MFF1 – Battlecode Postmortem

Me (Benjamin), Dan, Patrik and Martin took part in Cambridge's first Battlecode tournament, and ended up 7th / 8th. This short document at least aims to describe the core of our "strategy".

A bit of history

The very first version of our bot was written by me, which I called `basic`. It spawned four bots, one in each cardinal direction. It built harvesters on top of each ore it found, and connected them to the core. It planned all belt connections at once, by greedily connecting the first ore to the core, the second to the existing belts, and so on. It then built them, greedily navigating toward the nearest thing to build. I also taught it to rush the enemy core and break a belt by self-destructing, or, if possible, build a turret in front of a disconnected belt. We quickly shot to the top of the leaderboard, which was very motivating. Unfortunately, other teams soon started to actually develop their bots.

I then refactored my code to be easier to extend, calling the new bot `neobasic`. We developed some strategies together and went to work on implementing them. During the duration of the competition, we ended up take turns working on the bot. During the first week, I wrote the vast majority of all code, but then I got tired of it and Dan took over, then Martin, and so on. Others would occasionally submit a commit or two too, of course. We never started from scratch, evolving `neobasic` bit by bit. I'm not sure if there is any code left over from `basic` in our finals bot, but it can't be much.

We started off committing directly to `master`, creating PRs only for larger changes, but by the end, almost all changes had PRs.

Our macro strategy

We didn't have one, really.

Each bot has a state, which represents its current goal. We ended up having nine states in total:

- `EXPLORE` — Walk in an assigned direction, looking for ore.
- `PATROL` — Walk along our belts, trying to defend.
- `CONNECT` — Build an already planned conveyor.
- `BUILD_HARVESTER` — Build walls around an ore, plan a conveyor to the core, and finally build the harvester.
- `LOCATE_ENEMY` — Walk towards a possible enemy core location.
- `ATTACK` — Destroy enemy belts, but turrets in front of belts.
- `ATTACK_EXPLORE` — Look for and ore from which to launch a supplied attack on the enemy core.
- `LAUNCHER_DEFENSE` — Build defensive launchers near our belts.
- `GUARD` — Walk in a circle around our core, healing it and doing high-priority defense.

The bots don't really coordinate their states with each other. The most important choice about what to do is made at random — after connecting the first conveyor to the core, a bot will randomly choose whether to connect another ore, go attach the enemy, or go defend our base.

```
def state_after_connect(self) -> State:
    assert PATROL_PROBABILITY + EXPLORE_PROBABILITY + LOCATE_ENEMY_PROBABILITY == 1
    if self.patrol_lock:
        return State.PATROL

    p = random.random()
    if p < PATROL_PROBABILITY:
        self.patrol_lock = True
        return State.PATROL
    elif p < PATROL_PROBABILITY + EXPLORE_PROBABILITY:
        return State.EXPLORE
    else:
        return State.LOCATE_ENEMY
```

Listing 1: Our main long-term planning function.

Switching from normal attacking to a supplied attack was also done at random, with an attacking bot having a 1% chance each turn to switch to `ATTACK_EXPLORE`.

The blueprint

For a large portion of the competition, our bots would build a defense around our core, which we called the blueprint.

Figure 1: The final version of the blueprint.

The blueprint consisted of a splitter with a wall on each side. Depending on the version, the walls would be switched for sentinels when a condition was met. We also placed a launcher in the front, to defend the splitter, once we realised how powerful launchers are. We started off building this defense on all sides of the core at the very start, but we later switched to building them only after a belt was actually routed into the splitter, to conserve resources.

Unfortunately, the blueprint ended up being a detriment in the end, possibly due to costing too many resources, so we removed it.

The Autoevaluator

As many teams noticed, we requested lots on unranked matches, day and night. This was, of course, done automatically.

I wrote a script very early into the competition that would use the CLI to request a match against the top 10 teams, excluding us, and display the results. I used this to test some strategies, since testing against our own bot had massive downsides — we frequently tried to counter strategies that we never implemented our self, which is hard to test locally. Later, Dan vibecoded a small Python app that would set one of several versions as active and request a couple of matches, all in a loop, and display the winrate of each version.

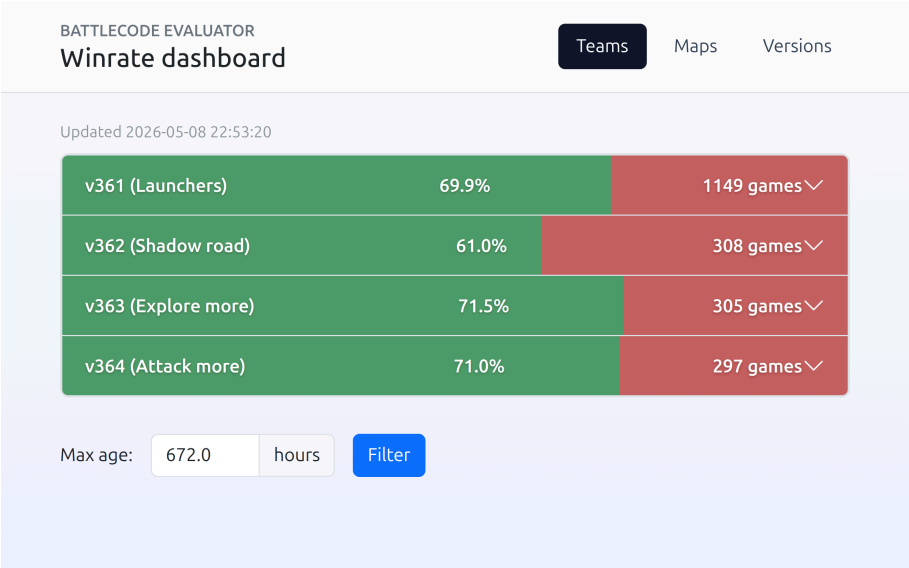

Figure 2: The final version of the autoevaluator.

I ended up hosting this app on my VPS and maintaining it for the rest of the competition. When I first started it up and went to sleep, I half expected to wake up to a message from the organisers telling us to stop, but that never happened.

We went through several strategies for how to choose which matches to pick. The final version ended up pretty simple:

- For every team on the radar:
  - Pick 5 random maps.
  - For every version under test:
    - Request a match for that team, version, and maps. Pin the version of the enemy bot to the first match requested.

This strategy was chosen to minimize noise, as it's as fair to the different versions as possible. For most of the contest, we requested matches against the top 10 teams, excluding us, but at the end we hardcoded a list of all international finalists and a couple of promising UK finalists.

Once we got the autoevaluator running, we ended up testing (nearly) all PRs using it for a couple of hours. Near the end of the contest, we realised we didn't have time to test everything and had to switch to testing only "dangerous" strategy changes. For a lot of changes, the change in winrate ended up being only one or two percentage points, which was probably not statistically significant. The autoevaluator was definitely useful, however, it helped us tune constants, validated changes in strategy (such as removing the blueprint, or building walls around harvesters), as well as catching some regressions.

The organisers ended up slashing the unranked match rate limit twice. The limit started off being 10 matches every 5 minutes and ended up at 5 matches every 10 minutes. Several teams discovered that the limit was counted on a per-user basis, instead of per-team. After the second reduction, we got the idea to pool our API keys, effectively multiplying our testing throughput by four. I really expected this to get banned by the organisers, so I wrote a message to Oli. Surprisingly, he didn't mind, so we implemented it.

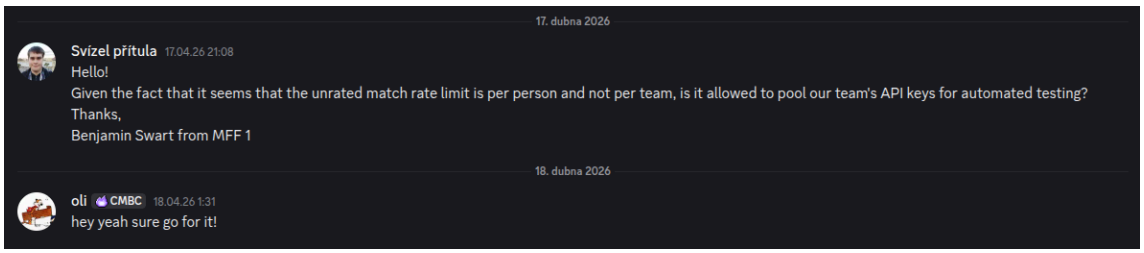

Figure 3: Approval from Oli.

Implementation

We put a lot of thought into our code architecture. Nevertheless, the codebase ended up becoming a bit of a mess.

One very nice thing we ended up doing was creating separate classes for every unit type. The main `Player` class created an instance of the correct class during the first turn and forwarded `run` calls to it. All classes had a common ancestor, `AutonomousEntity`, where we had some shared code.

Unsurprisingly, the most complex class by far ended up being `Bot`, for builder bots. Every builder bot maintains a state, and depending on its state, it calls a sequence of `try_` methods, in order of priority. Every method computes if it can and should do a given action, and if the answer is yes, it makes steps towards completing that action. All action methods have a decorator, which will stop them from running if the move/action cooldown is nonzero.

```
        if self.state == State.EXPLORE:
            self.try_build_blueprint()
            self.try_heal_neighbor()
            self.try_move_heal()
            self.try_repair_conveyors()
            if self.coroutines:
                return
            self.try_claim_ore()
            if self.coroutines:
                return
            self.try_build_harvester_turret()
            self.try_explore()

        if self.state == State.PATROL:
            self.try_build_blueprint()
            self.try_heal_neighbor()
            self.try_defend()
            self.try_move_heal()
            self.try_repair_conveyors()
            ...
```

Listing 2: A section of `Bot.run`.

```
@needs("move")
def try_avoid_gunners(self) -> None:
    dangerous_pos = self.get_dangerous_positions()

    my_pos = self.ct.get_position()
    if my_pos not in dangerous_pos:
        return

    dir = random.choice(DIRECTIONS)
    for _ in range(len(DIRECTIONS)):
        pos = my_pos.add(dir)
        if (
            pos not in dangerous_pos
            and self.map.get(pos, WALL_TILE).is_passable()
            and self.ct.get_tile_builder_bot_id(pos) is None
        ):
            self.move_towards([pos])
            return
        dir = dir.rotate_left()
```

Listing 3: An action method that attempts to avoid gunner fire.

While we tried to split larger bits of logic into separate classes, the main `bot.py` file still ended up having 2009 lines.

The map

Once nice helper class we created is the `Map`. The map stores the last seen environment and entity for every tile in the game. It also tracks which tiles are supplied by conveyors/harvesters, what resources a conveyor carries and in what quantity, and what map symmetries are still possible.

Every time a tile is refreshed, a callback called `on_chart` also gets called. This allows bots and other units to maintain sets of interesting tiles, such as exposed ores. However, we only ended up making use of this feature occasionally, as many actions simply scan the vision range for interesting tiles. This is probably a mistake, as this can cause bots to oscillate, if a tile exits the vision range due to the bot navigating around and obstacle.

```
def on_chart(self, pos: Position, tile: MapTile) -> None:
    if tile.env is Environment.ORE_AXIONITE or tile.env is Environment.ORE_TITANIUM:
        if pos in self.blueprint:
            # Avoid conflicts with blueprint
            exposed = False
        elif self.ct.get_tile_builder_bot_id(pos) not in (None, self.ct.get_id()):
            # A tile with another bot on it
            exposed = False
    ...
    if exposed:
        if tile.env is Environment.ORE_TITANIUM:
            self.exposed_titanium.add(pos)
        else:
            self.exposed_axionite.add(pos)
    ...
```

Listing 4: Sections from `on_chart`.

The pathfinder

Another interesting helper class is `Pathfinder`. We combined two pathfinding approaches. First of all, we had an A* implementation. We split the runtime of A* across multiple turns, as it's somewhat slow. We also cache the result unless something important changes. While A* is still running, our bot switches to a fast BugNav implementation instead.

The beltfinder

This is not a class, but a rather long method. The algorithm used to plan belts is nothing special, just a variant of BFS with multiple queues to implement weights. What is interesting is the way it's ran. The algorithm is implemented as a coroutine, which is executed across several turns by our coroutine system. The method is a generator function, and the runner requests elements until we approach the time limit, when it stops.

```
def continue_coroutines(self) -> bool:
    while self.coroutines:
        while True:
            try:
                next(self.coroutines[0])
            except StopIteration:
                break

            if self.ct.get_cpu_time_elapsed() > HURRY_UP_TIME:
                return False

        self.coroutines.pop(0)

    return True
```

Listing 5: The coroutine runner.

We didn't end up using the coroutine system for much else, with the exception of planning launcher.

Where did it go wrong?

Based on what I described, our codebase probably sounds pretty good. Unfortunately, the reality is quite different. There are many places where one action makes assumptions about what other actions can and cannot do, and any violation of these assumptions can cause unpredictable behaviour. For example, one action might destroy a wall to allow a bot to walk through it, while another will then panic and try to repair the wall.

Several functions also ended up being very long, with `try_connect_conveyors`, which handles building conveyors, detecting and avoiding obstacles, and building turrets to destroy said obstacles, being 149 lines of complicated `if` statements.

Ultimately, I feel like we made the mistake of writing many helpers, but not any abstractions. Not that we couldn't have used more helpers, our code is full of complex, very similar `if` statements checking various things on the map. Many helpers, such as `is_passable`, `can_build_on` or `can_build_on_with_destruction`, had unclear names, so it wasn't clear to everyone what they did. This caused people to sometimes use a helper that did something else than they expected, near the end of the contest, we discovered that some functions expected `can_build_on` to return `True` for tiles with our buildings, while other assumed it would return `False`. People also possibly just didn't know about some helpers.

Conclusion

It was a lot of work, but also a lot of fun. I'm looking forward to next year!

Figure 4: Our rating graph.