Just Woke Up - BattleCode 2025 Postmortem

Tim Gubskiy and Andy Nguyen

Introduction

Hi everyone, thanks for coming to read our postmortem. This is being written and released *significantly* later than expected, but we thought we should take the chance to document our 2025 battlecode journey and also share some tips for people hoping to compete in the coming years. We are both undergraduate students at Princeton University studying Electrical and Computer Engineering with minors in Computer Science. Tim has been doing battlecode since high school, with his first competition being Soup in 2020, and this was Andy's first year doing BattleCode.

A lot of this document is us word vomiting our thought process and iterative improvements we made throughout the competition. There are some good ideas scattered throughout, but they may be pretty game-specific. Feel free to dig through it, but if you're looking for some generic battlecode tips or a TLDR, please skip past everything to the last few pages of the postmortem!

Game Overview

We'll keep this brief since most of you hopefully already know how this game works or have read others' post-mortems. This year's game was a 2D Splatoon, where you need to paint over 70% of the map with your team's paint color to win. These were the units you could use:

- Paint Tower
  - Makes paint for units
- Money Tower
  - Makes money you can use to build/upgrade towers and build units
- Defense Tower
  - Shoots enemies to earn money
- Soldier
  - Can paint individual tiles on the map and attack towers
- Mopper
  - Can clean enemy paint and attack enemy units
- Splasher
  - Can splash a large area with paint and can paint over enemy paint

Sprint 1

Sprint 1 is always a bit of a toss-up and a chance for all teams to get a feel for the meta and see what strategies work and don't. Here was our general approach for the first tournament:

Infrastructure

Our code infrastructure was very similar to our code from previous years, which were pretty much ripped from other teams that had good modular code. We had a very object-oriented approach, having a super Robot class, then sub-classes from Units and Towers that inherited from Robot, and finally individual classes for each unit type. This made it easy to share methods between different unit types, but still have individual files for each unit type to make code organization easier. In previous years (back in high school), I sort of just dumped everything into 1 file, and it made it really difficult to find different parts of my code. Although it was simple to spin up, this year's structure was a huge improvement over the past years.

Another key improvement made to our infrastructure this year was to make state machines for all of our units. Our flow for figuring out what our units should be doing is we first run a **senseNearby** function that senses all the information we can around us, updates our mapData object, then we run a **determineState** function which uses our surroundings, resource levels, and previous states to decide what state our unit should be in, and then finally we execute the respective functions for each different state. This was especially helpful over previous years as it made it easy to figure out what our units should be doing without using long chains of if statements and various conditions that make it progressively more difficult to add new states and behaviours.

Communication

For Sprint 1, our communication was pretty limited. We basically just marked where we were building SRPs by placing a 1 in the center of the SRP. This was a big focus for us in Sprint 1 since SRPs were extremely powerful and we had a dynamic tiling approach, which benefited from very explicit communication about where an SRP was being built and where one wasn't.

Pathfinding

Our pathfinding for Sprint 1 was very simple. We created a function called safeFuzzyMove, which, as the name suggests, basically moves vaguely towards a target destination, but will avoid moving to tiles with certain properties, prioritizing moving on ally paint and avoiding towers. This pathfinding worked fine for the simple default maps but was already starting to be lacking in the new Sprint 1 maps, and definitely needed to be overhauled in future tournaments.

Exploration

Our exploration was actually shared for all of our units, and for Sprint 1, we just stole XSquares' exploration code from a previous year. This was lowkey a mistake because we didn’t really understand how it worked, and it ended up not working very well, so we got rid of it for future tournaments and created our own exploration logic.

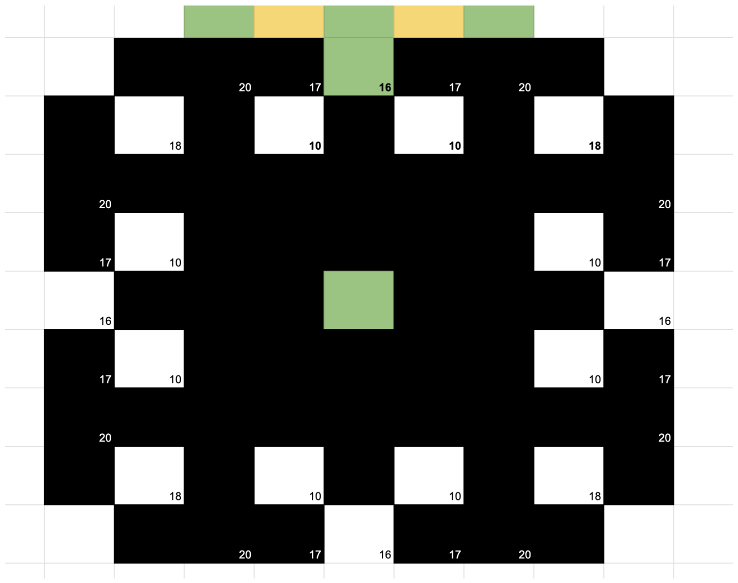
Tower Sacking

While tower sacking/tower flickering became a more relevant strategy later in the tournament, when Gone Whalin’ demoed it for the world in Sprint 2, I believe we were actually the first team to try it, managing to take a game off of Super Cow Powers in a scrimmage, which was huge for us because Super Cow Powers always seemed like an unbeatable God in the world of Battlecode. This strategy was super powerful in Sprint 1 since towers had no cost to build, and you could quickly destroy and rebuild a tower for free, gaining 500 paint instantly. This was, of course, promptly nerfed into the ground, but we did compete with this strategy in Sprint 1. It served us pretty well, although generally better micro and macro from other teams managed to win over this strategy.

SRPs

This was the most interesting part of the Sprint 1 tournament and likely what we spent the most time optimizing for. Many teams for Sprint 1 simply used static SRP patterns, determining where to build them by a formula that takes the x and y coordinates and tells you if it's an SRP center. Teams like Super Cow Powers used this and managed to pack a ton of SRPs, but we knew that if you could dynamically pick where to build SRPs, you could pack more SRPs onto the map since things like ruins and walls can screw up your static SRPs. Our approach was to distribute the computation of figuring out where we can build SRPs over many turns, by maintaining a 2D boolean array the size of the map, which specifies whether you can build an SRP at that location or not. We would update the array when we discovered a ruin, setting a 9x9 square around the ruin to be excluded for building SRPs, as well as exclusions around other SRPs and walls. This sounds very computationally expensive, and indeed it was. When first implementing this, we often went over the bytecode limit if we discovered 2 ruins at the same time or ran into similar situations, but through some array writing optimizations, we were able to get this usage under control. The benefit of doing all this computation every turn was that when we wanted to check where we could build an SRP, we just needed to do a single query into the 2D array, making it computationally trivial to check every possible location around us to find the best location to build an SRP.

To make tiling optimal, the exclusion zone that we built around existing SRPs had little holes in it, which would specify that you could build an SRP overlapping another SRP, as long as it is in a few specific locations. Here is what the exclusion pattern looked like for the first SRP pattern:



Another consideration was that we were able to get away with just using a boolean array for SRP exclusion zones in Sprint 1, since we were only building money towers and flickering them, meaning we did not want to build SRPs that overlapped the money tower pattern. In future patches, we transitioned back to building normal towers without flickering, and instead of using a boolean array, we used an int array, which we incremented whenever something was causing a tile to be excluded and decremented it when it was no longer there. For example, while trying to build a tower, you shouldn’t build an SRP next to it, but as soon as the tower is done, you can remove the exclusion from the array and are now able to build an SRP next to the tower.

General Strategy

Our individual unit micro in Sprint 1 was not fully polished; as such, our general strategy was quite crude and definitely had room for improvement. I believe in this tournament, our soldiers basically rushed building towers and then painted around the map as much as possible while putting down SRPs whenever possible. Our moppers didn't really even attack in this version, just trying to clean up paint wherever they see it, and finally our splashers just rushed in and tried to bomb as much enemy paint as they could. We primarily won by coverage. I think our Sprint 1 bot didn't even attack towers yet, since we thought it wouldn't be worth it (boy were we wrong). Other than flickering and nicely placed SRPs, our bot didn't do anything particularly special, and our final placement reflected that, making it into the top 16 but losing our first game against Baby Ducks.

Sprint 2

Adapting to Patch Changes

The main changes we had to adapt to in the post-Sprint 1 patch were quickly updating our SRP patterns, since they were heavily nerfed in the new changes. They made the patterns overlap far less, which was a huge nerf to us since we specifically invested a lot of time in making sure we would optimally overlap the patterns. However, our algorithm was very easy to adapt because all of the pattern checking was set using a 2D overlap exclusion array, so we could just update it with the new pattern. We also needed to move away from tower sacking, as they added much higher chip costs associated with towers.

New Tower Building

While for Sprint1 we could fully avoid needing to build different towers since we were tower sacking, with the sacking nerf, we needed to move away from that. Initially, our idea was to deterministically build a tower in a specific location based on the coordinates. We did this by multiplying both coordinates by different prime numbers, and then adding them, and then modulo by a number and setting custom thresholds to adjust the frequency of different tower types (sounds complicated, but basically a random number generator with the coordinates as the seed). This was an okay solution, but it did not take into account what kind of towers we had already created, and we found that on some maps, the winner fully came down to which side got luckier with the types of towers created.

To fix this, we ended up using markers to indicate what type of tower is being built, so multiple soldiers can collaborate, and whatever the tower was marked as in the first place is what it would be built as. We marked towers using a secondary paint marker in one of the top three locations adjacent to the ruins, and the position that the marker was placed indicated the tower type. Using a different marker type than the one we used for SRPs means we didn’t have to waste any bytecode trying to figure out whether a marker was indicating an SRP or a tower pattern.

Because sacking was no longer a viable strategy, we also began upgrading our towers to get the best use out of them. For Sprint 2, our strategy here was a little coarse, but ended up working alright. We basically made sure that towers would only upgrade themselves if we had a ton of money to spare, and a key macro decision was to make the money threshold for upgrading a tower dependent on how many allies the tower sensed around it. A tower that had 3 or more allies in its range would get upgrade priority since it is clear that it is in a higher traffic location, and is relied on by more units.

Improved Long Distance Communication

While I already talked a little about how we used paint markers to communicate, we also heavily utilized the other form of communication available this year, a complicated messaging system. This year, robots were able to send 4-byte messages to towers, and towers could send messages to robots if the robots are within a pretty close range of the tower, and crucially, connected to the tower by ally paint. Towers were additionally able to broadcast messages to other towers, without needing to be connected by paint, at a much further distance. Our communication system for Sprint 2 relayed 2 pieces of valuable information: tower locations and backup requests for moppers to clean up enemy paint. Knowing the location of all ally towers is insanely useful early game, especially on very large maps, since units need to refill their paint to continue to be useful. If you spawn at a money tower and don't know the location of a paint tower, there is a good chance you will die looking for it. Our crucial innovation, which I saw very few teams implementing even at the finals, was that at the start of every game, we wasted a few turns ensuring we were connected by paint to our spawn tower, either by moving to a connected location or painting a connection ourselves. This meant that if our towers were within broadcast range, we would be able to know the positions of all of our starting towers right off the bat, information that served us insanely well.

For our actual communication, we used some simple byte encoding to encode the information we wanted to share in as few bits as possible, to make the best use of the limited 4 bytes we had access to, only 32 bits. We spent 3 of these bits encoding the message type, 12 bits of the x,y coordinates of the map location we want to share, and another 2 bits for the tower type, with bits to spare in case we wanted to add anything in the future. All of this was done using bitshifts to make it as bytecode efficient as possible. You can take a look at our Comms class under Utils to see exactly how we made this work.

Pathfinding

For Sprint 2 I finally added some actual pathfinding. While every year I have toyed with BFS or other advanced pathing algorithms, I always come back to the tried and true bugnav. This year the bugnav I used was slightly more advanced though, and served me very well. I stole the code from a very well-ranked team the previous year, and as I am writing this almost a year later, I unfortunately don't remember their team name and cannot credit them, but if you recognize the code, please let me know, and I will add an acknowledgement… I made some heavy modifications to this navigation to work better for this game, since units want to prioritize tiles with paint when they move. We made an elaborate tileScore function that would assign a score to a map location, based on paint color, adjacent units (since we were penalized for being adjacent to other units), and a HEAVY penalty for getting into range of enemy towers. Then, in our bugnav pathing, we would use this score for navigation unless we were in a wall following mode, in which case we would use regular bugnav, with the exception that a tile that was in range of an enemy tower was treated as a wall effectively.

Automations

AB Testing

One monumental addition to my workflow this year was the use of some sort of automated testing. I always had some sort of versioning in past years, I would make a new bot and test it against an old version to see if it was better, but I was always going off of vibes. I would test it on a few different maps (usually my favorites) and then see the results, not very scientific. This year, I “borrowed” a script that was circulating in the Discord from the team camel_case (https://github.com/jmerle/battlecode-2023/tree/master) that enabled me to AB test two bots on every single map I had at my disposal. Although the script took some time, it would show me on which maps my new bot won, lost, or tied, and I could go through the specific map replays to see where my new bot was lacking, where it was improved, and why on some maps it came down to a side-based coin toss. This heavily changed my workflow, although sometimes there were some downsides. I would make a change that I felt very confident in, and maybe spent a lot of time on, and for some reason it would lose to an old bot. It is frustrating when something like this happens, but in the end you need to make a judgment call. Just because a new bot is worse against your old one doesn't mean it will be worse against other teams. That's where my second script came in.

Automatic Scrimmaging

I created a script that would automatically log in to the battlecode website and request to scrimmage with as many enemy teams as I chose. I would usually do this after I built up enough significant changes, and I wanted to see how it would stack up against around the top 20 teams, scrimming against whoever was generous enough to have their unranked scrimmages available. I would always then go through the replays where I lost, or ones where I won against a team I thought was better, and see what the heck happened, an insanely valuable learning experience and a great way to improve your bot.

General Strategy

Soldiers

In Sprint 2 our soldiers had 5 states they could be in: Connecting to tower, Exploration, Combat, Refilling, Building, and SRP Building. They also had a few state invariant actions they took every turn, a pre-state action, and a post-state action.

Exploration

This state was our general movement state, which the robot would be in when it was looking for new ruins, a place to build an SRP, or searching for enemies. Within exploration, we had a few sub-states as well, depending on what kind of information we had.

By default, we would just explore semi-randomly. The soldier would pick a random direction, extend that direction to the edge of the map to get a target MapLocation, and then would pathfind towards that location until it was within a few units of it, and then pick a new direction and start again. One cool addition we added to make sure our starting units spread out as much as possible was to make our initial explore location dependent on our spawn location. If our spawn tower placed our unit to the north of itself, we would explore north, if it placed us east, we would explore east, and so on. This meant that towers could make sure soldiers explore in different directions by spawning them in different places, a common strategy we saw used by great teams in past years.

If we had more than 5 towers placed on the map (that the soldier was aware of) they would start pathfinding towards any empty paint they sensed with the intention to fill it in. This was intended to make it so we wouldn't waste paint filling in random empty tiles in the early game, but once we had sufficient map presence we would start increasing our paint coverage since that was one of the win conditions for the game.

For Sprint2, we also had some logic to make our bot patrol the enemy paint area by making them move perpendicular to where they sense enemy paint. The intention of this was to explore the enemy's territory without actually stepping on their paint, since there was a heavy penalty for this.

Lastly, an important addition in our soldier logic was adding some sort of memory to the soldier in the form of a “return location”. If our soldier was in the middle of some task, building an SRP or a tower, and had to go refill, once they were back in their explore state, they would immediately pathfind back to their return location to finish their task instead of randomly exploring again.

Building Towers

Soldiers would enter this state when they intend to build a tower at a ruin they discover. If the ruin doesn’t have a marker indicating what tower type it should be, the soldier who discovered it will decide its tower type, based on the ratio of paint and money towers that it knows about, and then mark the ruin appropriately. In this state, the soldier would try to find the best tile around the ruin to stand in, based on a scoring system primarily composed of what type of paint the tile was covered in. A choice we made that was very helpful was that we didn’t actually perform the painting actions in the tower build state, but made it a state invariant action at the end of our turn. When a unit discovered a tower or ruin, it would update its MapData object to note what color each tile around that tower should be. Then in the state invariant action it actually performs the filling, which I'll talk about in that section.

Building SRPs

This state was very simple, as most of the computation for this was done before we even got into the state. Soldiers will enter this state when they are within range of a tile that could have an SRP centered on it, and they exceed a certain threshold of towers already built on the map. This threshold was calculated at the start of the game based on the size of the map. We found that on small maps, we want to prioritize towers, since wasting time on SRPs often means we'll get rushed by the enemy, and ruin locations are generally closer to spawn, but on big maps, we have some more time before we start having any combat, and the benefits of the SRPs will have more time to compound.

Combat

Soldiers would enter this state when they sensed an enemy tower nearby, had at least a certain amount of health, which we tuned with a constant, and had at least 1 other soldier with them to engage in battle with.

To increase the amount of damage a soldier can do before dying, we made sure to employ a kiting mechanic that you most definitely have seen other teams do. This is a crucial piece of battlecode micro in almost every year. Because units have a separate action and movement cooldown, you can move and attack on the same turn. To optimize damage, you want to make sure that at the end of your turn, if possible, you are outside the range of the enemy you are fighting. This is very easy with towers since they cannot move, so on a given turn, if you are out of range, you enter range and attack them. If you are in range, you attack them and then move out of range. This makes it so that for every two hits your soldier gets on a tower, the tower can only get one hit on the soldier. This is made even better if you are attacking with multiple soldiers and are able to sync up their attacks, something we didn't employ in Sprint 2, but added later.

Refilling:

If our soldiers got below a certain paint threshold, they would enter this state, returning to the closest paint tower that they were aware of. If there were no paint towers, they could return to a money tower, and if that tower was empty, they would wander aimlessly until they found a tower or died.

Surprisingly, this state was one of our most complicated states, since it's difficult to decide what the optimal choice for refilling is, depending on how many towers you know of. If you come to an empty paint tower, is it more efficient to quickly wander to the other paint tower you know of, or should you wait until the tower has more paint? Our strategy was that if you knew of a paint tower, you would sort them all in order of how far they are from you, go to the closest, and unless the tower was destroyed, you would stand at the tower and wait your turn to refill. A crucial optimization was to make sure that the soldiers were standing on ally paint around the tower, and they would use their little remaining paint to paint under them so they wouldn't passively lose paint over time, causing them to die. In this year's game, there was also a penalty for having adjacent robots, so our robots would all sort of stand in orbit around the tower, not touching each other, until the tower had enough paint, and the first bot in the turn order could collapse on it and refill. Our soldiers were also gracious enough to paint under moppers that were waiting to refill, since they couldn’t do it themselves and would die otherwise.

Post State Actions

In this state is where all of the actual painting happened. We made a helper function that made it so that whenever you would paint a tile, you would automatically paint it the correct color for that tile, based on tower patterns and SRPs nearby. This was useful because you can take paint actions to do things other than completing patterns, but it would be useful to not have to repaint tiles. For example, our soldiers would start by prioritizing painting under moppers who are close enough to a ruin they are trying to complete. This is because if a mopper is helping you clean a ruin, you want to minimize the cooldowns they incur to make them clean enemy paint as fast as possible.

Next, they would prioritize filling the tile underneath them to minimize the passive paint loss, then they would fill tiles around ruins, in a spiral coming out from the center of the ruin, then they would fill around themselves in a spiral centered at their location. Because SRP patterns and tower patterns were all encoded in our MapData object, anytime they filled a tile, it would usually be helping toward one of these goals, and we didn’t actually have any painting action in the Building Tower or Building SRP states themselves.

Moppers

Improving mopper logic was one of the things we did between Sprint 1 and Sprint 2 that I believe had the biggest impact on our bot's performance. Our moppers had 4 possible states: Exploration, Combat, Refilling, and Mopping. At the start of our turn, our moppers would sense their surroundings and pick their state as follows. By default, they would be in the explore state. If they sense any enemy paint near a ruin, they will go into the mopping state. If they sense an enemy, they will go into combat mode, and if they sense any enemy paint elsewhere, they will go into the mopping state as well. If they are low on paint, they will go into refill mode.

Exploration

Our exploration logic for moppers was very similar to the soldiers' but with a few slight differences. Firstly, soldiers were able to communicate where they needed moppers to clean through the messaging system. If a mopper was requested at a location, they would pathfind to that spot during their explore state. Moppers also had the return location logic of the soldiers; if they ran out of paint while they were busy cleaning something, they would return to that spot after refilling.

Combat

Combat with the moppers was quite strange since they had two abilities to deal damage, a mop swing and a regular mop action. Mop swinging was only beneficial if you were able to hit 2 targets at the same time, or if your target was out of range, since the swing could reach further. Our logic first checked if there was any direction we could swing in, before or after moving, that would hit 2 or more targets. If there was, we would move and then swing. Otherwise, we would try to get in range of the best target we could find and mop them normally. If we couldn't get in range, we would try to swing again in their direction

Mopping

This state was relatively simple; if we sensed paint, we would move towards it and try to mop it up. We made a few optimizations to make this state better, like prioritizing standing on ally paint to reduce cooldowns and prioritizing enemy paint that is close to ruins so that we would be able to paint our own pattern and build a tower.

Refilling

Our refilling logic was actually identical to the soldiers; we used the same refill function for all three unit types.

Splashers

Splashers were one of the strongest units in the game, primarily because they could instantly replace enemy paint with ally paint in a large area, and getting them working well made a big difference in this year's competition. Our Splashers only had two states: Exploring and Refilling, but they also technically “attacked” or painted the map in their explore state.

Explore:

In this state, the default exploration was the same as for the moppers and the soldiers, but we prioritized moving towards enemy paint if we sensed any. We then wrote a big “algorithm” to determine where the best spot to splash would be and if it was even worth splashing. Since a “splash” would cover a 3x3 area, we made a small 2D array that kept track of how many adjacent enemy paint tiles there were for each map location in our range, and we would splash the max value location that was in range, or reachable within one step. We also had a minimum enemy paint threshold of 3 to avoid wasting splasher paint hitting something a mopper could easily clean up, unless the enemy paint was close to a ruin, in which case it would be more beneficial to quickly clean it up without waiting for a mopper.

While simple, this logic served us really well. The splashers would basically rush into enemy territory, avoid standing on enemy paint themselves, while destroying the enemy's progress, getting deeper into their territory quickly.

Refilling:

Once again the same as the other two units.

Towers

Our tower logic was pretty simple for Sprint 2; their job was to spawn units, defend, and relay messages between units. We added a lot of logic, which we frequently tweaked to adjust how our towers would spawn units and in what order. What we found worked for Sprint 2 was spawning 2 soldiers, then if we needed moppers, spawn 2 moppers, and finally spawn a splasher. This ratio would change depending on certain factors, like how many allies were around the tower, and if the tower was one of the spawn towers or one created later in the game. Once we have established map presence, we focus on building more moppers and splashers than soldiers, since they are more useful in the late game.

For defense, we added some micro to ensure we were spending our attacks as meaningfully as possible, targeting the weakest soldiers in order to kill them first, minimizing the number of enemies that can attack us at once.

Results

Our bot in Sprint 2 did better than I could've possibly imagined, not only did we make it into the top 8, which was already a huge accomplishment for us, but we also managed to beat Super Cow Powers to get there, who for as long as I have done battlecode I always perceived to be the greatest battlecoder in the world as he has always topped the charts, frequently dominated in Sprint tournaments, and if you look at any code he shared in the discord it looks like he's writing battlecode with wizard hieroglyphics (he has some weird java python wrapper that lets him write more bytecode efficient java with code gen). We had 5 insanely close games against him, narrowly beating him 3-2. Of course, part of this was a ton of hard work making our bot as strong as possible, but we also got really lucky with map selection, and the fact that his bot's strategy did not expose the glaring weaknesses in ours.

To see these weaknesses we just need to look to our next set against Asteroid, one of the strongest international teams this year, who managed to beat us 3-0 by purely rushing down our starting towers, destroying our money tower and making it so that no matter how many ruins we paint there's no chance we can build a new tower to get back into the game. This was potentially luck, as it happened that the three maps he beat us on had extremely close starting spawn towers, but nonetheless, we realized that we needed to heavily amp up our defenses for future tournaments.

Qualifiers

There wasn't a ton of time between the end of Sprint 2 and the qualifiers submission, but we made some adjustments to learn from the mistakes we made in the previous tournament. The only change in the rules between the tournaments was that spawn towers now start at level 2, a small nerf to the rushing strat since they have more health, but it makes it even more important to protect them since they are worth more to replace.

Towers

The main tweaks we made for towers were slight adjustments to the spawning micro that we honed in by doing AB tests against our various versions of old bots, but more importantly, we added defensive logic when our towers get rushed down by enemy soldiers. We did this by making our towers prioritize spawning moppers whenever they sensed enemy soldiers in range. Although this could sometimes set us back in terms of building other towers, having our starting tower have a chance of not dying was way more important and allowed us to successfully defend some rush attacks in the qualifier tournament.

Soldiers

We added a few small optimizations to soldier logic, but nothing overwhelming. One tweak we added was to make it so that as soon as a soldier finished building a tower, they would instantly go into a “quick refill” mode, in which they would just refill their paint at that new tower and proceed as usual.

We also made our soldiers more aggressive, making it so they would attack enemy towers even if they did not have ally soldiers alongside them, and we made them more aggressive with tower building, opting to finish building the tower instead of going to refill if they were low on paint but had just enough paint to finish the pattern. Our tower logic also made soldiers more aggressive by spawning the first soldier in the direction of the enemy, and spawning all soldiers after a certain point in the game, pointing towards the enemy base.

Exploration

We noticed that when our soldiers would explore in a straight line, they would always come really close to a ton of ruins but miss them cause they were just out of sight. We added a zig-zag exploring mode to the soldiers to make it so that on the way to a specific endpoint, instead of moving straight to the target location, they would either zig left or zag right on the route. This is insanely helpful when you are trying to explore in a cardinal direction away from you (North, West, South, East). When moving north-west, you move just as fast north as you would moving straight north, so adding this small optimization, most of the time, would allow our bots to explore just as fast but cover much more area.

Building

We slightly reworked where our soldiers stood while building towers. We realized we were orbiting the ruin too much, sometimes making it so we would miss a turn where we could've filled an empty tile, so optimizing this was a small improvement that made us build ever so slightly faster. Additionally, we made it so that if a paint tower was in progress of being built but a soldier with more map knowledge arrived at the ruin, they would be able to re-mark the ruin to build a money tower instead of the paint tower. This was to help avoid situations where we wouldn't have enough money towers and be completely shut out of the game, unable to spawn more units.

Ruining Ruins

A strategy we noticed a few teams doing in Sprint 2 was painting a single tile near an empty ruin, even if you're not in a building state, so that the enemy would not be able to complete the ruin unless they brought a mopper to clean it up. In our soldier logic this was endearingly labelled as a "cuckLocation" in our code. If we noticed a ruin had no ally paint around it we would just add a tiny blop in the easiest location for us to do so. This was especially useful when doing a rush strat, because even if you fail to destroy any buildings you at least slow the enemy's expansion down, potentially giving you the ability to get a headstart on tower building.

Moppers

Post Sprint 2 we mostly made minor optimizations to the moppers. The main changes we made to moppers was to improve their combat ability, improving their target prioritization to prio low-paint enemy soldiers to reduce enemy numbers quickly, and we made them prioritize defending towers over removing paint. Their mop swing logic was slightly improved as well as we found that the previous code was actually not optimal (oops) and we corrected a few bugs in that logic. We finally made a change that if the mopper did not know of a paint tower that it could go refill at, it would just bum rush enemy soldiers trying to get paint out of them in order to survive longer, as that would be better than most likely dying searching for paint towers.

We briefly experimented with the idea of using moppers to transfer paint to dying soldiers, known as our rez (resurrection) state, but in our AB testing this seemed to do way worse so we scrapped the idea.

Splashers:

For qualifiers we mainly did bug fixes and small optimizations on splashers, with no huge behavioural changes.

We found in some replays that sometimes our splashers would just… get stuck, looking at enemy paint that they wouldn't splash because it was below the threshold we set in code. To fix this we actually increased our minimum splash threshold to 4 instead of 3 but added a condition that if it could splash all of the paint that it sensed then it was allowed to do so. This usually resolved this problem, and we added a few conditions in our exploration logic to make our bot less often get stuck trying to move towards enemy paint when the direction to the enemy paint was direction.center.

We noticed in Sprint 2 that our splashers would sometimes run into tower range. This was a bug in our code that we were able to patch up for qualifiers.

Results:

We qualified!! It was a little underwhelming to qualify since unlike the international tournament where only the top 4 go, for US teams the top 12 get to go, and the tournament started at top 12… So as soon as we saw our name in the bracket we knew we made the cut. But nonetheless it was still insane to see our bot go on and place top 4 in the qualifiers, beating out some big battlecode names, but ultimately getting bested by Om Nom and Clog Will Mog. It's hard to say exactly what we could've done better in these two match ups. Om Nom was always the team that we had the worst match up against as their bot seemed to have a lot more aggression than our bot was prepared to handle, and they were always able to back it up with insane expansion behind it. We ended up losing 3-1, usually by getting our towers picked apart while our economy was outperformed at the same time.

Against clog will mog it was actually really close. We lost 2-3 in game 5 and there was one game, that it seemed like we were winning pretty handily before clog will mog built an insanely well placed defence tower in a critical choke point, which for some reason our bots kept running into, killing our units while also feeding the enemy money, since defense towers gain money when they destroy enemy units. Luckily though, catching this lapse and implementing the strategy ourselves for the finals may very well have been one of the main things that brought us to win battlecode 2025 (foreshadowing kind of).
Overall we were insanely happy to have made it to the finals. After around 6 years of doing this competition, being able to go to the finals at last was a dream come true.

Finals

Incredibly pumped to be going to MIT for the finals, we only had a few days to try to squeeze out any additional performance we could out of our bot. Once again I don't believe we had any super radical changes in strategy, but we added some minor optimization here and there that great boosted our winrate against our old bot versions, and a few changes that we believed would make our bot better suiting to dealing with the teams we were having the most problems with in scrimmages, confused and om nom.

Soldiers

The only thing we really changed for soldiers was their exploration logic slightly, and their tower selection logic. We added a step in our exploration, where if the soldier had enough paint and enough money to build a tower, instead of exploring aimlessly, they would b-line straight to the last place where they saw an empty ruin without a tower, accounting for tower locations that they only learned about through comms. This made it so that if our soldiers had the resources to build, they wouldn't waste as much time wandering around, improving our ability to spread and build up quickly. A last small change we made was ensuring that when our soldiers attacked together, they would synchronize their movements to move into tower range on even turns. This meant that when we were attacking enemy towers they would only be able to hit one of the soldiers per turn, giving our soldiers more time to damage the towers before dying.

Moppers

We brought back the resurrection strat!! Even though in our AB tests the results of rezzing seemed to be not super meaningful, we agreed that it was a feature we believed should make our bot better, and even if the numbers against our own bots did not show it, we thought it would work better against other teams. Other than that we made no major changes

Splashers

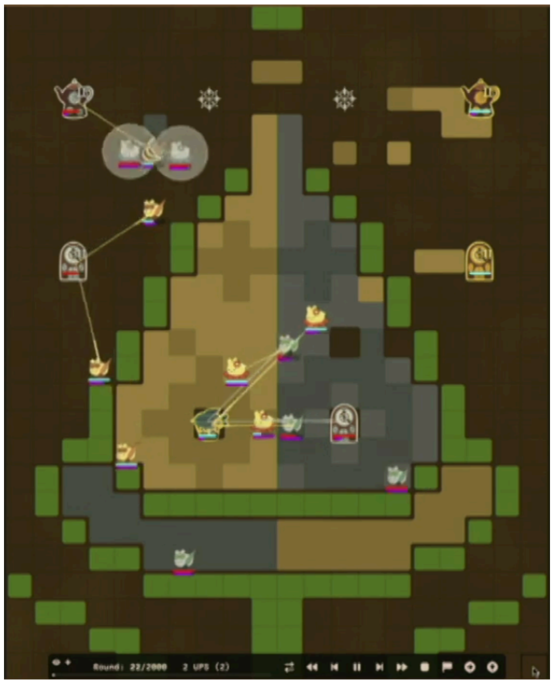
We made no major changes to the splashers, except for adding some bytecode limit checking logic, making it so that if our bots ran out of bytecode while computing the best location to splash, they would stop scanning nearby mapInfos and proceed with their turn. This made it so that if for some reason they were computing a lot on a given turn, they would still take some actions instead of exceeding their bytecode limit, causing them to effectively do nothing that turn.

DEFENSE TOWERS

After getting mildly crushed by some defense towers in the qualifier tournament, we started looking into if defense towers could actually be used viably. Up to this point very few of the top teams used defense towers. Everyone agreed they were very niche and likely just not worth using, but we knew that we needed to do something to counter some of the hyper aggressive teams that we often struggled against in scrimmages and in past tournaments so we tried really hard to get them to work.

Ultimately the idea we settled on was to build defense towers under the conditions that: we have less than 2 defense towers, the tower was in a choke point, determined by the fact that it had at least 1 wall on two opposite sides of the tower, and finally that the ruin was within 50 units squared of the center of the map. These conditions might sound kind of arbitrary, and they were, but that's what seemed to work best for us.

When we performed AB testing against our past bots, and using defense towers did not seem to be very good… Against our past bots it usually went even in wins, or even slightly lost. But our bot wasn't super aggressive, and we believed that this, in theory, SHOULD be better against the teams that we have the worst matchups against. So we sent it, and the results were unbelievable. Watching back the replays from the final tournament there were several matches where I could probably credit our win to a super well placed defense tower that just shut down the enemy for the rest of the match. And most importantly this let us win a few key games against our biggest threats, Confused and OmNom, likely being the reason we were able to come out victorious. Example insane defense tower build against confused, instakilling a bunch of units as soon as they spawned:



Results

Firstly, going to finals was an incredible experience. The battlecode team were amazing people and it was great to meet them, and getting to just chat and hang out with all of the other finalists was a fantastic experience. We definitely made a few friends at the competition and I was really happy to have had a chance to meet everyone there.

Our performance in the tournament itself was more impressive than we were expecting. Part of this was definitely getting lucky on the maps, but it was awesome to see the last-minute changes we made have an actual impact in many of our games. In the last few sets of the tournament, we managed to beat Om Nom in winners, got bumped out of winners by Confused, beat Om Nom again in losers, and then somehow won 2 sets against confused back to back, culminating in the most suspenseful game 5 I could've ever imagined.

Making it to finals was already a dream come true, we never imagined that we would actually be able to win the whole competition, but we still worked at it no matter how slim the chances were, and the fact that it actually happened was an absurd cherry on top, an experience that we'll never forget.

TLDR, Tips for Future Competitions

Start Simple

Battlecode can be a little overwhelming in terms of complexity, especially if it's your first year competing. My goal at the start of a new competition is to start very simple, get a bot that does something slightly intelligent up and running, and slowly try to understand the nuance of the new game by talking with others, watching replays and slowly adding new ideas to your bot. There's no need to immediately try to get a bot that does everything perfectly, get something barely working and upload it to start scrimmaging against people, and take everything one small step at a time.

Stay organized

As you add new functionality to your bot try to continue to keep your code and logic simple and make sure your code stays organized and easily readable. This is one thing I struggled with in past years and have slowly improved on. The first year I competed in battlecode, in 2020, I had a single file for my entire bot. One long spaghetti code file, with every unit and state all together. This made it pretty difficult to find my own code, and even harder to try to add new features and refactor. Take a look at some of the top teams code organization and try to copy the file structure to your own. Based on this strategy we ended up with having an inheritance file structure with a Robot class, a Tower and Unit class that inherit from Robot, and sub classes for each individual type of tower and unit from that year's competition.

Another thing we did to stay organized was to use a state based approach to our turn actions, setting up a state for each type of behaviour we wanted our bots to do (ex: combat, building, exploring, etc.). Each state would have its own function in our code, at the start of the code we would do a universal "determineState" function, which would pick the state for the turn, and then at the end we would have a state invariant action function that would happen regardless which state we were in. This organization made adding new behaviour way easier, as we did not have to comb through tons of branching if statements to figure out what our bot was doing.

Keep it Simple

When you look through some top teams code, or read what some people are talking about in the discord you may stumble upon a conversation about some insanely complicated path finding algorithm, whacky bytecode optimizations like loop unrolling or using strings as a datastructure. This may be very intimidating and you may think that in order to do well in battlecode you need to be implementing crazy stuff like this in your code, but for the most part that is not true. You can do well in battlecode without any of these wild optimizations and algorithms, but if you're interested in trying them out, like a lot of the people on top teams might be predisposed to, feel free to try, just know that it is by no means required or needed in order to make a highly competitive bot.

Read other teams' code and steal what you understand

Some top teams have created beautiful code in past years to do things like attack micro, pathfinding, AB testing, exploration and so on. I would highly recommend looking at past years code for inspiration, and occasionally perhaps implementing some of that code into your new bot. Be sure to do this carefully though. It might be tempting to see xSquare's micro code, or some teams insane unrolled BFS pathfinding and think you should go ahead and copy it into your bot, but I believe that it's important to have a good understanding of whatever code your bot is running, so that you are able to properly adapt and modify it to better suit the needs of the current years game. Stuff like automation scripts though feel free to just steal, they're usually pretty simple anyway but hugely helpful.

Work Hard, Dont burn out, and Have fun

A strong bot is not built in a single day (unless you're insane). It takes days and long hours of iterating, talking to others, watching replays, and trying new strategies. This year was the most time I've ever been able to allocate to battlecode as I was basically on winterbreak the entire duration of the tournament and would just wake up (ha) and do battlecode the entire day minus a few breaks for meals and the occasional hang out with friends. This might not be sustainable for everyone though, the most important thing is to pace yourself so that you don't burn out. I've seen teams, and personally experienced, start by working on their bot non-stop for the first week of the competition and then slowly fizzle out as they run out of ideas for their bot and get demotivated by seeing their rank on the ladder slowly drop. Stay consistent to the very end of the competition, have fun with what you're doing and put in as much time as you think you can without burning out and you'll be setting yourself up for success.

Watch Replays

Watching replays is one of the best ways to learn how to improve your bot. Seeing how other teams manage to beat your bot with new unique strategies will give you new ideas for how you can improve your bot, both in defending against other strategies and possibly implementing them yourself. Just running your bot against itself works alright but it only tests your bot against the strategy that you have managed to think of, and there might be countless other strats that other teams are running.

Do automation maybe

This was the first year that I did any automation and I found it insanely helpful for iterating my bot, being able to make a change and then test if it actually improves it over a test sample of every single map available to me. I say maybe this is a good strategy because I personally found it immensely helpful but I know of several high ranked teams that managed to make it all the way to finals without any sort of automated testing. So maybe this is something that you could test out in your development process, but if it's not for you it's not strictly required to make a competitive bot

Get creative, don't necessarily do what everyone else is doing

Every year there are a ton of unique strategies discovered for battlecode. Try to be creative with your bots strategy and don't necessarily just copy whatever you see other teams doing, maybe you'll come up with a new idea that could differentiate your bot from others and give you an edge in the tournament. While the general game plan was basically the same for all teams, the top teams all had some creative gimmicks in their bots that helped differentiate them, whether it was a rush strategy, tower flickering, defense towers, or some other whacky strategies, don't be afraid to try something different.

Similarly, Battlecode's bytecode limits sometimes restrict the kinds of data structures and algorithms that you can realistically execute within the bytecode limit. Feel free to try out new algorithms and ideas, not just whatever you might have learned in your intro algorithms class. This year I needed a way to figure out if a tile was too close to any ruin or SRP and remembered a data structure I learned in class called a k-d tree. It ended up being not even close to fitting within the bytecode limit, so instead I creatively utilized a 2D integer array to represent the entire map and mark excluded tiles whenever I discovered a new ruin or SRP. This distributed the computation over multiple turns and also had the benefit of benefiting from fixed bytecode operations like array copies. Don't feel boxed in by the algorithms invented by others and feel free to try something crazy and new.

Share ideas and talk to people

Another great way to refine ideas and explore different strategies is to chat with people in the Battlecode Discord. People will share different ideas, chat about why they may or may not work, and a lot of people are very friendly and willing to help you out if you have questions about anything. I've seen some people trying to gatekeep their strategies thinking it will give them a strategic advantage. I don't believe this is necessarily the best way to approach battlecode. Everyone can scrimmage against your bot and see your strategy anyway, but one of the best ways to refine your strategy is to share it and see what others can come up with. If you do want to keep your ideas private, at least chat about them with your team or friends who aren't doing battlecode. Sometimes you'll get ideas just talking about the game with someone else.

Iterate and Refine Like Crazy

This year there isn't one thing that our bot did that I can point to as being the reason our bot won. When I look back at our code I just see that we did a bunch of tiny basic things extremely well which we managed to do through a ton of iterations and refining our strategy. If you take a look at our github you'll see a whopping 57 versions of our bot (with progressively devolving names). Done over the period of about 25 days that means we basically made more than two iterations of our robot per day. I believe this was super useful as whenever we made a change we could quickly compare to our older bot versions, if the change was useful we would keep it and create more iterations, and if it was bad we can branch off our previous bot and try again.

Final Thoughts

This year's game was an absolute blast, we loved participating in it and this was the perfect way to cap off my last year of battlecode as an eligible student. Shoutout to Teh Devs for organizing this incredible competition and hosting an amazing finals, and thanks to all the other finalists for being great people to talk to and sharing a passion for this crazy competition. Good luck to anyone reading this in future battlecode years, hope you guys have a ton of fun participating in BattleCode!!