and it's a small map, then the adamantium well is probably hard to get to, maybe tucked away in a corner. In this case, we build exclusively mana carriers. If it's a bigger map, then there are probably still adamantium wells we can find, so we build mostly mana carriers but still build adamantium carriers. Vice versa if we've only seen an adamantium well.

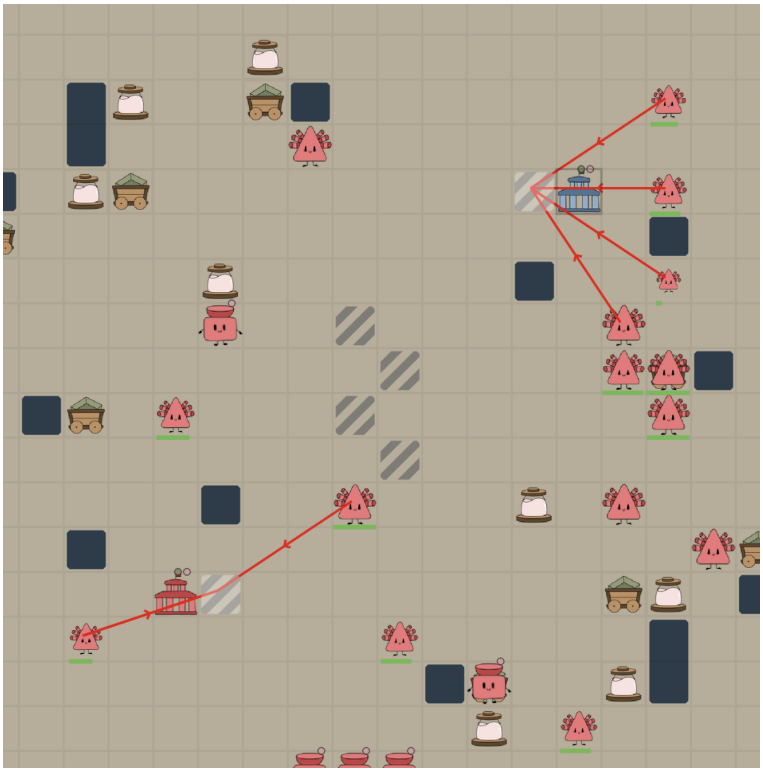


Figure 4: Launchers move behind the enemy base to make room in the front for more.

2.4.3 Testing Versions Efficiently

In the past, if we wanted to test two different versions of our player, we would have one of the teammates run that version against all previous versions locally. When there are > 30 maps, this can take hours and renders that person's computer useless due to the processing power. Around this point, we finally got around to adapting a feature from **Producing Perfection**, another finals team, who wrote a script to run players against each other using Github Actions. After the script ran, we would get a nice match summary detailing how our bot did against an older version we specified.

2.5 The Cheung Dynasty

Still, Winston was running our codebase, but now, he wasn't just surviving. Mostly singlehandedly (with the exception of Section 2.4.3 and some ideas of Section 2.5.5), Winston was able do amazingly against the entire competition and maintain the top seed going into qualifiers and finals. We lost in the winner's bracket in qualifiers due to a map exploiting an extremely rare bug in our code, but we won both loser bracket rounds 5-0 and moved on to the final tournament.

2.5.1 Tragedy Struck

We entered the qualifying tournament as the top seed, but unfortunately lost to the sixteenth seed Baby Ducks in the winners side. There were two main problems that caused this catastrophe. The first was a map with 3 HQs all bunched up in the corner. On turn 1, the top left corner is supposed to:

1. Create launchers and carriers, and find good locations to put them on.
2. Record the locations of other friendly HQs.

As previously mentioned, we use flags to tell carriers what kind of resource they're hunting, so we want them to be closer to our HQ than any other. On this map, however, the combination of the adjacent HQs and the map edges around us meant there were no locations closer to the top left HQ than the other two, so that HQ cycled through every single position trying to find a good spot, using up a lot of bytecode. As a result, it never got to the second step and didn't record the other two HQs. Because other friendly HQs were set to null, any functions that use those HQs would throw null pointer exceptions, making the top left HQ completely useless. As a result, all resources deposited into that HQ was wasted, costing the game. After qualifiers, we made extra sure this bytecode issue wouldn't happen again by adding more bytecode checks. The second issue was pathfinding on a particularly large map with lots of walls. As more bots got stuck, new bots would continually oscillate rotation directions (refer to Section 2.1.2 for more details). Our bots got stuck in one section more than our opponent, and we ended up wasting a bunch of resources. With this being said, guessing rotation does way better on most maps, so it was worth the cost (especially against enemy teams who were also increasing movement speed on most maps by guessing better rotation directions).

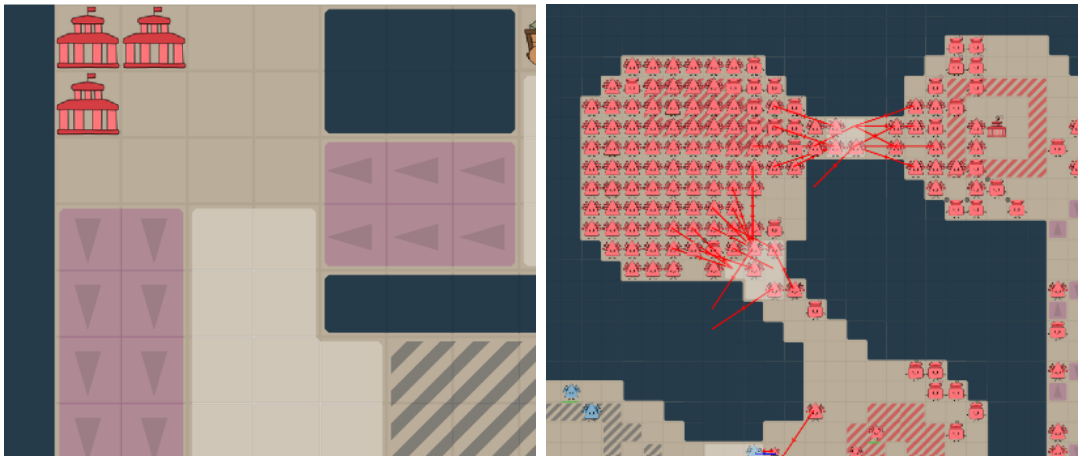


Figure 5: The two maps where we failed, the one where the HQs were bunched up, and the one where our launchers got stuck pathfinding.

2.5.2 Healing

As always, the devs introduced a new balance change after the sprint tournament. The previous balance changed introduced the capability for owned islands to heal units, but this

balance change buffed it enough to be viable. A rudimentary strategy would be to keep track of the nearest island you own and go there if you're low on health, but there's a catch. Let's say that in a skirmish, you have four launchers, and they have two. Easy win. But if three of your launchers have low health? Those three will go to heal and leave the fourth, who was ready to battle, all alone. That launcher will then die and become useless even though it was your strongest one. So, unless your health is very high, if you have friendly robots with low health, you should go to heal, too. Stick together and come back to fight later. But when you get to the island, where should you situate yourself? Islands take up multiple squares, and you have to choose which one to use to heal. It seems like an irrelevant decision but can make a huge difference if enemies come to the island. We implemented a scoring system that rates possible locations, and you go to the spot with the best score. A spot will get more points if it's in a cloud (so that you are unlikely to be attacked while healing), near other friendly launchers (so that you can work together to defend yourself if attacked), or near you (so that you can start healing as soon as possible).

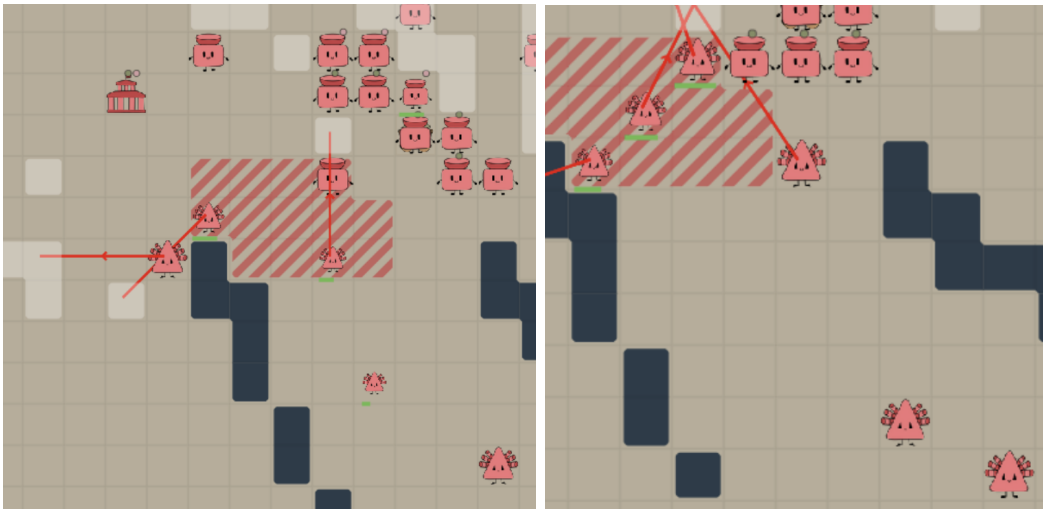


Figure 6: Launchers going to heal and then those same launchers going back into battle.

2.5.3 Elixir

Last year, the balance changes after sprint 2 completely changed the meta, with every team having a soldier rush before the balance changes and every team having a sage spam after the balance changes because gold was finally a worthwhile resource. This year, that didn't happen. After the balance changes, elixir was equally as useless. Very few teams used elixir at all. We ended up implementing elixir capability because it was useful for a few select kinds of maps, but we didn't use it as part of our main strategy, nor did any other team. In particular, we used elixir if the map is sufficiently big and it's sufficiently late into the game because that means that there will be a lot of units, so a destabilizer blast attack will be very effective. We only convert mana wells because sacrificing the mana to convert an adamantium well meant a temporary halt in launcher production, opening yourself up to a rush. When picking which mana well to convert to elixir, we consider all of our mana sectors and adamantium sectors and find the mana sector with the smallest total and average

distance from adamantium sectors. This is because we need adamantium to turn it into elixir, and if the adamantium is close by, the conversion is quick.

We also tried to add amplifiers at this point, but we couldn't quite get them to work well. We noticed that **The Gradiloquent Grinders** (another team in finals) were using amplifiers to communicate the exact locations of enemies as opposed to just sectors in order to facilitate blind attacks, which seemed to work quite successfully to their advantage. In the last few hours before submitting for the final tournament, we tried mimicking this behavior, but clouds did not seem to be prevalent enough for this to make a large difference. Moreover, our amplifiers were too unpolished for it to be viable and we still weren't sure exactly when to make them. While there were some attempts to use them, no amplifier code ever beat our code without amplifiers, so they were scrapped.

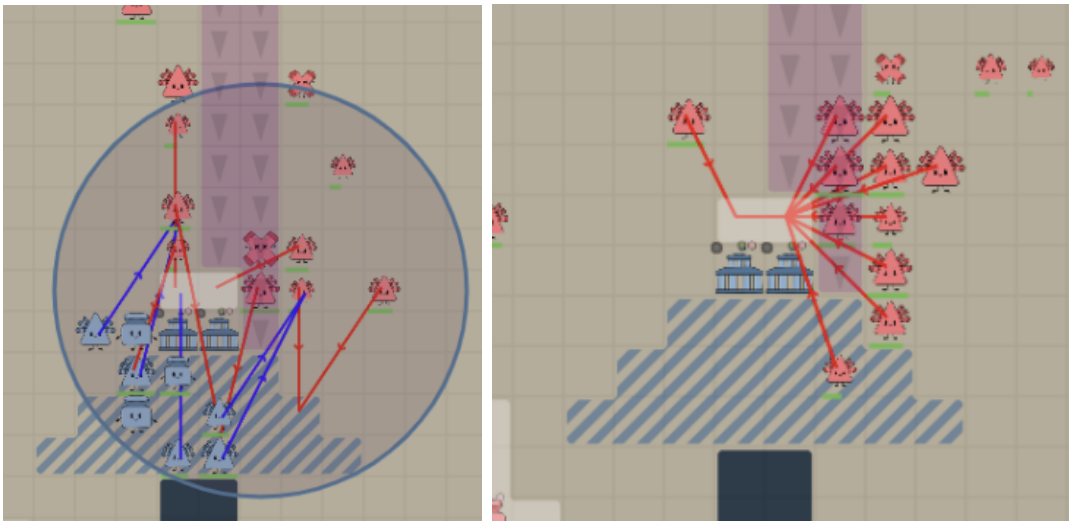


Figure 7: Destabilizers slowing down the enemy to help out our launchers and us having control only a few turns later.

2.5.4 Final Carrier Changes

Currently, carriers get assignments, and then they gather the resource they're assigned to. Pretty simple. At that point, the carrier kind of lives in a bubble and doesn't care about what's happening in the rest of the game. But what if it's advantageous for our carriers to switch which resource they're gathering? In particular, say that there ends up being a combat sector near our HQ. In other words, enemies are approaching us. Well then, we need launchers - we're under attack! So carriers currently gathering adamantium should start gathering mana so that we can build up a stockpile and begin building reinforcements as fast as possible. Carriers taking the state of the game into account and adapting to it is important because it makes them part of the combat and makes us more flexible.

We also made carriers lattice around the well, meaning that they stay 2 units apart from each other. If they all clump together around it, then not only is it difficult for new carriers to find space, but it's difficult for launchers to make it through if they need to go through the well to get to their destination. Launchers lattice at heal locations for the same reason. Finally, we made carriers "box out" enemy carriers when going to mana wells. In many games, both teams' carriers were vying for the same mana well, so if you can prevent enemy

carriers from using these wells by not letting them get close enough, this can crush their economy.

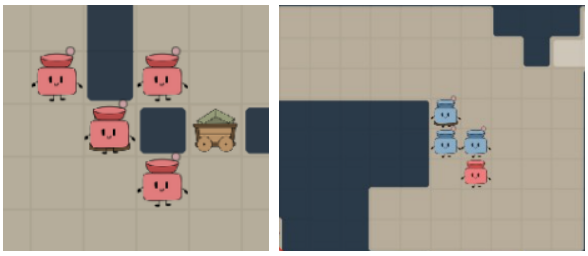


Figure 8: Carriers lattice around a well diagonally so that units can move in the squares adjacent to them and, blue carriers box out a red enemy carrier.

2.5.5 Final Launcher Changes

We've been taking advantage of symmetry with respect to enemy HQs so we know where to hunt. But...the whole map is symmetrical! If we see mana wells or adamantium wells by our HQ, we have a pretty good idea of where some other ones are on the map. There are two advantages to going towards these. First, it's more mana options for our carriers. Second, if we send launchers towards these locations, then the enemy can't get any mana, and they won't be able to build launchers. This will weaken their defense. Now, we're focused not only on making ourselves stronger, but also making our opponent weaker.

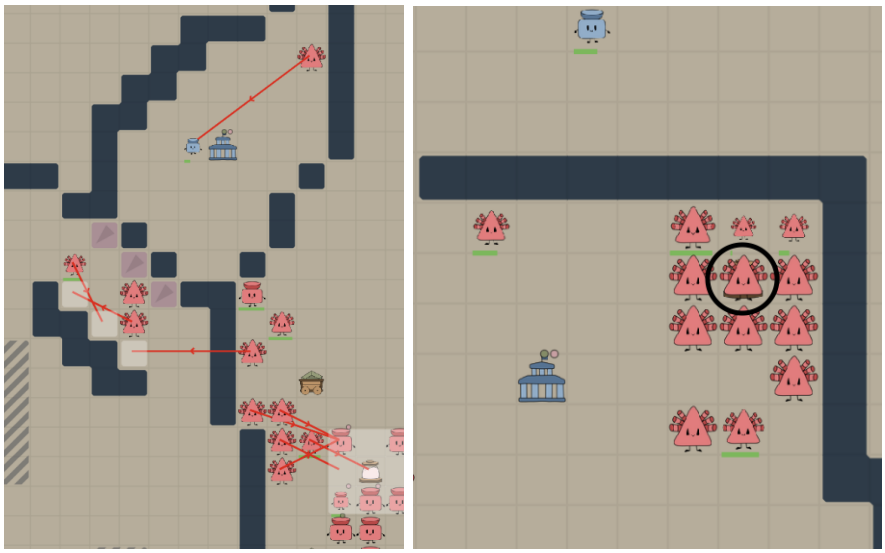


Figure 9: Launchers guarding enemy wells.

Throughout the tournament, launcher attacking micro has never been our bots' strong suit (unlike previous years). One of the main differences with this year's micro is that the vision radius and attacking radius of troops are almost identical. It is even possible to not see an enemy on one turn and then be in attacking range on the next turn. For the majority of the tournament, if we didn't see any enemies, we would just do normal activity - even if we had just left the vicinity of an enemy! The key scenario to handle was what to do if we had just seen an enemy but no longer see enemies. Our first attempt was to just wait

for some fixed amount of turns, but this didn't end up working super well. Instead, what worked is trying to go directly outside[1] of the vision radius of the closest enemy troop. As a result, enemies would be forced to come into your vision radius to move forward, giving us a slight advantage. We also added another slight optimization: if we see an enemy and deem it worthy to move forward to attack this enemy, we now may see some enemy that is better to attack, so we should switch our enemy target after moving.

3 Final Thoughts

This is our third strategy guide. In our first, we had a section giving advice to new players. In our second, we had insights into how we approached the game as a returning team rather than a new one. Now, with this as our last year eligible for Battlecode, we thought we'd dedicate a section to reflecting on the activity as a whole.

3.1 This Year's Game

This game repeated a lot of the drawbacks from last year, with fixed unit costs leading to very similar strategies and a useless, difficult to obtain resource that very few teams used. Although the new communication restrictions were annoying at times, they actually were pretty interesting and forced us to come up with creative methods. The addition of amplifiers and the fact that you can spend resources to make communication easier was a nice touch. A big improvement from last year, though, was that whereas 2022's anomalies were completely ignorable, you could not get through 2023 without coding for currents and clouds. They were 100% part of the game and needed to be part of your strategy. We like it when all the mechanics are relevant. The downside of that, though, is that some maps became very frustrating. Maps where it's easy to get stuck behind walls or be unable to make progress because of backwards currents were not very enjoyable. Jail, in particular, was a map where spawning towards the wells, the strategy that literally every team used, would result in your units being stuck in a jail. The result was that every single match went to 2000 rounds and went to whichever team happened to stockpile resources rather than build more things. Walmart and Potions were similar in that regard. It was better when there were things like rubble, which made it *harder* to move, but not impossible. There were far too many maps where nothing happened for 2000 rounds. On the plus side, the carriers were very well-designed robots. The fact that their speed changed based on their cargo and their capability to use their cargo to attack was very fun. Launchers were simple but effective as well.

3.2 David's Reflection

One thing I like about Battlecode is the teamwork. Battlecode has so many different components to it, and different components speak to different people. As a majority-systems programmer, I love building infrastructure. When our state stack was getting too large our

[1] There's actually a bug here. Bonus points if you can find it! Fixing it actually made our bot worse, and it was too late in the tournament to see why exactly. So we left the bug in.

first year, I figured out how to refactor our code into a system where you toggle the state at the beginning of each turn based on the current information, and we still use that feature now. Last year, using failure handling concepts from distributed, I spent days building the capabilities for our HQs to share resources and pass tokens around so that we could still manage everything effectively if one of our HQs died. This year, using concepts from databases, I created a system where we could manage sector information given the new communication restrictions. But that's only one part of Battlecode. Maxwell spent a lot of time watching matches and analyzing the strategies of other teams, and that kind of stuff just didn't click for me like it did for him. I see Battlecode matches like a beginner sees a chess match; it's just a bunch of pieces moving around. Winston had in-depth language knowledge and was able to create these custom data structures that would use minimal bytecode or create a navigation library using all these different pathfinding algorithms, all of which I had very little knowledge about. Everyone played their part at some time or another and utilized their different strengths.

I think the Battlecode codebase is very nice. It's a relatively simple class inheritance and coding style that intuitive for advanced programmers and beginners alike. It's a great way to make programming more fun. I think the games are rather complicated, though. There are more rules than your average board game, and they keep changing every week. There's a general lack of balance, leading to most teams converging towards the same strategy. Perhaps it would be better if there were levels, maybe a very simple game for beginners, a moderately complex game for most people, and then an extremely complex game only for teams who want a ridiculous challenge. I also don't love the odd desire for ridiculously difficult maps. Watching matches go to a 2000 round tiebreaker because neither team could get past all the walls is not fun. I prefer to watch teams at their best. Although I wish Battlecode was earlier and didn't coincide with the spring semester (though I understand that's necessary for MIT students taking it as a class), I really like the pace with a tournament every week. It rewards hard work and persistence. Overall I had a blast, and it was a great activity to practice my programming skills on and have fun competing in.

3.3 Maxwell's Reflection

I agree with David about basically everything, so I'll keep my section shorter. First off, if you want to get involved in Battlecode, you should go for it! Working on a bot with friends is super fun and rewarding, regardless of how many strategies you can add. If you want to try and improve as much as possible, reading post mortems like these is definitely the right way to go. There are lots of good ideas in them, and they usually link to people's GitHub code, which can be super helpful (as you can see, a lot of our code this year was copied or heavily inspired by others lol).

Overall, my favorite parts of working on the player was watching games and finding optimizations that could be made. Lots of times, the bot doesn't do exactly what you want it to do and you only find these things out by watching a 500 round game and finding a bad movement decision between rounds 134 and 135. I had a lot of fun with Battlecode, from finding areas of improvement as mentioned above, to celebrating victories with my teammates, to going to MIT and meeting all the other amazing competitors. I'm grateful that I was able to participate in this competition, and hope the competition can continue to

bring joy to it for years to come.

3.4 Winston's Reflection

Darn, David and Maxwell sure did say everything. I'll keep this short and sweet too I guess.

Battlecode is absolutely my favorite way to start the new year. I found out about it in high school but was too scared to pick it up alone. I had read postmortem after postmortem, everything from the fabled regex pathfinding to Cory Li's bytecode hacking, and all I could think was, "Wow..." To anybody reading this who might have the same feeling, all I have to say is: Do. It. You will not regret it. If you can find a few friends to do it with you, then all the better. Battlecode really does have a place for everyone: for the newbie who is just learning how to code; for the ambitious competitor trying to sneak their way in; and for the veteran team who is trying to bring it all home in their last year. Battlecode poses fascinating challenges and fosters a community that cares deeply about the game and everyone involved, and that's what I love most about it.

3.5 Bharath's Reflection

My teammates above really covered all the bases. After 3 years of doing Battlecode, it's been a great story to tell all of my friends. A month-long hackathon? I guess that's the best way to describe it. It's super fun, and the vibe of working with your teammates towards a deadline is 100% worth it. Match analysis, seeing changes come to life, watching the hundreds of pink lines on screen to analyze our team's path finding, and coming up with new ideas is thrilling. It is hard to implement new ideas, because there's so much trial and error, but it's all worth it in the end. As this is our last Battlecode, peace out from Bharath and the 4 Musketeers.