Cambridge Battlecode Postmortem

Team Make Fire

Ismail Fateen

2026-05-22

Contents

—

Introduction

I, Ismail Fateen, am writing this postmortem to help future Battlecode participants. My team had no experience with Battlecode or similar competitions, and we were only a team of two. Regardless, we almost qualified to the finals in Cambridge. I hope this postmortem shows people how easy it is to *actually* get started and improve rapidly.

Our Team

I, Ismail Fateen, am a competitive programmer (IOI 2024 participant and will go to ICPC WF 2026 in Dubai). I currently study Computer Science at the Arab Academy for Science, Technology, and Maritime Transport. My teammate, Ahmed Emara, is also a competitive programmer (EOI Silver Medalist). He studies Computer Engineering at Ain Shams University. We know each other from the EOI community.

Ahmed competed in MIT Battlecode 2026, but he was alone and therefore had no motivation to seriously put in the effort. I wanted to as well, I gathered a team of former IOIers (medalists) but they were not motivated enough, so we didn't actually get to do anything. After this experience, I decided to look for people who are actually interested, not just skilled.

The story behind the team name is pretty lame, but I find the name cool regardless. Initially, we had a team of 4 — Mina Ragy, Ismail Fateen, Ahmed Emara, and Karim Mohammed. Our initials is then MRIFAEKM. Make Fire has edit distance 1 with Make Firm — one of the permutations of our initials. Obviously, 2 of the members left the team and the initials don't add up anymore.

Game Overview

In this version of Battlecode, the objective of the game is to destroy your enemy's core. After 2000 rounds, if neither team succeeds at doing so, the winner is determined by tiebreakers — relating to the amount of resources collected by each team.

The map consists of 3 types of cells: empty cells, walls and ores. To gather resources, teams should build harvesters on ores and then build a network of conveyors/bridges to allow the harvesters to produce resources that would reach the team's core and be part of the team's global resources.

For the uninformed readers, I would suggest [reading the documentation] (https://docs.battlecode.cam/spec/overview) for more details about the game, as it does a much better job than I would.

Important note: in several places, I over-simplified what our bot does (i.e. if you check the code it will not be exactly the same). This is to prevent going into too much detail. I made sure the general idea is equivalent to our code though.

Strategy & Implementation

Sprint 1

2.1.1. Pathfinding

For pathfinding, we used Bug2, a well-known bug navigation algorithm. We copied the implementation from 4Musketeer's 2023 Battlecode (we just asked Claude to translate to Python with the cambc API). Unfortunately, this is the end of our Pathfinding throughout the entire competition. We relied on Bug2 and never changed it.

2.1.2. Bot Types

In Battlecode, it is important to assign roles to bots (so you don't focus on economy **too much**, or focus on offense **too much**). At the time of sprint 1, 'suicide bombing' was the meta. When builder bots `self_destruct`, they would do 20 damage to the tile they were on, essentially destroying it if it's

a road or a conveyor. Builder bots were also relatively cheap, so spamming them was not difficult.

We had 2 types of bots. “Explorer” bots and “Attacker” bots.

Our core would spawn bots whenever we had enough titanium to spawn 8 bots (essentially making them more expensive than they are). Builder bots assign themselves roles in the following way:

```
def get_batch1(self, ct: Controller):
    hw = ct.get_map_height() * ct.get_map_width()
    if hw <= 1200:
        return 200
    elif hw <= 2500:
        return 400
    return 500

def get_batch2(self, ct: Controller):
    hw = ct.get_map_height() * ct.get_map_width()
    if hw <= 1200:
        return 800
    elif hw <= 2500:
        return 1000
    return 1200

if ct.get_current_round() <= self.get_batch1(ct):
    self.BOT_TYPE = "EXPLORER"
elif ct.get_current_round() <= self.get_batch2(ct):
    r = random.randrange(1, 100)
    if r <= 50:
        self.BOT_TYPE = "EXPLORER"
    else:
        self.BOT_TYPE = "ATTACKER"
else:
    r = random.randrange(1, 100)
    if r <= 80:
        self.BOT_TYPE = "ATTACKER"
    else:
        self.BOT_TYPE = "EXPLORER"
```

Essentially, we have 3 different phases. In the first one, we always spawn explorer bots. Then it’s a 50% chance for each, then 80% chance attacker and 20% chance explorer. `get_batch1` and `get_batch2` determine at which round number the probabilities change. We thought that this should be changed

depending on the map size: in small maps, it's easier to gather resources so we can attack earlier.

These magic constants are mostly arbitrary with very little testing, so I wouldn't say this is a brilliant idea :)

2.1.3. Explorer Bots

The only purpose of explorer bots is to bring us titanium.

They BugNav towards a random location on the map, and when they arrive they repeat on a new random location. Once they see an ore, they override the target to be the ore location instead.

When they are adjacent to an ore, they build a harvester there and start `going_to_core`. When they enter this state, they essentially BugNav to our core location, but with a twist.

We have another variable, `bridge_waypoint`, initially equals None. This stores the bridge target of the last bridge we built. What we each turn is:

- If `bridge_waypoint` is None or we reached it, we find the 'best' bridge target and build a bridge to it. Here 'best' means closest to our core. We also set `bridge_waypoint` to be the target we built towards.
- Otherwise, we BugNav to the `bridge_waypoint`

At the time, bridges were strictly better than conveyors and clearly needed a nerf. So we relied on them and never built conveyors.

2.1.4. Attacker Bots

Attacker bots were even simpler, they move to a random location and if they find themselves standing on an enemy tile, they `self_destruct`.

However, their definition of "random location" was slightly different than explorer bots:

```
r = random.randrange(1, 100)
if r <= 30:
    return Position(random.randrange(0, ct.get_map_width() - 1),
                    random.randrange(0, ct.get_map_height() - 1))
elif r <= 50:
    return get_vertical_symmetry(ct, self.CORE_LOCATION)
elif r <= 70:
    return get_horizontal_symmetry(ct, self.CORE_LOCATION)
else:
    return get_rotational_symmetry(ct, self.CORE_LOCATION)
```

Since the map is symmetric, it's easy to figure out potential enemy core locations. The "random location" returns one of potential enemy core locations with probability 70%, and otherwise returns a purely random location.

2.1.5. Results

This bot actually reached the Round of 16! We lost 3-2 to `Clankers` there.

Sprint 2

In Sprint 1, our code was written in one file which was quite annoying, this is where we started splitting into multiple files.

2.2.1. "Ore Fucker"

A very annoying meta of the time, "ore fuckers" are bots that go around and build barriers on ores, making it harder for the opponent to harvest them. Our implementation for this was also quite simple: similar to attacker bots, they just BugNav to one of the possible enemy cores and build barriers whenever they see that they can!

2.2.2. Deterministic Bot Roles

From the sprint 1 bot, we would determine the bot's role randomly. This is not good because we can easily be unlucky and over-commit to one specific role. This is where a new idea comes to mind.

Since we are able to control at which position each bot is spawned, we can make the core decide the role of the bots and spawn them in positions depending on their role! For example: we could choose that if we spawn south of the core's center, then we should become an Explorer bot.

We had these helper constants:

```
BOT_ORDER = ["EXPLORER", "ATTACKER", "ORE_FUCKER", "ORE_FUCKER",
"EXPLORER", "ATTACKER", "ATTACKER",
             "EXPLORER", "ORE_FUCKER"]
BOT_DIRECTION = {
    "EXPLORER": Direction.SOUTH,
    "ATTACKER": Direction.SOUTHEAST,
    "ORE_FUCKER": Direction.NORTHWEST,
    "BASE_REPAIRER": Direction.CENTRE
}
DIRECTION_BOT = {
    Direction.SOUTH: "EXPLORER",
    Direction.SOUTHEAST: "ATTACKER",
```

```
    Direction.NORTHWEST: "ORE_FUCKER",
    Direction.CENTRE: "BASE_REPAIRER"
}
```

This bot order was mostly arbitrary and barely tested. After spawning these 9 bots, we repeat from the beginning again - like a cycle.

In Sprint 1, we would spawn builder bots whenever we can afford 8 of them. This was slightly changed in Sprint 2:

```
def _can_afford_spawn(self, ct: Controller) -> bool:
    titanium, _ = ct.get_global_resources()
    if ct.get_current_round() <= 500:
        return titanium >= ct.get_harvester_cost()[0] * 6 and
ct.get_unit_count() < 40
    return titanium >= ct.get_harvester_cost()[0] * 3 and
ct.get_unit_count() < 45
```

In the first 500 rounds we make builder bots more expensive so they can afford to build harvesters and bridges back to the core. We thought that `harvester_cost` is a better indicator because they must be able to build harvesters eventually.

One of the balance changes in sprint 2 was that you could have at most 50 units at any given time, therefore the second condition was added to ensure that we still have space for Launchers/Turrets

2.2.3. Launchers

Launcher defense was popular in this meta because you only need to defend a few bridges.

We added launcher building to two places:

- Before building a bridge, see if we can afford to build 6 launchers. If we can, then we build one launcher adjacent (possibly diagonally) to the bridge position.
- When we see a harvester and can afford to build 6 launchers, we build a launcher adjacent to the harvester.

The launcher logic itself is simple:

```
def run(self, ct: Controller) -> None:
    if not self.ran_before:
        self.init(ct)

    enemy_target = self._find_enemy_bot(ct)
    if enemy_target is None:
```

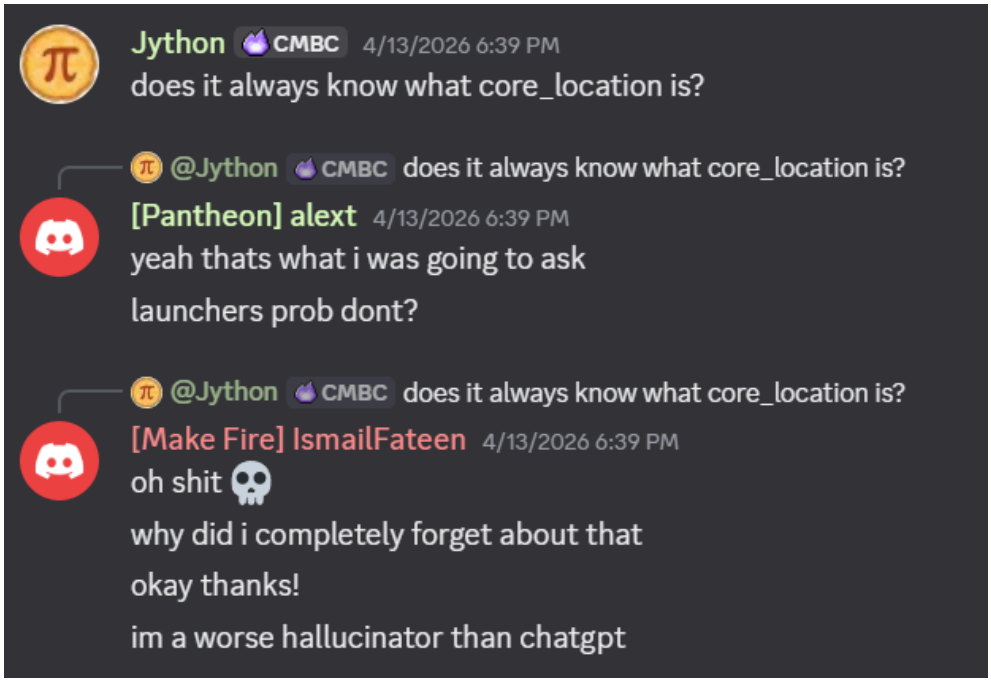
```
        return

    best_land = self._best_launch_landing(ct, enemy_target)
    if best_land is not None:
        ct.launch(enemy_target, best_land)

def _find_enemy_bot(self, ct: Controller):
    for id in ct.get_nearby_entities(GameConstants.ACTION_RADIUS_SQ):
        if ct.get_entity_type(id) == EntityType.BUILDER_BOT and
ct.get_team(id) != ct.get_team():
            return ct.get_position(id)
    return None

def _best_launch_landing(self, ct: Controller, enemy_pos):
    best_pos = None
    best_dist = 0
    sym = self.CORE_LOCATION
    for pos in ct.get_nearby_tiles():
        if ct.can_launch(enemy_pos, pos):
            d = pos.distance_squared(sym)
            if d > best_dist:
                best_dist = d
                best_pos = pos
    return best_pos
```

Find an enemy bot, throw it to the cell farthest from our core. While this sounds simple and effective, there is a *huge* problem that was only discovered on April 13, and only after I revealed my code to the public to ask for help.



2.2.4. Attacker Bot Rework

After sprint 1, `self_destruct` was nerfed. Therefore, 'suicide bombing' was not a valid idea anymore, and attacking had to be reworked.

Attacker bots now only choose the random target location to be one of the 3 potential enemy core locations (rather than trying a completely random location with probability 30%). At this point, people did not have good defense yet, so it was wise to make attacker bots go for the core directly and be aggressive. This strategy actually got us to our peak rank (2nd place) briefly, just because of how difficult it was to defend.

Note that:

- once we are close enough to the chosen potential enemy core location, we re-generate a new one if it's not the correct one.
- once we determine the enemy core location, the "potential enemy core location function" always returns the correct enemy core location.

This allowed us to identify the enemy core relatively quickly.

Whenever we are close enough to the enemy core, we look for conveyors to attack. (Here, close enough means "if we place a sentinel here, it can target the enemy core"). We look for conveyors (or bridges) that have a resource on them (using `get_stored_resource`). Once we find one that we can attack, we enter the "attack conveyor" state.

Our "attack conveyor" state is quite simple and self-explanatory. The code will do a better job:

```
def _handle_attack(self, ct: Controller) -> None:
    if self.attacking_target == ct.get_position():
        if ct.can_destroy(ct.get_position()):
            ct.destroy(ct.get_position())
        if ct.can_fire(self.attacking_target):
            ct.fire(self.attacking_target)
            return
        enemy_direction =
self.attacking_target.direction_to(self.ENEMY_CORE)
        for direction in DIRECTIONS:
            if ct.can_move(direction):
                ct.move(direction)
                break
        if ct.can_build_sentinel(self.attacking_target,
enemy_direction):
            ct.build_sentinel(self.attacking_target, enemy_direction)
        elif ct.get_sentinel_cost()[0] <= ct.get_global_resources()
[0]:
            self.attacking_target = None
    else:
        self.nav.move_to(self.attacking_target)
```

Here, `self.attacking_target` is the conveyor position we chose (which may even be a friendly conveyor! as long as it has titanium and is close enough to enemy core, it's good), and `self.ENEMY_CORE` is obviously the enemy core.

If we haven't arrived on the conveyor yet, we go to it. If we have, then we attack it: after sprint 1, `self_destruct` was replaced with builder bots having the ability to `fire` on a tile they're on, dealing 2 HP.

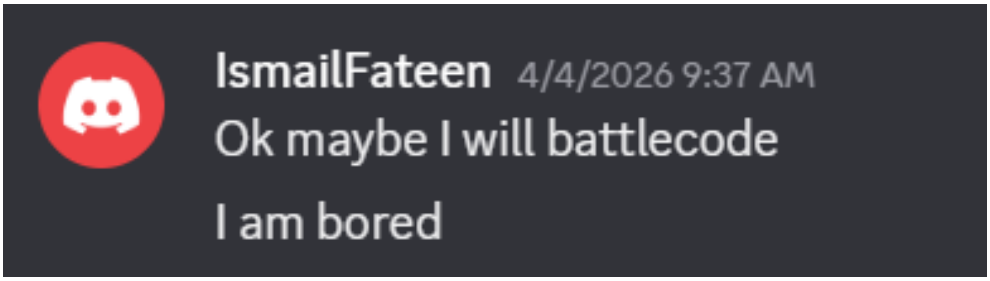
Once the conveyor is destroyed (either using `destroy` if it's a friendly conveyor, or when we cannot fire anymore because the building is gone), we attempt to move away from it so we can build a sentinel. If we cannot afford a sentinel, we just give up (proceeding to find another target, where we go at a later time with hopefully more titanium).

2.2.5. Results

Again, we managed to reach Round of 16 in sprint 2, where `Lorem Ipsum` beat us 3-2. ~~flex: Lorem Ipsum was actually 2nd place in this sprint, and we almost beat them~~

Sprint 3

Between sprint 2 and sprint 3, I (Ismail Fateen) thought that I should quit Battlecode and focus on other things (uni, EOI Scientific Committee work). Little did I know, I was already hooked.. I did in fact quit for less than 24 hours, but then I came back.



I decided to rewrite the bot yet again, to give up some of the ideas I thought were bad and have a clearer structure.

The final bot for this sprint was bad, but it built a good foundation for what's coming next.

2.3.1. Better State Management

Firstly, we gave up "Ore Fucker" because people started to do well against it. We renamed "Explorer" to "Economy". Previously, we used variables like `self.is_going_to_core` to track state: this is terrible. It is very easy to create bugs using this approach, and as we've learned in this process: ~~battlecode is mostly about having the least amount of bugs~~ bugs are terrible.

Instead, we used Python enums to track state, and each enum calls its own function.

```
class EconomyPhase(Enum):
    SEEK_ORE = auto() # if this is the current phase, call
run_seek_ore
    GO_ORE = auto() # run_go_ore
    BUILD_HARVESTER = auto() # run_build_harvester
    CONNECT_TO_CORE = auto() # run_connect_to_core

class AttackPhase(Enum):
    FIND_TARGET = auto() # run_find_target
    GO_TARGET = auto() # run_go_target
```

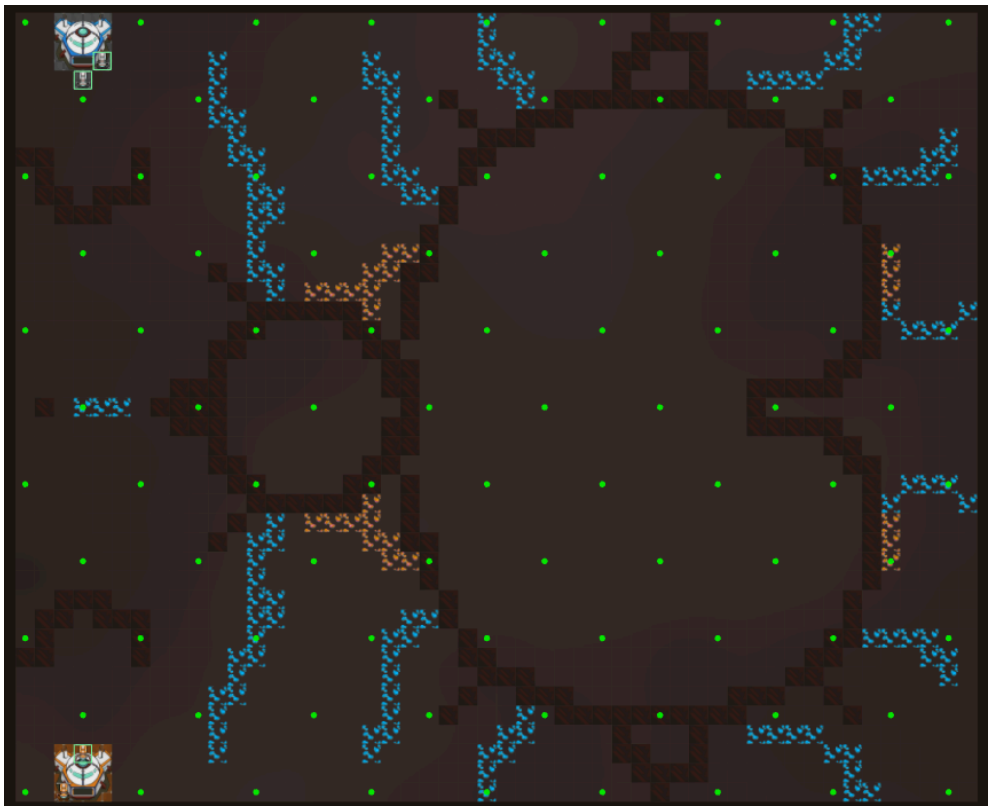
2.3.2. SEEK_ORE

"How do we find nearby ores efficiently?" is a problem of utmost importance in this game.

Random movement is bad in sparse maps. Spawning 4-8 bots for each direction is expensive.

This is where my first (and probably last) non-trivial idea comes to play. Ever heard of a spiral? I wanted to implement something similar to this.

Consider this set of points:



If we managed to visit just these points, we would be done! We will have seen the entire map (vision radius covers all the unvisited points).

```
def get_waypoints(ct: Controller, radius: int) -> list[Position]:
    waypoints = []
    dx = radius * 2
    dy = int(1.5 * radius)
    n = ct.get_map_width()
    m = ct.get_map_height()

    for row, y in enumerate(range(0, m, dy)):
        offset = 0 if row % 2 == 0 else radius  = stagger
```

```
        for x in range(offset, n, dx):
            if is_in_bounds(ct, Position(x, y)):
                waypoints.append(Position(x, y))

    return waypoints
```

This function generates such set of points, we used $\text{radius} = 3$, though it could've probably been increased and it would've been better with full coverage still. The code is fairly self-explanatory.

Awesome, now in what order do we visit them? If we visit it from top-left to bottom-right, that's not similar to a spiral is it?

Now, the purpose of this postmortem is to show how easy it is to get started on such competitions, so I'll be brutally honest: we just asked ChatGPT (btw I promise it was not used for the writing of this postmortem).

Apparently, sorting with this key is good enough:

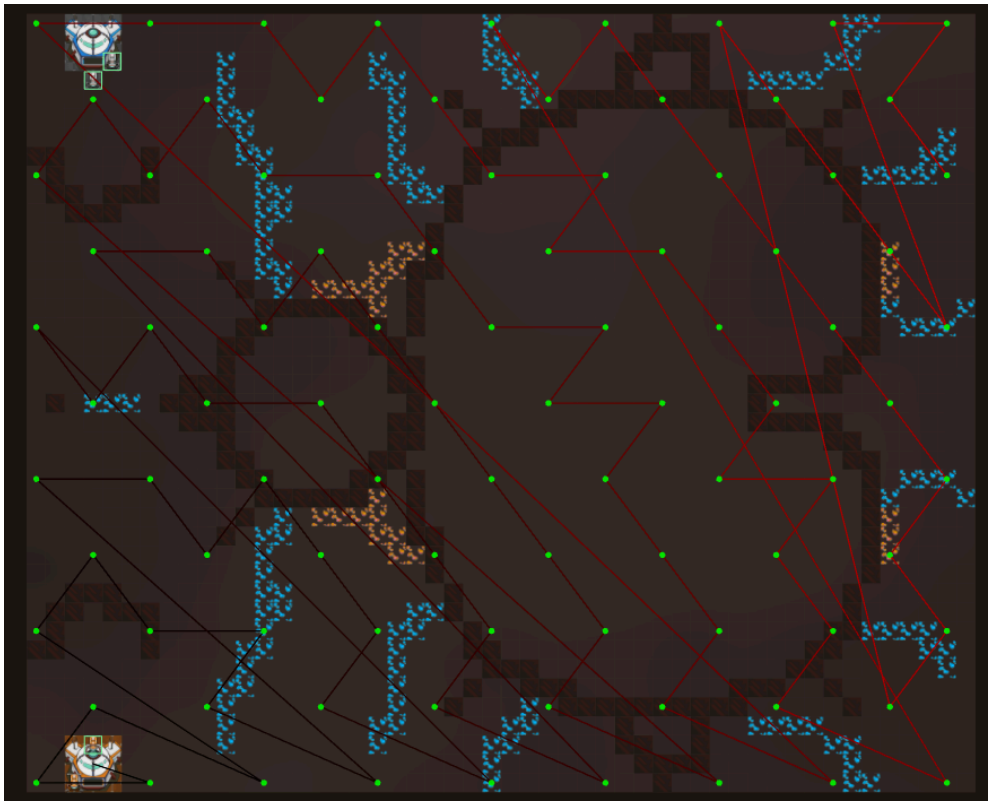
```
def key(p: Position):
    dx = p.x - self.CORE_LOCATION.x
    dy = p.y - self.CORE_LOCATION.y

    dist = math.hypot(dx, dy)
    ring = int(dist // (2 * 3))  = which "layer"

    angle = math.atan2(dy, dx)  = -pi → pi
    if angle < 0:
        angle += 2 * math.pi  = normalize to 0 → 2pi

    return ring, angle
```

This is how the order looks like on the same map above (for team gold, we start at its core then go the point to its right and so on):



While this is not as good as we wanted, this definitely gives us a behavior similar to what we want.

Now `SEEK_ORE` does the following:

- BugNav to the next waypoint (if we visited all waypoints, unlikely, BugNav to a random location instead)
- Look at all tiles in vision radius: if one of them is a titanium ore, set the `TARGET_ORE_LOCATION` to it and switch the state to `GO_ORE`

That's pretty much it!

There's a significant problem here, any guesses?

All economy bots visit the waypoints in exactly the same order! Is there even a point to having more than one economy bot here? We fix this later.

2.3.3. GO_ORE

Simple, BugNav to `TARGET_ORE_LOCATION`. If we're stuck, go back to `GO_ORE`.

Once we arrive, we switch to `BUILD_HARVESTER`.

2.3.4. BUILD_HARVESTER

Initially, this state only happens when we are standing on an ore.

- When we are on an ore, we attempt to move away from it then build a harvester.
- Otherwise, that means we moved and tried to build a harvester but failed, so we move back to the ore location to try again next time. (I know, this is inefficient but that's what we did ¯_(ツ)_/¯)

2.3.5. CONNECT_TO_CORE

This is likely the biggest change in Sprint 3, because bridges are now nerfed so hard that conveyors are necessary if possible.

This is the only place where we **don't** use BugNav.

Our assumption is that we **are** going to build a bridge, so we first find the best target for the bridge (which is currently a naive "the cell that's closest to core and is unoccupied"). After we decide on a bridge target, we attempt to visit it via BFS, where the only moves allowed are cardinal moves (east, west, south, north). If we are able to visit it via BFS, we will then build conveyors instead of a bridge. I will leave the implementation details out of this, but that's basically the idea.

2.3.6. Attacker Bot

The bot is largely unchanged, the only change is switching to the new enum method of managing state. It has two phases: `FIND_TARGET` and `GO_TARGET`. `FIND_TARGET` is the part that would go to the enemy core and look for vulnerable conveyors, while `GO_TARGET` is the part that goes to the vulnerable conveyor and attacks it.

Sprint 4

Guess what, we rewrote our code yet again. This time, we started having TLE problems, and I thought that the import overhead was the cause. Of course, I was wrong, but by the time I discovered that it was already too late.

2.4.1. Unified Bot Roles

In sprint 4, we tried out a new approach: make all bots equivalent. The reasoning behind this is: a bot that focuses solely on economy but sees a great offensive opportunity will never use it. Overcommitting is not good.

Instead, all bots would do the same thing, and they would have these unified phases instead:

```
class Phase(Enum):
    WANDER = auto()
    GO_ORE = auto()
    PROTECT_ORE = auto()
    BUILD_HARVESTER = auto()
    CONNECT_TO_CORE = auto()
    ATTACK_CONVEYOR = auto()
    ATTACK_HARVESTER = auto()
    HEALING = auto()
```

`WANDER` is the default. From it, a bot checks, in priority order, whether it can do anything useful right now:

```
def run_wander():
    if try_heal(): return
    if try_go_ore(): return
    if try_connect_core(): return
    if try_attack(): return
    nav.move_to(WANDER_RANDOM_LOCATION)
```

Any bot near an ore harvests it. Any bot near a damaged building heals it. Any bot near a disconnected harvester routes conveyors. Any bot near an enemy harvester/conveyor attacks.

Since all bots do the same thing, we must still somewhat balance economy/defense vs offense. To do this, we change the random location generation algorithm:

- With probability $\frac{1}{3}$, we generate a location within radius 10 from our core. (Economy/Defense focused)
- With probability $\frac{1}{3}$, we generate a location within radius 10 from the enemy core (or one of the candidates if it's not determined yet) (Attack focused)
- With probability $\frac{1}{3}$, we generate a completely random location (Exploration focused)

2.4.2. PROTECT_ORE

Between `GO_ORE` and `BUILD_HARVESTER` we added a new `PROTECT_ORE` phase. When a bot arrives on an ore tile, before building the harvester it, the bot builds launchers at the four diagonal neighbours (if affordable and a launcher slot is free)

Only once there is at least one launcher adjacent, or no more launcher slots are buildable, does the bot proceed to `BUILD_HARVESTER`.

Note: until this point, launcher is still bugged and does not behave as we wanted it to (because `CORE_LOCATION` is always `Position(-1, -1)`)

2.4.3. Healing

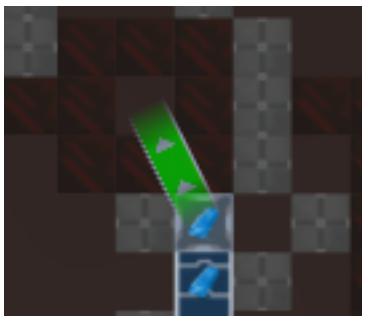
`Phase.HEALING` is straightforward — navigate to the nearest damaged friendly building and `ct.heal()` it. Any bot in `WANDER` will enter this state immediately when `try_heal()` finds a target.

Healing was necessary because we did not even implement defensive turrets, and launchers are not enough because they could not cover the entire area (and they are relatively expensive).

2.4.4. Incremental BFS

Sprint 3′s conveyor routing picked the next bridge target by raw Euclidean distance to the core. This is fine.

Until it's not.



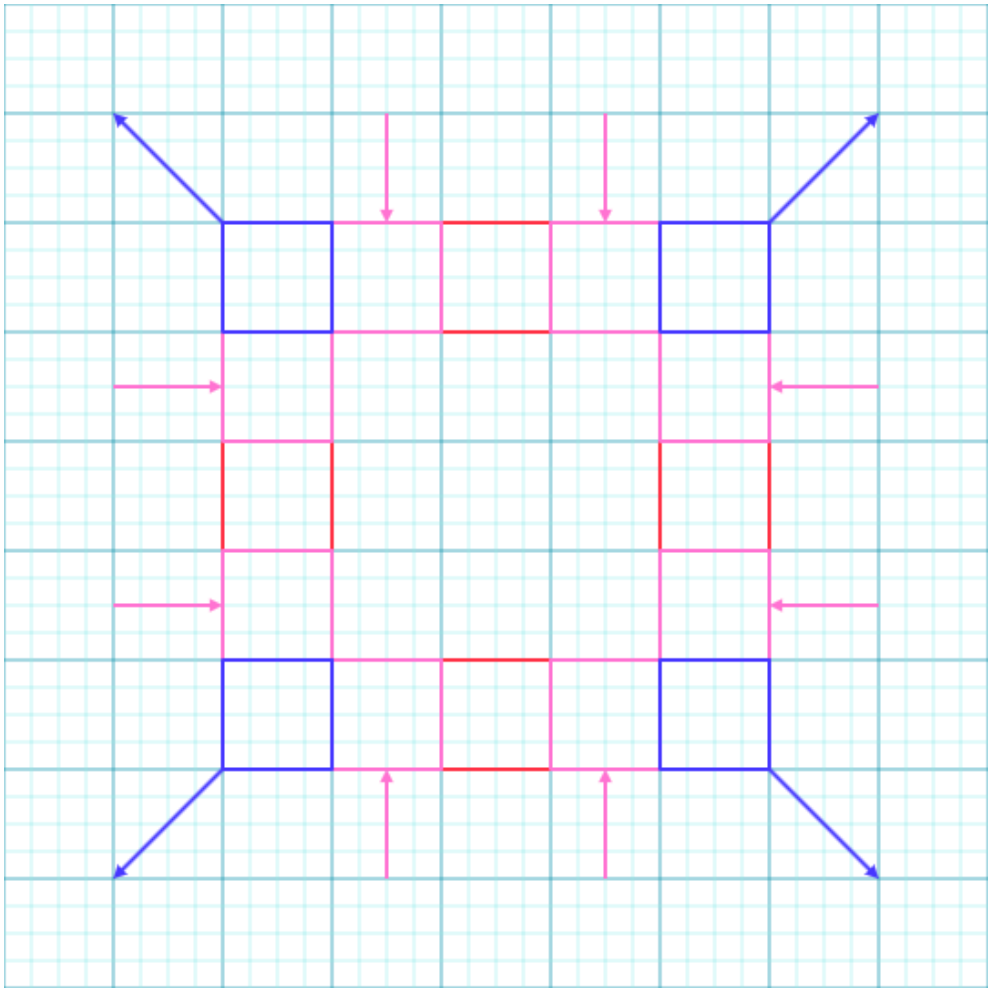
As you can see in this picture, we may build bridges to unreachable cells! Our solution to this problem is: every round, perform a BFS (Breadth-First Search) from the core, treating unseen cells as walls.

Now, instead of picking the next bridge target by raw euclidean distance to core, we pick it by the BFS distance calculated. Therefore, we only build bridges to cells we have proven are reachable before (this also protects us from building a bridge over a wall to a cell that's reachable but very far due to walls).

There's a small problem though. You seriously want to do a BFS for the entire map every single round? Even if we somehow cheese it and it doesn't TLE, it still does use our valuable time! Instead, we made it split over several rounds, processing at most 500 cells per round.

2.4.5. Axionite

At this point, we realized axionite is necessary to do well. We decided that it's too late to implement dynamic axionite (i.e. dynamic foundry positions) as it would be tricky. Instead, we built a base around our core:



- Red: Foundry
- Pink: Splitter (arrow inwards indicates input direction)
- Blue: Sentinel (arrow outwards indicates facing direction)

We have a constant, `AXIONITE_START = 350`, which means that until round 350, our bots ignore all axionite ores.

`run_wander` actually does this, not what I said before:

```
if ct.get_current_round() >= AXIONITE_START - 20 and
try_build_foundry():
    return
if try_build_splitter():
    return
if try_build_sentinel():
    return
if try_heal():
    return
```

```
if try_go_ore():
    return
if try_connect_core():
    return
if try_attack():
    return
nav.move_to(WANDER_RANDOM_LOCATION)
```

but it was important to introduce the core base idea first.

Of course, we had to change the "`find_best_bridge_target`" function to accommodate for this. It should not choose a target that's a foundry/sentinel, it should always go to splitter positions when it's close enough to the core.

2.4.6. Attacking Changes

As you saw above, attacking has two phases: `ATTACK_CONVEYOR` and `ATTACK_HARVESTER`.

Let's digest the three functions one by one: `try_attack`, `run_attack_conveyor` and `run_attack_harvester`.

2.4.6.1. `try_attack`:

- Look at enemy conveyors in our vision, if one of them is close enough to enemy core (a sentinel there would target it), we choose it as our `ATTACKING_TARGET`, and switch to `ATTACK_CONVEYOR`
- Look at enemy harvesters in our vision, if one of them is vulnerable (has a free space beside it), we change `ATTACKING_TARGET` to it and switch to `ATTACK_HARVESTER`

Okay I just checked the other two functions. What.

They are full of bugs, let's explain them in sprint 5/intl quals instead.

2.4.7. Results

Also reached Round of 16, lost 5-0 to `something else` (2nd place in finals)

Sprint 5

Sprint 5 was not a serious sprint because we made a lot of changes, and therefore had a lot of bugs. The few days between sprint 5 and the international qualifiers were spent improving on these changes, fixing some of the bugs and adding new simple strategies.

We still somehow reached top 16, but lost 4-1 to `okbro`. We didn't consider this performance indicative of our bot because we knew it was full of bugs. So let's just move to international qualifiers.

International Qualifier Round

This is a big one.

2.6.1. Hybrid Approach to Bot Types

Unified bots were doing bad. Why? Because they were still focusing too much on economy, even when we made them go to locations near the enemy's core with probability $\frac{1}{3}$. They simply see ores on their way and activate `GO_ORE`.

Unlike economy, defense and offense are always important. So we chose to have 3 types of bots instead: `EVERYTHING_DOER` (almost the same as the sprint 4 bot), `DEFENSE_FOCUSED` and `ATTACK_FOCUSED`. This allowed us to ensure we always work on defense and offense, while giving economy bots (`EVERYTHING_DOER`s) more flexibility.

2.6.2. Waypoints Fix

Remember from sprint 3, when we had the idea to make economy bots go through waypoints around the map first before going to random locations? This had a problem that having 2 economy bots would make them go around the same paths, which is inefficient.

Instead, we made it such that **only the first bot** goes through these waypoints, the other `EVERYTHING_DOER` bots go to random locations immediately.

2.6.3. Launcher Defense Fix

Remember our embarassing bug that made it to the game until April 13? We fixed it. Instead of throwing enemy bots to the farthest node from our core, we threw them to the farthest node in the direction they came from.

2.6.4. GO_ORE fix

`GO_ORE` had a significant problem. It appeared especially in maps like labyrinth. Basically, `try_go_ore` looks for an ore, if it finds one it chooses it as the target and switches to `GO_ORE` (instead of `WANDER`)

The problem is, the ore it found may be *very* far (due to walls). On our way to that ore, we may see other ores (heck, maybe even pass over them), and completely ignore them, because we locked in to that ore that's very far away.

So instead, while we are in `GO_ORE`, we call `try_go_ore` again. If we find a closer ore, we switch our target to it.

Luckily, before any problem happened, I already anticipated a very bad bug. There might be a situation where we cycle between 2 ores. In other words, we initially see an ore and choose it as our target. Then, on our way we see a closer one, so we choose it as our target. On our way to the closer one, we see the first ore we saw again, and it's closer, so we switch to it.

For this reason, I maintained a `visited_ores` set which would get cleared whenever we arrive in an ore. And whenever we switch our target ore, we add the previous one to the set to make sure we never use it again. This severely improved our performance in maps like `labyrinth` and `metropoiltan_dystopia`.

2.6.5. Back to the two attacking functions

2.6.5.1. `run_attack_conveyor`:

- If we are not on the conveyor position, move towards it
- Otherwise
  - If it's a friendly conveyor, destroy it
  - If it's an enemy conveyor, fire on it
  - If it's an empty tile (after destroying/firing enough), move away from it, then build a sentinel on the target
- Make sure we don't spend over 30 rounds attacking a single conveyor (naive way of escaping healed conveyors). Of course, if we spend over 30 rounds attacking a single conveyor, we add it to a `was_healed_before` set to make sure we don't attempt to attack the same conveyor right after, doing an infinite cycle.

2.6.5.2. `run_attack_harvester`:

- If we are not adjacent to the harvester position, move towards it
- Otherwise look at its nearby positions: empty cells first, then cells with a building of ours, then cells with a road of theirs.
- If there's an empty cell, attempt to build a gunner on it
- If there's a friendly cell, destroy it then attempt to build a gunner on it
- If we're on an enemy cell, fire on it then move away and attempt to build on it. (likely we cannot build after one time which is fine, it will repeat again)

2.6.6. `ATTACK_FOCUSED`

It's attack focused. Simply, move to the enemy core (same method as earlier, go to the potential locations until you find it).

On its way, if there's a vulnerable conveyor/harvester, set it as the target and switch to `ATTACK_CONVEYOR`/`ATTACK_HARVESTER` respectively.

2.6.7. `DEFENSE_FOCUSED`

Wait.. we never mentioned defense? How do we defend exactly, other than healing?

2.6.7.1. `try_destroy`

The most expensive function ever. Iterate through all tiles in the vision radius. If it's a conveyor/bridge, see if it feeds an enemy turret. If it does, set the target location to it and switch to `ATTACK_CONVEYOR`.

If it's a harvester and it feeds an enemy turret, set the target to it and switch to `ATTACK_HARVESTER` (this is the expensive part because we have to check for each harvester, all of its adjacent cells)

2.6.8. Results

We were 1 win away from qualifying to the finals at Cambridge. We lost in winners bracket to `muteki`, then lost to them again in the losers bracket. Yet, they're my favourite team. Funny enough, their favourite team was also bwaaa, who later knocked them out in finals. Idk what's with cambc and liking the team that eliminated you.

These were the biggest changes in each sprint. Note that none of these is exhaustive, and many sections don't go into the details. This is intended. I wanted to show that our bot is not particularly brilliant (does not use voronoi like pantheon or something :skull:). I believe that, particularly this edition of battlecode, the most important thing is to have as few bugs as possible (which is easy to do if the bot is simple).

Besides, going into detail is not very productive. You can likely get a similar version of my bot working by just prompting Claude Code properly (maybe even a better bot). If interested in the full code, I have published the code on [my GitHub](https://github.com/ismailfateen/makefire-battlecode).

If you would like to discuss anything, feel free to send me a friend request on discord `@ismailfateen` and let's talk about it!

Final Thoughts

Overall, this competition was really fun and inspiring. We will definitely be joining later editions (whether by Cambridge, MIT or any other organization).

I'd say the biggest win is the community, surely some of the coolest and strongest people I've met.

Next time, I will stick with multi-files from the beginning to have a better organized code, which will make it easier to debug later. I will also definitely be taking more notes from the games I watch (to be honest, most of the games I watched were purely just for fun, and that wasted a lot of my time with no real benefit).

I will also be making more use of AI, as it can speed things up a lot. One thing I will look into doing is creating tools for the AI to use so that it has a better feedback loop. Quick feedback makes agents really strong.