Cambridge Battlecode 2026 Postmortem

Siddhesh Umarjee, VoxelRifts
(part of) Team cheesynachos

I. Introduction

We are from Team cheesynachos which ended up in the top 12 in the International Qualifiers. This was our second battlecode, after MIT Battlecode this same year. We improved on several fronts and made new mistakes which is definitely a sign of progress! We want to share our strategies, learnings and implementations in this *report* with the motive providing a direction for first-time battlecoders along with a hidden motive of articulating our postmortem thoughts to better understand our own bot. You can find the code for our bot here.

A. What is Battlecode

Battlecode is a competitive (multi) agent programming competition. The participants have to program bots to play a 2-player turn based game. The rules of the game are announced at the start of the tournament (and change dynamically due to balance patches over the duration of the competition). As a result, success hinges on a teams ability to design and implement robust bots from scratch and react to the balance patches as well as the strategies which are being implemented by other players (*the meta*).

II. The Game

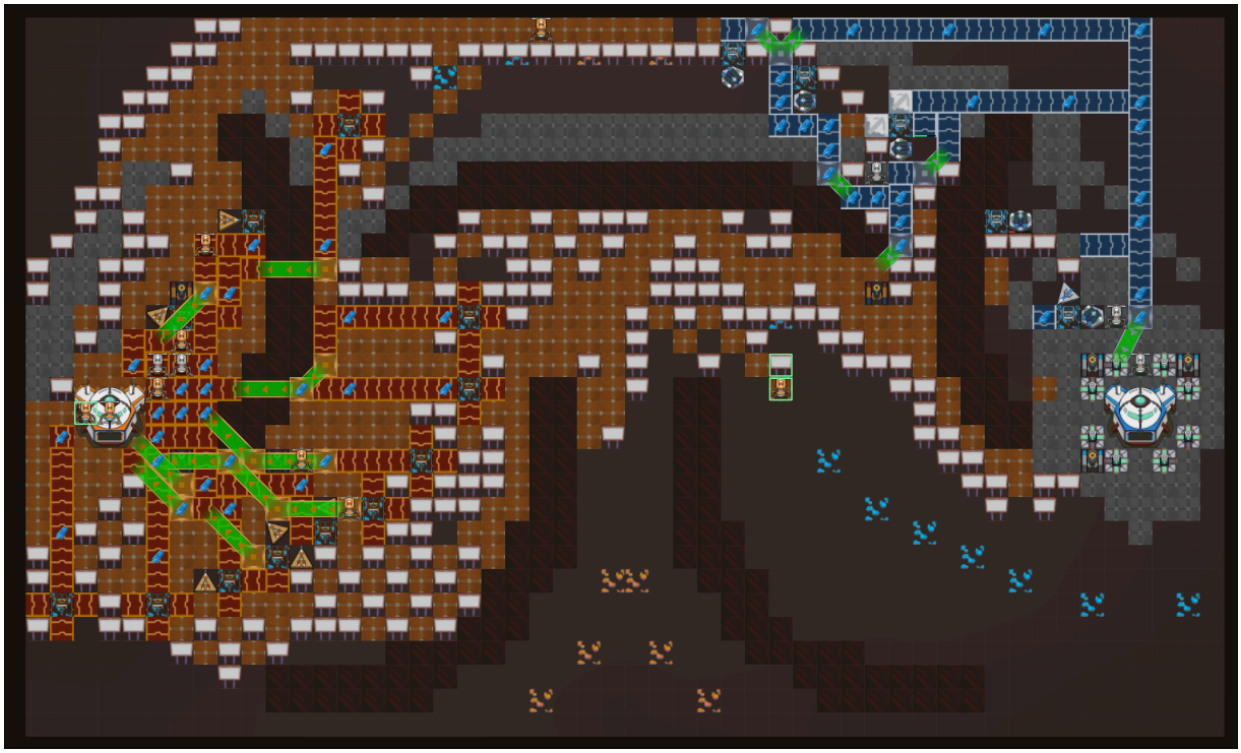

Fig. 1: A game of Cambridge Battlecode in progress

This Cambridge Battlecode was set on an astronomical surface (Saturn's moon, Titan to be exact) where an ore "Axionite" needs to harvested, processed and collected by your fleet. However another corporation is competing with you and you must either collect more resources or destroy the enemy core to win. This contrast of an offensive and defensive win conditions resulted in some interesting strategies.

A. Environments

The stuff which is out of our hands can be called the *environment* It includes all the map-related elements and as the game progresses, the enemy units and buildings. In this Battlecode event, the initial environment was defined by the location of our core and enemy core, the map size, wall placement and placement of ores. We design bots to assess the environment and take action! As play progresses the bots also *see* enemy units and buildings and must navigate that dynamic environment.

B. Resources

We aim to collect the most resources. Each action we perform incurs a cost and depletes the collected resources. In this Battlecode event, there were 2 types of resources: Titanium, which was easy to collect and was required for several, almost all actions and Axionite, which had to be *refined* before collection and hence was slightly difficult to collect. However it was necessary for certain buildings and provided greater damage when used as a ammo. It also played major role in the resource-based win condition as the side which collected more Ax by end of play won irrespective of which side had more Ti

To collect a resource, a *harvester* had to be placed on an ore of that resource and conveyor belts or belts had to placed to make a pipeline from the harvester to the core. The harvester would produce 10Ti every 4 turns, allowing possibilities of reusing the conveyor belts/bridge.

C. Units

The *units* are what we can control using code. In this Battlecode event, there were 3 types of units we could control:

a) *Core:* Its location is fixed by the *map*. We use the core (and only the core) to spawn new builder bots. Also, if our core is destroyed, we lose.

b) *Builder Bots:* It is the only mobile unit. It can construct buildings and heal one of the nearby tiles.

c) *Turrets:* These are static buildings that can fire at certain tiles. There were 4 types of turrets which were fairly well-balanced.

- Gunners are turrets with limited range, high DPS and can rotate. They were usually used close quarters attacking and disabling other turrets.
- Sentinels are turrets with a wider range with low DPS, but could fire through other buildings. They were mostly used for defence.

- Breaches are turrets which cost Axionite and have really high damage with splash. They were mostly used for… Nothing. (sorry Jimboko, they were not viable)
- Launchers are the only turrets that don't do damage (hence, don't need ammo), but can throw adjacent builder bots to a walkable tile in range. Launcher tech went through quite a few changes as the meta evolved.

D. Buildings

There were a number of buildings which could be constructed by builder bot. They included harvesters and foundries for obtaining raw ore and refining axionite respectively; two types of conveyors for transporting resources; bridges to allow conveyor cross-overs over terrain or other conveyors; roads for walking over; barriers to block tiles and markers to communicate information between bots.

III. Tooling

Learning from our mistakes in MIT Battlecode, one of our team members worked exclusively on tooling. We wrote/edited a set of python scripts that allowed us to quickly test our bots which was a real pain point in our previous battlecode attempt.

The main script was "run_tests.py" which allows you to test a certain bot against other bots on a subset of maps. All this information is set up in a config file. The multi-threaded execution was the most helpful feature, along with generation of a handy csv file which we could refer to for the outcomes of the matches. This is helpful in checking for any anomalies in bot behaviour.

As time went on, however it became clear that testing against previous versions of our own bot was *not* ideal. So we stole the auto-scrim script from Team Blue Dragon and modified it to automatically queue unranked games against teams +- 5 places of us on the ladder. The scripts for fetching match data and visualizing it as a matrix of team vs map winrates was *really nice* as well. I highly recommend people to invest some time in building such scripts for battlecode events.

Other than that, we used python's cprofile along with snakeviz when trying to optimize our bots as well. It gives a really nice view of how much time each function takes. Here's what it looked like before we optimized our pathfinding routine

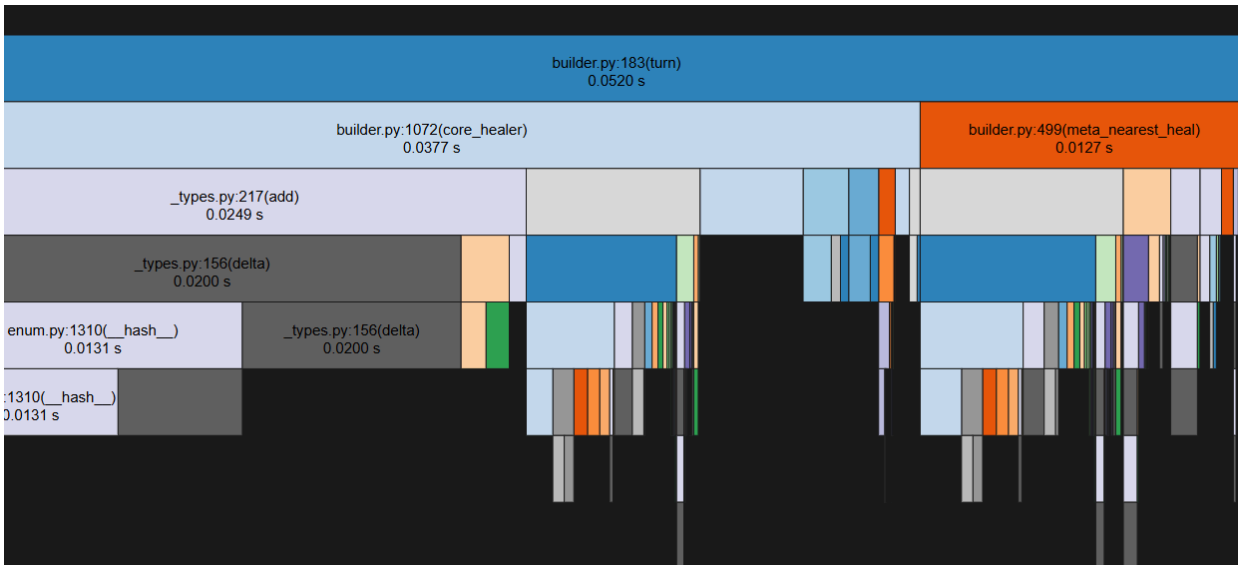

Fig. 2: Most time taken by _types.py is exclusively due to Position object allocations

IV. Pathfinding

We implemented 2 different types of pathfinding over the course of the event.

A. *Silly (Yes that's what we called it)*

Our initial pathfinding implementation was taken from our implementation for MIT Battlecode 2026, which itself was inspired by Team OmNom's pathfinding from MIT Battlecode 2025 that they mentioned in their post-mortem.

The problem with bugnav is that it is very flimsy when it comes to interactions between dynamic entities, which, most battlecode events seem to **really** care about. The strategy specified by Team OmNom is to use bugnav as a planning step, only for figuring out *where* to go in vision radius, then using some other algorithm to actually perform the pathfinding to that position.

That's basically what we did for "Silly Pathfinding". We update a virtual position based on bugnav simulating k turns into the future (or until the virtual position goes out of vision). Before Sprint-2 i.e. before it became clear implementing A* would actually be practical for this competition, we would simulate 2 turns of bugnav before trying to pathfind to it using basic heuristic based Pathfinding. Why 2 turns? Because more than that no longer guaranteed that the heuristic based Pathfinding would be able to reach the virtual position.

However, as we came to know of other teams (**cough* Blue Dragon *cough**) using A* quite successfully, we changed the heuristic based movement routine to an optimized A*

B. AStar

After Sprint-2 we migrated from our original Silly Pathfinding strategy to an optimistic A*. Usually, there are two major issues with implementing A* in a Battlecode bot, but they didn't cause problems this time.

1) **It is usually too inefficient to compute within a turn.**
   This was a big deal during MIT Battlecode since the bot performance there is measured in "amount of Java Bytecode executed". However, Cambridge Battlecode used execution time as a measure, and we had 2 full ms to work with.
   Another important thing to consider is that A* only takes longer than 2ms to compute if you're doing pathfinding over a long distance on the map, which is fairly rare.

2) **The best pathfinding relies on the bot knowing the whole map, however bots can only see within a vision radius.**
   This is quite simple to get around. You can just optimistically assume that unseen tiles are actually passable, and run A*. If the bot moves and it finds an "assumed passable" tile is actually a wall, it can run A* again. Over time, as the bot accumulates information about the map, the amount of A* recomputations will drop.

Our first implementation of A* broke almost 30% of our main bot behaviour, even though it was technically more robust than Silly Pathfinding. The problem was that the new system made the bots not move until a valid path to the goal had been found. This was different from our old Silly Pathfinding implementation which still moved in the general direction of the target, even if the exact target position was unreachable.

Over time, we had unknowingly relied on that behavior. For example, our rush bots tried to pathfind directly to the enemy core, which is actually unwalkable. With the Silly algorithm, they would still advance toward the enemy base. With A*, they failed to find a valid path and simply stood still while the enemy destroyed our core. We fixed those issues by re-adding Silly Pathfinding and using it in a few cases where we actually wanted that behaviour.

After Sprint-3, we started to exceed the 2ms limit on bot execution, so we reimplemented the A* algorithm but this time using the optimization tips that were provided by @randomuserhi on the Discord Server.

1) **Eliminating ALL allocations in hot paths**
   The largest optimization was removing almost all Position object creation from the pathfinding loop. Instead of allocating coordinate objects for every node expansion, the algorithm operated entirely on tile indices, significantly reducing CPython allocation and garbage collection overhead.

2) **Introduced global Position object caching**
   Since parts of the engine API still required Position objects, we preallocate reusable instances for **every coordinate** in a global cache. So, converting from the linear tile indices to Position objects is an allocation-free simple list lookup.

3) **Removed object field lookups**
In CPython, dynamic attribute lookup is costly, so frequently used values were flattened into locals or arrays to keep execution on fast integer and list operations.

Using the above strategies, our bots using >2ms per turn, were reduced to using ~1200µs which was very helpful for improving our micro and macro decision-making.

V. Conveyor Network

The conveyor network was at the heart of this Battlecode event and so it was one of the first things we worked on. The core of the problem is knowing when to merge lines and when not to. A direct conveyor/bridge line to the core becomes saturated at four harvesters, so merging too many resource streams onto the same route significantly reduces throughput. If the bots had perfect information about the network, this problem is easy to solve with Max-Flow (even if not super fast). But this is Battlecode and so the solution is never **easy**.

A. Pre Sprint-1

Before Sprint 1, the conveyor network part of the game had a major balance issue; Bridges were both cheaper and faster than using an equivalent number of conveyors. This meant that most teams (including ours) started with a simple bridge only greedy strategy. The algorithm tried to find the next best bridge target by looping through all the possibilities and picking the tile closest to the ally core. This mostly worked fine except it did accidently bridge to unreachable areas sometimes, which we fixed later.

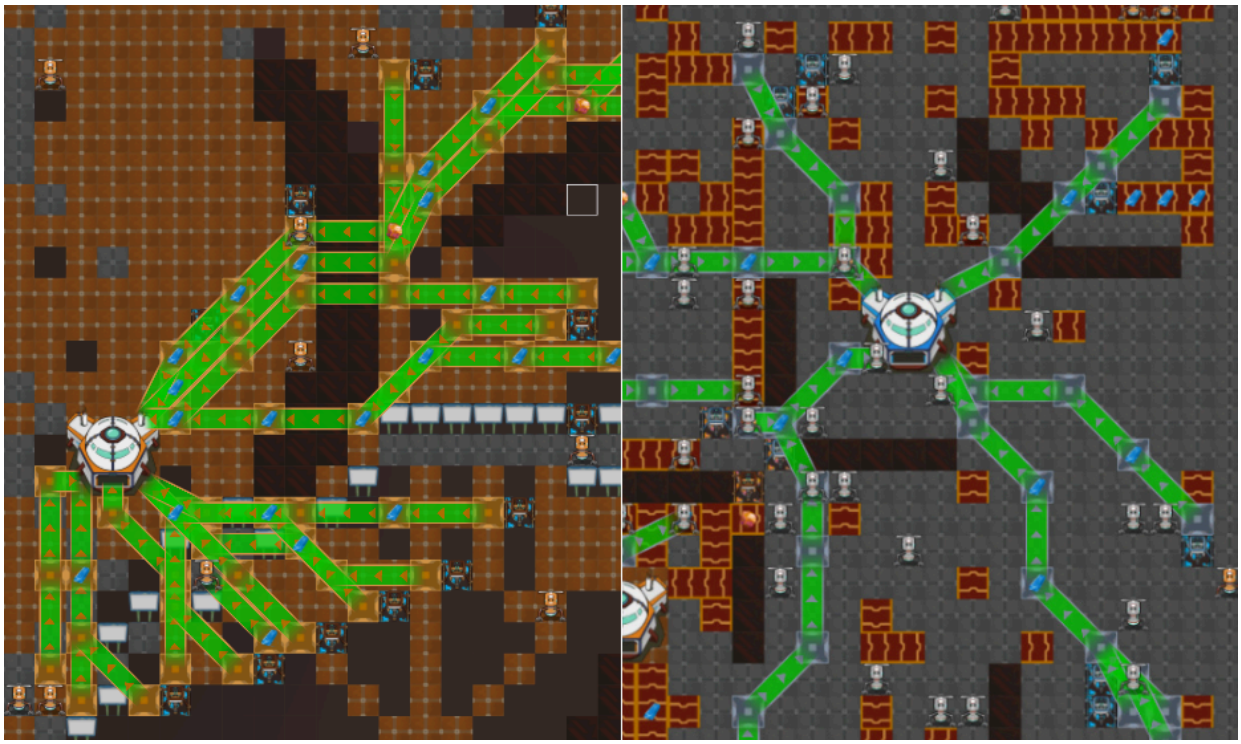

Fig. 3: Left: Bridge-only without forced lane merges; Right: Bridge-only with forced lane merges

B. Post Sprint-1

However since this issue was patched immediately after Sprint-1, teams had to be a bit more conservative about bridging after that. Many teams seemed to be working on Max Flow based algorithms to optimize throughput but we opted for a simpler solution which turned out to work completely fine even till the end of the contest, after fixing the reachability bug.

The trick we used, was simply to rely on the conveyors backing up. The algorithm looks like this:

Algorithm 1: Bridge target selection

```
▷ Returns (best_target, is_final_connector)
function ComputeNextBridgeTarget(from)
    ▷ Try all core tiles to see if they are in reach
    for t in target_tiles do
        if d²(from, t) > R_bridge OR t not in vision then
            CONTINUE
        end
        if t is closer than current best then
            best ← t
        end
    end
    if best ≠ None then
        return (best, True)
    end

    ▷ Find best tile minimizing distance to core
    for t in nearby_tiles do
        if t is invalid then
            CONTINUE
        end
        if t is allied AND (entity is not (walkable OR valid transport)) then
            CONTINUE
        end
        if t is closer to core than current best then
            best ← t
            is_transport ← whether t is a valid final transport
        end
    end

    ▷ If best target is a friendly transport, we assume it is connected to core and
    ▷ deem it the final bridge
    if best ≠ ∅ then
        return (best, is_transport)
    end
    return (None, False)
end
```

We still look for the next best bridge target using the same "distance to ally core" metric. To select whether to place a bridge to the best target or simply place a conveyor line, we just check if we can reach that position in less than 5 cardinal moves. This won't be true if there is a wall in between, and a bridge will be used instead.

There is no explicit logic for merging with existing conveyor lines. That only happens if the position of a conveyor/bridge happens to be the best bridge target. And even then, the merging is avoided if that target conveyor/bridge already has resources on it. This condition ensures that when the line get backed up, more conveyors won't merge into it.

Even though we technically use more Titanium than required, the throughput is almost never bottlenecked since merging is so rare. This algorithm also avoids the main weakness of the "never merging" strategy, which requires utilizing all sides of the core to maintain throughput. Also, it's easy to implement so there's that.

Fig. 4: The greedy strategy sometimes builds unnecessary bridges if holes are formed

C. Reachability

There were specific maps (*squares*, *not a blank white page*, *squiggles*, etc) released later in the competition in which parts of the map were completely unreachable. In those cases the greedy best_bridge_target algorithm constantly got stuck due to placing bridges to unreachable areas. To fix this issue, we implemented a BFS-based reachability algorithm.

This algorithm computes a "reachable bitmask", which tracks which tiles are reachable for the bot by a 1 bit at position x + y * map_width for each reachable tile. This is done through 2 main steps:

1) Seeding and Propagation:
   We know that if there is an allied building somewhere, another builder bot must have got to that position somehow. Hence, allied building positions seed the bitmask with 1s before the function is called. When new tiles are revealed, this function first filters them to non-wall tiles (new_walkable), then finds which of those are adjacent to an already-reachable tile. These become the BFS starting points.

2) Flood Fill: From those seed entries, a standard BFS expands through new_walkable tiles only, marking each visited tile in the bitmask and enqueuing its unvisited walkable neighbours. The flood stops naturally when no more walkable neighbours exist. This means reachability only grows into newly revealed non-wall tiles that are connected (through other walkable tiles) back to the existing reachable region.

Algorithm 2: Reachability Algorithm

function UPDATEREACHABILITY(new_tiles)

▷ Collect non-wall tiles from the newly seen tiles
$new\_walkable \leftarrow \{\}$
for t in new_tiles **do**
  if environment$(t) \neq$ Wall **then**
    $new\_walkable \leftarrow new\_walkable \cup \{t\}$
  end
end
if $new\_walkable = \varnothing$ **then**
  return
end

▷ Seed BFS with walkable tiles adjacent to already-reachable tiles
$queue \leftarrow \mathbf{deque}()$
$in\_queue \leftarrow \{\}$
for t in $new\_walkable$ **do**
  for n in $neighbours(t)$ **do**
    if $reachable[n] = 1$ **then**
      $queue \leftarrow queue + [t]$
      $in\_queue \leftarrow in\_queue \cup \{t\}$
      BREAK
    end
  end
end
if $queue = \varnothing$ **then**
  return
end

▷ Flood reachability through connected walkable tiles

```
    while queue ≠ ∅ do
      cur ← queue
      reachable[cur] ← 1
      for n in neighbours(cur) do
        if n ∈ new_walkable and n ∉ in_queue then
          in_queue ← in_queue ∪ {n}
          queue ← queue + [n]
        end
      end
    end
end
```

Since the above algorithm is *seeded* by allied buildings, the more allied buildings we spam, the more area the reachability will cover. So we also implemented a "marker spam" strategy that persisted till the final bot. Every turn, a bot will look for an empty space to build a marker. In our case, the markers are purely free buildings that take up space and seed the reachability algorithm. (They also encoded symmetry but I forgot to actually read and use that information)

VI. Other Strategies

There are a few strategies we implemented and iterated on over time that are notable.

A. Buildings Surrounding Harvester

A Harvester placed on Titanium ore is very vulnerable. Since Harvesters output to all 4 cardinal directions, if the enemy comes along and places a single turret in any of those 4 places, it's very easy for them to completely take control of the ore. So, surrounding the harvester by a friendly building was a common strategy to prevent or delay enemies from being able to build at those positions.

We started with using barriers for this purpose, since barriers are buildings that bots can't walk on, and so can't damage using their simple `.fire()` function. However, it boxed our bots in way too many times so later in the game, we copied other people's strategy of using conveyors facing the harvester instead. This change actually boosted our ELO by about 50-70 points, which was surprising.

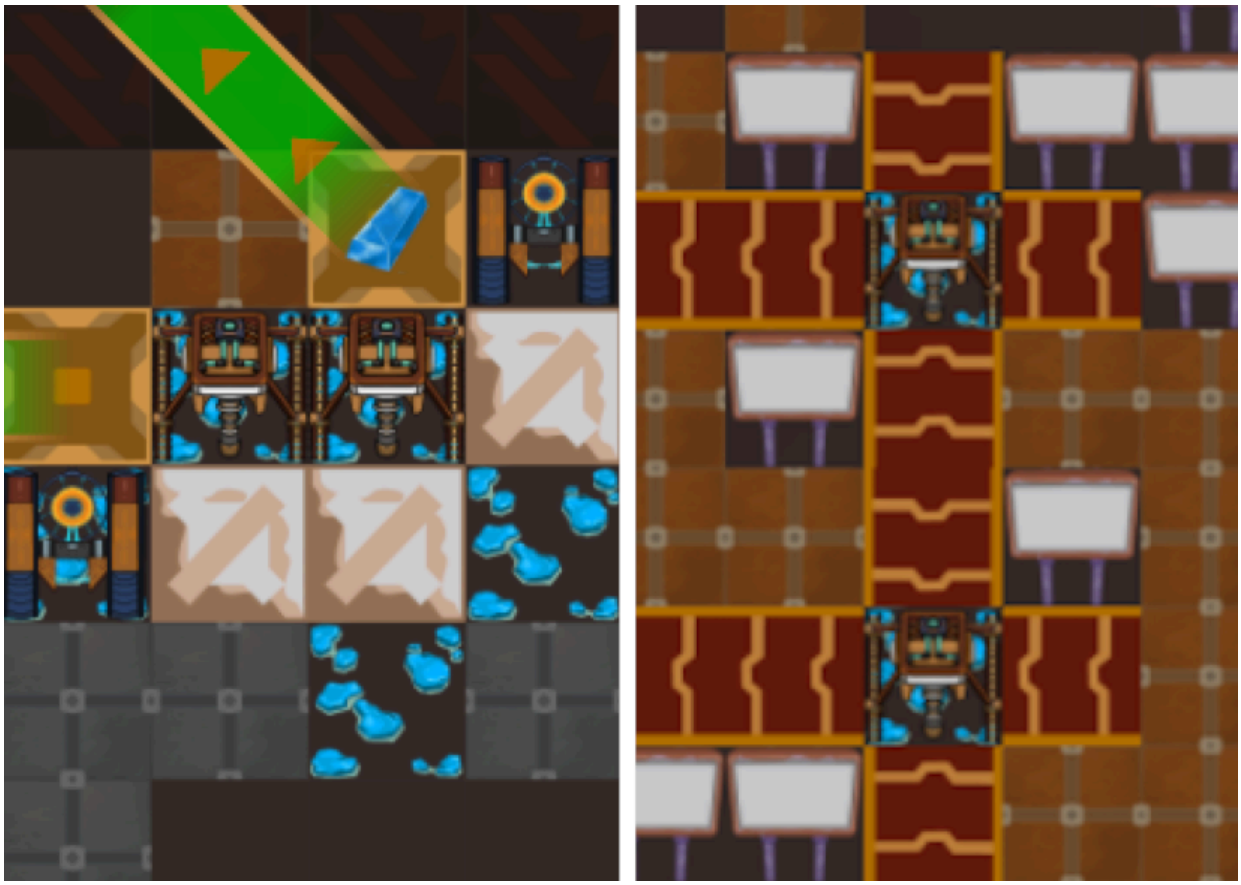

Fig. 5: Left, Using barriers around harvesters sometimes misses spots due to pathfinding; Right, conveyors work just fine for delaying enemies from attacking

B. Exploration

Taking a leaf out of our MIT Battlecode 2026 bot, our original explore strategy was just to pick a random direction. Then, to move in that direction until it wasn't possible or a timeout expired. This strategy stuck around for the first 4 Sprints. We made a few changes, such as biasing towards/away from the center based on core position, but none of those ideas really did anything big.

After Sprint-4 we changed to a strategy where bot explore picks a position randomly within some range of the core. This range increases linearly according the formula below, so the entire map will (usually) be covered around round 1000 and will *definitely* be covered at round 1500:
`dist = 0.02 * self.rc.get_current_round() + 20`

This helped out our defence as well, since bots were able to stick close to the core a bit more in the early-game. This was a weakness in our bot for the longest time.

C. Patrolling

Apart from changing our explore strategy, we tried implementing patrolling for our defence since bots were getting *really* good at attacking after Sprint-3. This was as simple as making

bots follow conveyor lines back and forth between the core and the harvester that the conveyor line connects from.

The pre-condition to enter patrolling mode was completely arbitrary; A bot starts patrolling if it has just connected a harvester to the core (or merged with a pre-existing line) and it is fairly far away from the core. There was no condition to exit patrolling mode (other than stuck detection).

In hindsight, this is probably a horrible implementation of patrolling, but any change we made to the condition made the bot worse. We will probably work a bit more on actually testing such ideas more rigorously in future battlecode events.

D. Core-Pinned Healer

We stole this idea very early on from Team Oogway, where they spawned a healer bot that always pathfinds to the center of the Core. This was a direct counter to the super early rushes that were meta right after Sprint-1, and while Oogway seemed to pivot off of this strategy in later games, we stuck with it.

This strategy was actually what allowed us to win in our International Qualifiers match against Team subscribe to caterpillow, since a bot at the center of the core is unable to be thrown away by launchers.

E. Attack Micro

Our attack micro was fairly straightforward.

A bot would list out POIs in their vision radius, which are "tiles that receive resources". This includes tiles that neighbour harvesters and foundries, tiles in front of conveyors and tiles that are the target of bridges. Each POI is scored based on the following:

- A LUT based on distance to POI where closer is better.
- Whether the POI is "quickly turretable" (Whether the bot can quickly destroy the building on that tile and replace it with a turret. This is a super important one.)
- An aggression value, based on whether we have secured Axionite and current round value.
- HP Loss penalty, if many turrets are targeting the POI
- Building HP at that POI
- Whether there is an enemy turret that has to be disabled in any direction from that POI.

We pick the POI with the highest score, and place a turret of our choice there. We place a gunner if there is a turret that has to be disabled somewhere in the gunner range. Otherwise we place a sentinel.

F. Conservative Deconstruction

Deconstruction of friendly buildings is a risky operation. If you do it immediately whenever possible, you could leave empty tiles that are prime targets for attackers to take advantage of. The fix to this is to delay deconstruction of a tile until you have enough Titanium to

immediately build the building you want. We encapsulated this behaviour in a "try_destroy" function and reused it all over the place.

G. Smart Building Replacement

A Strategy we did not see others talk about much was replacing buildings. The idea is, that if an allied building is being targeted by a turret, and healing the building won't save it because the turret deals too much damage, we can attempt to destroy and replace it in a single turn. This strategy was added after Sprint-4 and worked pretty well.

VII. Conclusion

The macro-heavy nature of this battlecode event was something we enjoyed very much. And while it had a much higher skill floor in terms of programming ability, the complexity of the game was just enough to make the meta constantly evolve over the 6 weeks which was really fun.

We went from Top 30 in MITBC 26 to Top 12 in CAMBC 26, so we will surely qualify next time by placing −6th on the ladder if the trend is followed. A video postmortem on this battlecode is in the works, and will come out soon™.

The international qualifiers were really really fun to watch live and our victory against subscribe to caterpillow had us on the edges of our seats. That's a high we're still riding on. Big Props to the Devs for making such a great game and doing all the admin work. Also a huge thanks to all the teams for being such good sports about the whole event, the discord community was very nice throughout.