Battlecode 2023 Postmortem

Strategy report for team *no thoughts head empty*! Also available at https://github.com/reeceyang/battlecode-2023

Carriers

We set carriers to only acquire either mana or adamantium immediately after they are created — i.e. they only target that one type of well throughout the duration of their lifespan. To find mana wells, carriers would start from an adamantium well (since they can usually be found more easily) and fan out in five different directions based on the bot's ID number modulo 5.

The ratio of mana to adamantium carriers was hardcoded based on the round number and the map size. Before round 60, the ratio of mana to adamantium was 4:1 on small maps and 1:1 on big maps. After round 60, the ratio was 9:1 for all maps. We found that going for adamantium early on big maps helped us gave us a long-term economy advantage.

Our carriers move between different locations adjacent to the well. This is actually a part of our pathing: if a bot is adjacent to its target location but the target location is occupied, the bot is only allowed to move to locations adjacent to the target or stay still. Bots naturally try to move around the target due to our pathing algorithm, so this conveniently helps create space around wells for other bots to slip through and prevent congestion. We also had robots sitting directly on the well start attempting to leave the well once their collected materials reached a certain threshold, so they wouldn't clog up the well while no longer being able to collect.

Launchers

Our launchers performed a "kiting" maneuver during combat: a launcher would try to move out of its target's vision radius after attacking. If the launcher could sense enemies in its action radius at the start of the turn, it would shoot and then move. Otherwise, it would move, check to see if it could see any new enemies, and then shoot. If there were no enemies, it would shoot into clouds or at the last known enemy location.

When not in combat, launchers would either try to group up with other launchers, go camp at the nearest enemy-occupied island, or head to the nearest reported enemy in the communication array. If there were no active enemies in the communication array, launchers targeted potential enemy HQ locations, favoring the rotationally symmetric location first if no enemy HQ locations were known. We also hardcoded in behavior for "small" maps (defined as maps with area less than 1100 tiles). On small

maps, launchers would head to the center until our team had achieved map control. On big maps, launchers would stay at the headquarters for the first fifty turns to defend against early game rushes, then freely engage in the other behaviors described above.

Amplifiers

We used amplifiers to determine the map symmetry. The HQ would build amplifiers alongside clusters of launchers. Amplifiers then followed these launcher clusters around the map, continually checking for wells, islands, and enemy headquarters, and storing information about these locations in the shared array. Since maps could be either horizontally, vertically, or rotationally symmetrical, amplifiers tested each newly found location against known locations to exclude possible symmetries until they settled on one confirmed symmetry. This symmetry would then be stored in the shared array.

Having knowledge of symmetry gave our bots an advantage in a few ways. If a well was occupied or difficult to path to, carriers could instead decide to go to the symmetric well location. If a carrier found that the island it was bringing an anchor to had already been claimed by its own team, it could immediately target the symmetric island instead. This information also made it easier for launchers to target enemy headquarters before all of their locations had been reported.

We also used amplifiers for enemy reporting. Once a launcher spotted an enemy, it would attempt to write that location to the shared array. Nearby launchers would then travel to that locations.

HQ Strategy

In addition to building robots, our HQ would also periodically “bait” our team’s launchers to enemy HQ locations. It did this by using our enemy reporting mechanism to “hallucinate” an enemy at the enemy HQ. Launchers would then respond as if there was an enemy there.

Communication

Teh Devs provided a “message queue” functionality that allowed robots to store messages until they were in range to write to the shared array. This was implemented with an `ArrayList` from `java.utils`, which was very bytecode-hungry. We found that it was sufficient to just use a regular fixed-length array (120 worked for us) and overwrite old messages by iterating through all of its values). This reduced the cost of storing the messages by 2000 bytecode.

We were provided a shared array of 64 items, with each item an int up to 16 bits. To store information more efficiently in this array, we used the bitpacking techniques shared by teh devs in the MIT Battlecode class. Bots could only write to the shared array when in range of an amplifier, their own islands, or their own headquarters. Mostly, we got around this by sending amplifiers everywhere.

Navigation

Our navigation algorithm was heavily influenced by Team Battlegaode's 2020 Postmortem. We used Bellman-Ford on a 5x5 range around the robot. Tiles with currents were given a cost equivalent to the cost of the tile the current flowed into. We also made it impossible to move into a current opposing the direction of movement.

To better optimize our code, we stored the openness (whether it's possible to move into a location) and current direction of each tile in individual variables instead of arrays, and wrote a Python script to generate the unrolled loops for the algorithm. However, this still averaged around 6,000 bytecode per turn. To prevent exceeding the bytecode, we relaxed the graph less times (maximum three relaxations, minimum one) based on the amount of bytecode left for movement at the end of the turn.

Since our Bellman-Ford implementation only considered a small radius around the robot, we implemented bug navigation to help the robot pathfind around larger obstacles. If the robot failed to make progress within a certain time limit using Bellman-Ford, it would switch to bug navigation for a number of turns. The robot would continue keeping an obstacle on its right side until it reached a point closer to the target than before the obstacle, at which point it would start moving directly towards the target again. We made robots stay off currents in bug navigation, except for carriers that were able to move twice and "power through" the current.

Overall Strategy

We invested more time in micro compared to macro strategy. Fixing small bugs in our code or making small improvements turned out to have a big positive impact on our scrimmage performance.

Our macro strategy was not as refined. Up until the MIT Newbie tournament submission deadline we didn't have a substantial macro strategy. We mostly ended up throwing things together ad-hoc based on what we saw in tournament matches and scrimmages, which had an okay result.

Takeaways

This was our first time competing in Battlecode and we all enjoyed it a lot! We definitely learned many lessons for next time:

- Tooling for iterating and testing quickly is very important. Many times we would make a change and not be able to tell whether it was helpful or harmful without spending a long time testing. At the final tournament we found out that many top teams had built up custom systems for quickly running many matches in parallel for A/B testing. It might be worth investing time at the start of the tournament to build up these tools.

- Keeping code organized is important, both in how team members collaborate and how code is structured!
  - At the start of the competition we each worked on separate bots in different packages, then got together to merge all the code into a single bot manually. The merge was probably more time-consuming than necessary, as not all the code was compatible and we had to spend time tracking down bugs afterwards. We later realized that it works better to all collaborate on the same bot, with the same files, and to just let git handle the merging process.
  - We also started out writing the code for the different bot types as a big long messy spaghetti. It would've worked better to structure the code in a more maintainable, clearer way: have one section for analyzing the current state of the robot, then have a separate section for taking some action based on that state. We were pretty good about using methods and classes though, so that helped a lot.

Looking forward to next year's competition!

Jade Chongsathapornpong / cjade@mit.edu / certaingemstone.github.io Kai van Brunt / kav@mit.edu / tidalove.github.io Reece Yang / rya@mit.edu / reeceyang.xyz

Project Structure

- `README.md` This file.
- `build.gradle` The Gradle build file used to build and run players.
- `src/` Player source code.
- `test/` Player test code.
- `client/` Contains the client. The proper executable can be found in this folder (don't move this!)
- `build/` Contains compiled player code and other artifacts of the build process. Can be safely ignored.
- `matches/` The output folder for match files.
- `maps/` The default folder for custom maps.
- `gradlew`, `gradlew.bat` The Unix (OS X/Linux) and Windows versions, respectively, of the Gradle wrapper. These are nifty scripts that you can execute in a terminal to run the Gradle build tasks of this project. If you aren't planning to do command line development, these can be safely ignored.
- `gradle/` Contains files used by the Gradle wrapper scripts. Can be safely ignored.