Cambridge Battlecode Postmortem

Austen Wayne

First Place Novice Division Bot :)

Disclaimer

Please note that this is in no way a formal or academically styled paper, and should not be treated as such, just a fun little report documenting my strategies, and hopefully encourage and inspire some new people to join competitions like these :D

Email: austen.wayne2007@gmail.com
LinkedIn: https://www.linkedin.com/in/austenwayne/

Table of Contents

1 | Game Overview

Cambridge Battlecode is a programming competition inspired by MIT Battlecode.

Over 6 weeks, we build bots that battle head-to-head in a turn-based strategy game, where we must build resource collection networks for resources, defend our bases and attack the enemy. Our bots are written in Python, and units face a tight 2ms of computation time per turn. The aim of the game is to either destroy the opponent's base (core) first, or survive 2000 rounds with the most resources. For more information, visit `https://docs.battlecode.cam/`

2 | Synopsis

If there's one thing that I've learnt from this competition, it's that Claude can't frickin strategise anything at all.

I think it's more than obvious that, with the prevalence of AI within society today, trivial skills such as programming are becoming less of a barrier to entry for a plethora of things, including competitions such as this Battlecode. However at the same time, this experience has also highlighted many of the shortcomings of such tools, and shown that artificial intelligence, at its current stage, will probably not really replace any capable humans for at least the next couple months, until mythos is released.

This postmortem will break down the process of how I developed, tested, and iterated over several strategies over the course of 6 weeks, and how I ended up with a ten-thousand word strategy document, but still had to spend hours debugging code myself.

A little background about myself; I can code in Python no problem. I've worked with object-oriented programming and algorithms enough to be fairly confident in my ability to make a decent bot, even though I had never participated in such a competition before. Alas, I fell prey to the cumbersome plight of the human nature to be lazy, and I didn't really feel like doing all that. It was also a good time to test out my brand new Claude subscription which I had heard so much good stuff about.

3 | Strategy; Claude please do it for me pretty pleeaaaaase

I really did try to get Claude to come up with the strategies, I really did, but it was not meant to be. Even with my professional prompt engineering skills, I couldn't get anything applicable out of it, solely due to the incredibly high skill ceiling which the simple premises of the game presents

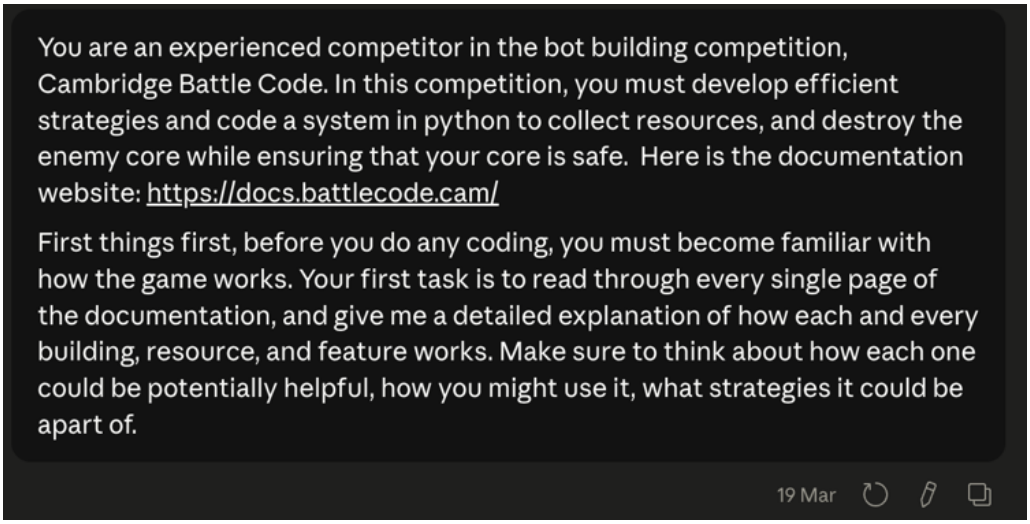


Figure 1: My first Claude prompt, attempting to get Claude to come up with strategies

If I wanted to stand a competitive chance, I realised I would have to write and develop my own strategies. I realised that Claude was much better at simply implementing the ideas which I would describe in detail rather than coming up with the details itself.

4 | Sprint 1; Kaboom, Kablow, Kaboom

I'll keep sprint 1 brief, since it's not relevant to the final bot, and it wasn't very good. I think around sprint 1 was when I hit my peak #1 ranking. Back then, I would say that it was a very different game to what it eventually became. One of the key strategies I adopted right from the start which I was very fond of was a kamikaze rush strategy.

Back in sprint 1, builder bots were able to self-destruct on a tile, destroying the builder bot, but also dealing 20 damage to the tile it was on. This was incredibly overpowered, as builder bots were relatively cheap, and 20 damage was enough to take out any walkable tile, except for armoured conveyors (which no one had, as it cost axionite).

The main way to attack the enemy's core is using turrets. Turrets require a constant stream of resources as ammo to attack, which means that you couldn't just build a turret, you would have to route resources to it as well, making it a bigger commitment overall. However, it's important to remember that everyone was also building their own economy and routing resources back to their core. This means that you could send out 2 builder bots, to the enemy core. Once at the enemy core, one would find a bridge or conveyor near the core and self-destruct on it, while the other would place a gunner (a high damage, short ranged turret) in that self-destructed tile, essentially using the resource lines the enemy built as fuel for your turret, rapidly destroying the enemy core.

There were actually some very creative strategies to get around this kamikaze strategy. One strategy that emerged was using your own builder bots as a body block on key tiles, as each tile could only have 1 builder bot on it at a time, and to deal the self-destruct damage, you needed to be on the tile. By blocking the key tiles with cheap builder bots, you could prevent the opponent from self-destructing your economy lines, which preserved your economy and prevented them from doing the kamikaze rush attack.

My economy back then... I will not talk about my economy back then, I didn't spend any time on it and it was horrible, essentially randomly exploring outwards while placing bridges. The only reason this was at all viable was because we started off with a high amount of titanium, and bridges were severely overpowered, allowing you to travel the distance of 3 tiles at the same scaling cost. Even with this horrible economy, I was still dominating across the leaderboards, which proved the strength of the kamikaze strategy, and led to the builder bot self-destruct damage being completely removed, killing the strategy.

5 | Sprints 2, 3, 4 and 5; The political and economic (specifically economic) state of Battlecode right now[1]

I am grouping all of these sprints together because frankly, I didn't release a complete bot during any of these sprints because I was working on a complete, comprehensive overhaul of every single system my bot needed. After the post sprint 1 balance changes were released, killing my original kamikaze strategy and also completely obliterating my economy (bridges were heavily nerfed by increasing cost and scaling, and starting titanium was reduced to 500, essentially killing any chance of my terrible economy of doing anything), it felt like I had to start from scratch. To be fair, I hadn't really made much progress, much of my earlier success was purely due to the unbalanced nature of rushing, and I think that the devs did a great job in balancing the rush strategy vs economy and defence (except for breach[2], buff breach). By removing the self-destruct damage and instead giving builder bots a weak attack that cost titanium, it would make responding to such attacks much easier to defend (where self-destruct happens instantly, you get a couple turns to heal your conveyors before they are destroyed). Additionally, because attacking cost more than healing, while also dealing less damage, it was strictly a negative to rush as long as the opponent was able to patrol and defend perfectly (key word being perfectly, because in reality, this was very hard, which meant that rush was still a viable strategy amongst top teams).

I knew that in the new era of bridge nerfs and resource restrictions, economy would be more important than ever, and I decided to devote a massive[3] portion of my time towards macro strategizing economy. Now, saying it like this is deceptively simple, because in reality, "economy" isn't just one system, it's multiple complex systems, including exploring, resource management, making a chain back to the core, optimising flow, it took several iterations and long brainstorming sessions (me staring at a blank piece of paper for ages) before anything that could resemble a decent economy emerged, but I decided it was worth it (also, a team called Blue Dragon was basically dominating across every single sprint after sprint 1, there was no point in trying for sprints).

Before I dive into economy I also wanted to quickly break down my work process. After sprint 1, I moved away from trying to do everything inefficiently in a Claude chat and started experimenting with Claude Code, which allowed me to have Claude edit my code directly and understand my codebase (yippee, more laziness). At this time, I also started to break down my bot.py files into separate, smaller modules to make it easier to manage, and to burn fewer tokens.

Essentially, my work process was broken down into 5 main steps:

[1] Ball knowledge required

[2] breach (n.) – referring to a game object that the devs think is "OP" but in reality is unusable

[3] You know what else is massive?

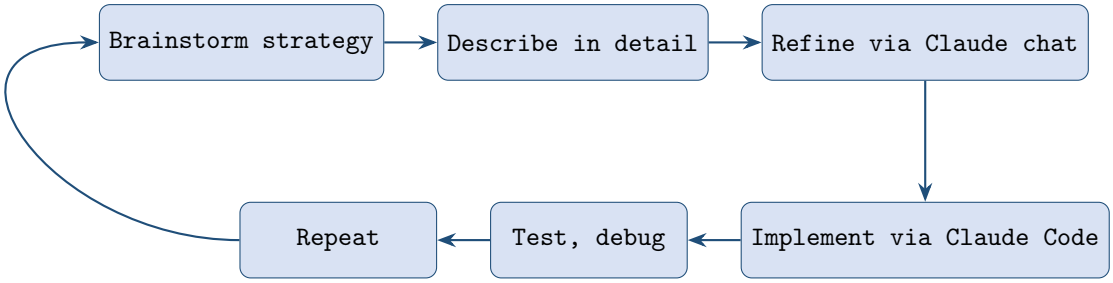


Figure 2: Strategy Implementation Cycle

One more thing before I talk about economy, I wanted to discuss the spawn strategy. I think this strategy was adopted by many teams later, since it's kind of the obvious and best way to manage spawn logic, but I was pretty proud of it when I first thought of it. The core takes up a 3x3 space totalling 9 tiles, and since the core can decide where to spawn builder bots, the spawn position can basically be used as a way to communicate what role a builder bot should take. This means you can have up to 9 unique builder bot roles, and you can spawn them in any time, instead of basing it off a round counter or something.

Ok actually last thing before I start talking about economy. Another core part of the infrastructure was the pathfinding and general building / tile awareness. Firstly, the optimal pathfinding algorithm to use was of high contention towards the start of the competition. Due to the limited compute time of only 2ms per turn, many people assumed that it wouldn't be enough for proper pathfinding algorithms such as Dijkstra or A*, and they were kinda right. Although bugnav was very popular early on, A* was also gaining popularity, and the consensus was that there was sufficient time and A* was the superior option in most cases. So imagine my surprise when I tested some of the earlier versions of my rewrite, and was faced with pure red. I could see the anger, the disgust, the distraught look in the builder bot's red little face – the face of the TLE (time limit exceeded)

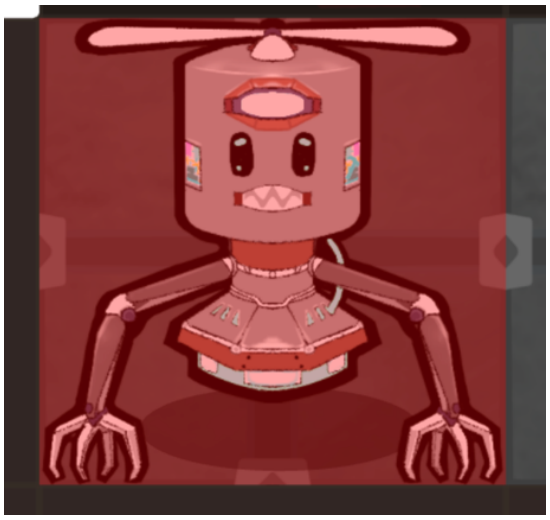


Figure 3: POV the anger and wrath you encounter when you take longer than one 500th of a second to tell your bot what to do

I knew it – the other teams were actively sabotaging me, they wanted me to waste precious time and tokens implementing a pathfinding algorithm they knew was total bs and would never work. Shame on them. Shaaaaame on them allllll.

I'm just joking, obviously. I ran some pathfinding tests on the server, and to my surprise, it was still TLEing even with relatively short computation paths which theoretically shouldn't have taken anywhere near the full 2ms. It turns out, foolish little me decided to believe what a dev

said that engine API calls, which is how the builder bots get information about the environment around them, buildings, etc., were "fast enough", the foolish part being that I didn't run my own benchmarking tests to see how fast "fast enough" was. In my mind, I thought it was just as fast as any Python operation, come to think of it, I have no idea why I thought that, obviously it would be slow as hell, each call crossing from Python to the Rust engine, taking approximately 1 microsecond per call (From memory, I can't be bothered rerunning the tests for the exact numbers). A bot sees ~70 tiles per turn in its vision radius. Naively answering "what's on every visible tile?" the obvious way, loop the 70 tiles, call `get_tile_env` + `get_tile_building_id` + `get_entity_type` + `get_id` + `get_team` + a bunch of other API calls on each is like 500 calls before the bot even has the chance to run any algorithm or make any decision. I had a `get_tile_info` function which would do this all, then pathfinding wants environment data, ore detection wants it again, the frontier scan wants it again, blah blah blah, all calling this function every single time they want the data. Basically, the same tile gets re-queried six or seven times per turn by six or seven different subsystems, and I run out of time before I even run a single algorithm. Come to think of it, even if the call was very fast, this was still kind of a terrible way to go about it. Thanks Claude.

Anyways, the way around it was to implement a tile cache, which is a dictionary looking like this: (x, y) → (env, bid, etype, team). It is populated exactly once per turn by `_scan_turn`, and then everything that needs it reads directly from there through dictionary lookups which are much, much faster. That wasn't enough though, a bad cache still takes like 500 API calls every turn. Instead, I made a couple optimisations to lower the amount of API calls required per turn:

- Only query newly-visible tiles. The environment (ores, empty tiles, walls) never changes. When a bot moves one tile, ~70% of its vision overlaps last turn's. `_scan_turn` tracks `_last_vision_set` and computes `newly_visible = current_vision - _last_vision_set`. `get_tile_env` is called only on that delta, ~15 tiles instead of 70 for unexplored areas. For tiles already cached, the old env is reused for free.

- Skip known walls entirely. Walls are permanent, can't have any buildings or builder bots on top, as soon as you know a tile is a wall, you don't ever have to do any new calls on that tile ever again.

- Batch the tile, building and unit queries. Now for this one in particular, I was kinda being lazy by using the `get_nearby_tiles()`, `get_nearby_buildings()` and `get_nearby_units()` methods. Realistically, I probably didn't need to know where every single building and unit was, and I could have optimised this more using micro strategy, aka only querying specific tiles for buildings when I needed it. But you know, it was like good enough at this point so I just settled for mediocrity.

With these optimisations, I reduced the number of api method calls from like thousands to a lot under 100 per turn, with it being even lower when the bot is stationary or not in new unexplored territory.

Another super innovative idea I came up with if I do say so myself is symmetry mirroring. Because the maps are symmetrical either horizontally, vertically or diagonally, you could guess the symmetry and start filling a new "guess tile cache" with the reflected tile cache. You then treat that tile cache as the real one, until an action you need to perform based on that information is invalid (e.g. trying to place a road on the tile, but it's actually a wall) – this invalidates your guess tile cache, and you simply recompute the guess tile cache based on the next symmetry, and keep going, There are only 3 possible symmetries, so you eventually get it right, and it saves you some API calls.

Ok ok fine, I'll finally start talking about economy. The economy phase is a four step state machine which follows this core logic:

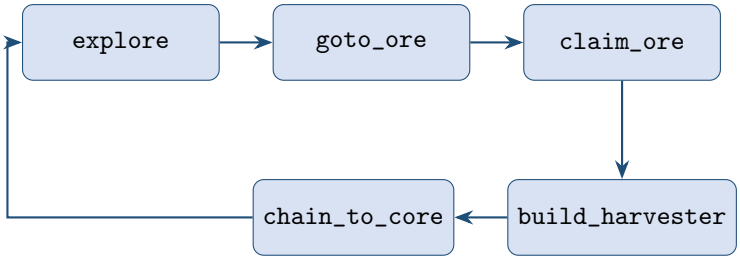


Figure 4: Economy logic

Let's break these stages down.

The explore stage operates on a frontier based explore, with specific direction biases that slowly decrease as the game progresses. The core starts by spawning up to 4 builder bots (I'm too lazy to explain how I determine the number of bots to spawn, it's to do with map size and how many ores are in vision) in each of the corners of the core, these 4 corners are all dedicated economy bot spawners, and each one has an assigned direction weight to start, so when they spawn, they adopt the economy bot role and their direction bias. Then, the explore stage happens. Essentially:

If a path to a frontier exists, just follow it (using A*), but each turn first scan vision for a closer unclaimed ore (`_scan_titanium_target`) and redirect if found.

A "frontier" is a known non-wall tile with at least one in-bounds unknown neighbour. `_find_frontier_target` scans local vision (~70 tiles); if nothing local, `_find_global_frontier` iterates the maintained `frontier_cache` perimeter. Both apply corner directional bias, `FRONTIER_BIAS_STRENGTH=20` is subtracted from the score of frontiers in the corner's preferred outward direction, so NW bots explore north, NE east, etc., and the four bots fan out instead of clumping.

`_scan_titanium_target` is the ore picker. Unclaimed titanium, stealable enemy harvesters, and (once the first chain is done) axionite are all valid targets. If a target is found, it moves onto the goto_ore stage.

`explore_radius` starts at 15 (so we explore closer areas first) and grows over time; `_fallback_to_core` keeps the bot moving when no frontier is reachable.

This was a later added feature, but it's technically a part of economy bot logic so I'll mention it always. `_step_repair` also lives here, when `pending_repairs` has entries, an economy bot interrupts explore/goto_ore to rebuild a destroyed building from `building_cache`, then resumes exactly where it left off (`_pre_repair_*` state is preserved). To help with rush attacks, economy bots also heal allied buildings opportunistically. `_economy_try_heal_in_vision` diverts up to 6 tiles to heal any damaged allied building; `_economy_try_heal_chain_divert` does a tighter 4-tile divert restricted to chain pieces, used mid-task.

Now moving onto the goto_ore logic, which is a bit simpler. If an ore is in vision range, we simply pathfind towards it using A*, however while doing this, we also opportunistically abandon that ore for any ore which becomes closer while pathfinding. This ensures we always pathfind to the closest available ore, and we aren't stuck trying to get to something that could be a long detour away.

Next up is claim_ore: An ore is "claimed" if there is a builder bot and a road on top of it. This was my way to communicate that an ore was already taken in the event of a lack of resources, when the bot might need to wait a while, so that no other builder bot would try to go and claim that ore. I built in a threshold for how much titanium is needed before a builder bot can start the next phase, which involves a scaled minimum amount of titanium plus at least like 30% of the resources to chain back to the core (a rough estimate), with the hopes that we get the remaining resources while chaining back to prevent enemy builder bots from intercepting our build back with turrets.

Following this is the build_harvester stage which involves several steps. Firstly, a conveyor is placed on a cardinally adjacent tile closest to our core, facing our core. We will refer to the tile

which the conveyor's direction is pointing to as the "target tile", which will be used a bit in the following sections. If the target tile is a wall, then a bridge is placed instead, targeting the closest empty tile in range to the core. Then, barriers are built in the other cardinally adjacent tiles (to prevent enemy turrets), before finally stepping away onto the conveyor / bridge to destroy the road and build a harvester on top of the ore

Figure 5: Economy bots claiming an ore and in the build harvester stage

Because of the barriers that are placed in this step, it requires all of our bots to be able to adapt to these barriers. Therefore, we have additional steps to many processes:

- Bots can bust through barriers by destroying, walking through, then replacing the barrier, treating them as a higher cost path, as they have to build a new barrier.
- Barriers on ores can simply be destroyed if an allied bot wants to claim it

Now for my magnum opus of this entire project honestly, (I'm pretty proud of this) – the chain back to core method. Now this is genuinely more complicated than it seems, because there are so many different things you have to consider. For example, when should you use a bridge? Bridges are very expensive, both scaling and build cost wise, so when is it worth it to build a bridge, or to detour around a path? What if you wanted to detour, but the detour was actually taking way too long and you realise in that position a bridge would have been better? What about flow? We need to make sure that a single resource line isn't saturated with more than 4 harvesters, otherwise it's a waste. But if we try to connect each straight to the core without ever merging, that's also wasteful. It's a complex problem, there are many, many different strategies and algorithms you can use and no single correct solution. I developed an algorithm which I think is both simple, effective, and efficient.

The first step is to plan the route, and I implemented a modified version of A*, different from the one bots use to move, with 2 key differences:

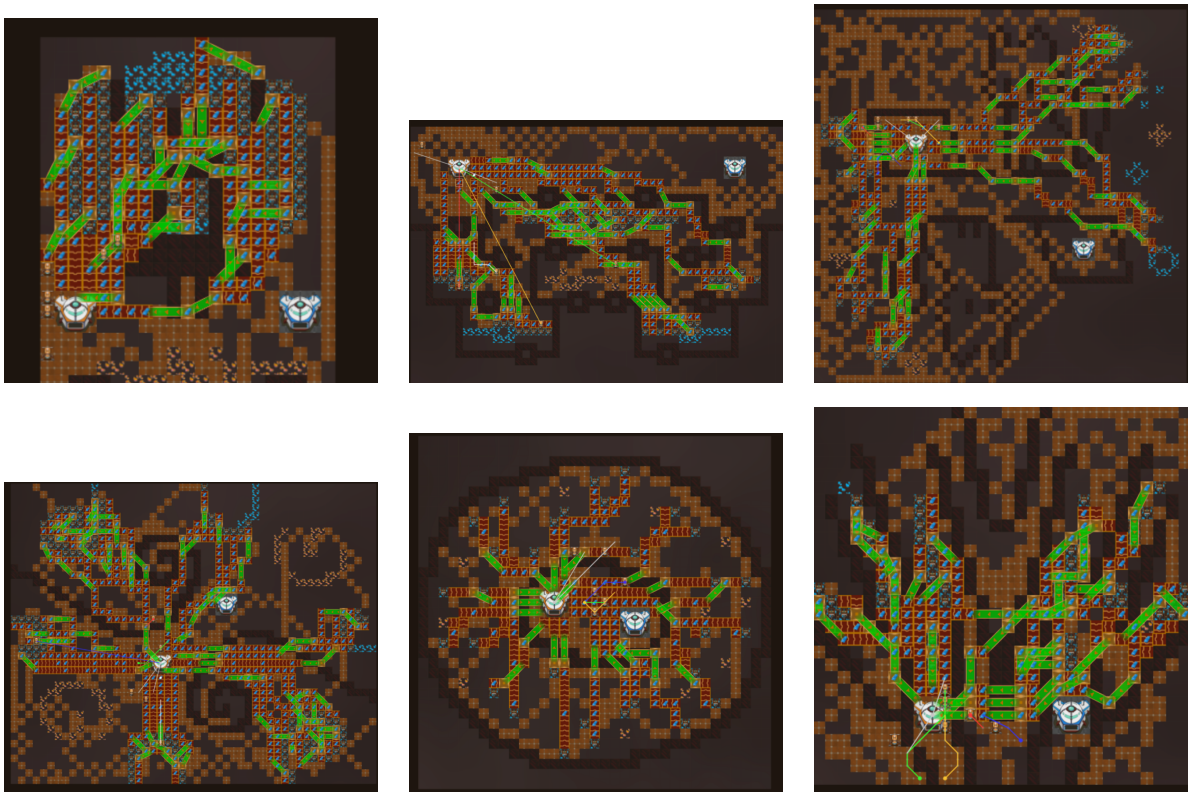
- Cardinal-only. Conveyors only push N/E/S/W, so the route can't use diagonals.

- Monotonic. It only expands neighbours that don't increase Manhattan distance to the goal. The chain always heads toward the core, it never doubles back. The goal itself is the single closest non-wall core-edge tile to `chain_start`, so the chain aims straight at the nearest face.
- Crucially, it treats walls and buildings as phantom-passable at cost 2. Instead of routing around obstacles, it plans a straight-ish corridor and assumes obstacles will be handled at build-time by bridges, keeping the planned path short and predictable.

The bot will then follow this path, placing conveyors. It looks ahead at the target tile of the conveyor to check if it is an empty tile. If it's a wall, it calls `_optimal_bridge`, which builds a bridge targeting the closest empty tile to the core which the bridge can reach. Then, it pathfinds to the bridge target using normal A*, before recomputing the monotonic A* and continuing this chain.

If it's an allied conveyor instead, it goes into observe conveyor mode. This is the capacity decision I implemented to try and achieve as good of a flow as possible. When the chain wants to route into an allied conveyor it didn't build, `_step_observe_conveyor` watches it for `OBSERVE_TURNS = 4` turns, calling `get_stored_resource()` once per turn. If it ever returns None (empty), then the network has spare capacity, and so the bot merges into the chain, builds a linking conveyor pointing into the foreign network, then marks the chain as complete. If it's full for all 4 turns, then it's saturated, call `_optimal_bridge` to bridge over it, pathfind to it, then continue like before. The pros of this method are that it's a relatively simple check. The cons are that it's not always accurate, and it can't check if it's saturated closer to the core. So while it does help with flow, it's not perfect

One exception I added, a bot's first chain never merges. Even on an empty observation it bridges over, so every bot's first harvester gets a dedicated path to the core, ensuring at least 4 flows to the core before they start merging. Ultimately, this entire chain back process is pretty efficient, and can create amazing resource networks across several maps. Below are some impressive resource networks which this algorithm has created:



I haven't even started talking about axionite production yet, which is probably the most complicated part of the chain back process, because of how many scenarios and conditions there are.

Now originally, I was planning to implement axionite through merging into existing titanium chains. I think this might have been a little easier to implement, however I was worried that because the foundries would be further away from the core and harder to defend, thus easier to exploit by the opponent. In reality, I don't think this concern was all that important, and it could have made it easier to implement, but I also think this would have been less efficient. This was my idea to implement axionite by merging an axionite ore into an existing titanium chain, using splitters to filter out titanium:

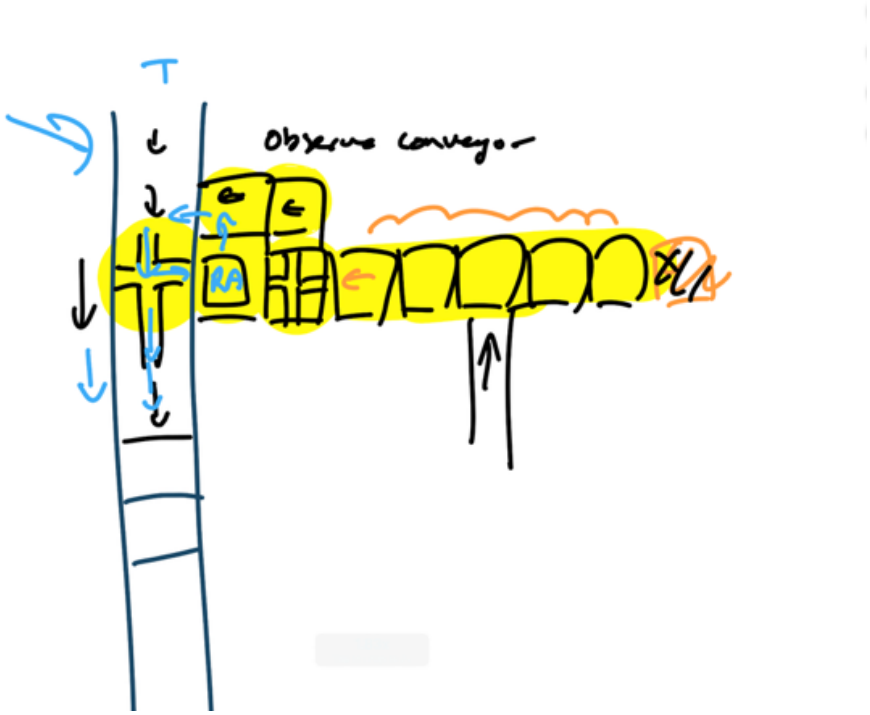


Figure 6: Merging axionite production (Sorry, the diagram isn't that good lol, it's probably a little confusing)

In the end, I decided to have axionite production happen right at the core. Here's the rundown.

- Builder bots cannot go for axionite on their first chain back, and also will only ever chain 1 axionite each.
- Builder bots will merge into a titanium chain, then follow that chain back to the core
- This is where it gets messy and requires a lot of explanation.

Before I dive into the explanation, here is the general state machine after we reach the core

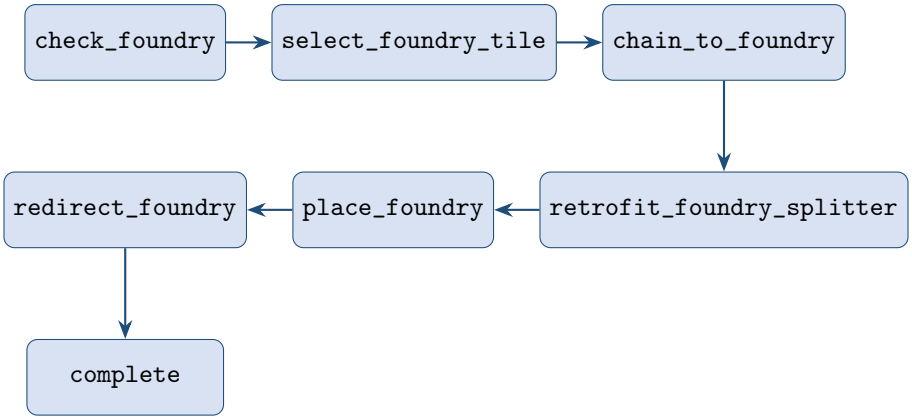


Figure 7: State machine of axionite

Scenario 1: Ends at the Core (the problem case)

The network's final building feeds directly into a core tile, with no foundry anywhere on it. Raw axionite would pour unrefined into the core. A foundry + splitter must be inserted at the endpoint. The bot detects this when the followed building's target is a core tile, stores the endpoint as `follow_chain_final_pos`, and splits into 1A or 1B based on `follow_chain_prev_type`, the type of the building one step before the final one. This distinction matters because it changes where the foundry can physically go.

Scenario 1A: bridge → conveyor → core

The building before the final conveyor is a bridge. `_do_scenario_1a` runs a 6-step sequence:

- Pick a foundry tile — a cardinal neighbour of `final_pos` that isn't a core tile, preferring an empty core-adjacent tile so refined output reaches the core directly (`_pick_scenario_1a_foundry_tile`).
- Splitter first — the final conveyor at `final_pos` is destroyed and rebuilt as a splitter, facing the foundry tile.
- Fix blocked conveyors — splitters only accept input from their back; `_process_splitter_bridge_fixes` rebuilds any conveyor that now feeds the splitter from a wrong side as a bridge.
- Build the foundry on the chosen tile.
- Bridge 1 — splitter side → core (skipped if the splitter is already core-adjacent).
- Bridge 2 — foundry side → core (skipped if already core-adjacent).

Scenario 1B — conveyor → conveyor → core

The building before the final conveyor is another conveyor. Here the foundry goes on `final_pos` itself, not beside it. But that's only safe if every conveyor feeding `final_pos` first becomes a splitter — otherwise you'd have conveyors dead-ending into a foundry.

The hard invariant `_do_scenario_1b` enforces: splitters are built before the foundry, never a foundry without a splitter. So 1B is a three-substep dance:

- `scenario_1b`: scan the 4 cardinal neighbours of `final_pos` and enqueue every conveyor whose direction feeds into `final_pos`, building `splitter_queue`.
- `scenario_1b_queue`: process the queue one conveyor per turn: pathfind adjacent → destroy the conveyor → rebuild as a splitter. The splitter direction is carefully chosen. If the upstream feeder is a conveyor, the splitter must face the feeder's direction so its back receives that output; `_pick_splitter_dir` then validates the back tile isn't the core or the foundry. The chosen direction is stashed in `_retrofit_x_dir` so it survives the destroy→build turn boundary.

- `scenario_1b_foundry`: Only now, with all splitters in place, is the foundry built on `final_pos`.

After the foundry lands, `_scan_foundry_redirects` checks whether any conveyor/bridge still targets the foundry tile and needs redirecting to a splitter instead.

Scenario 2/3 — Dead-Ends at Empty Space

The followed network terminates at an empty or road tile before ever reaching the core. The bot merged into a chain that's still under construction (or was broken). Scenario 2 = the last building was a conveyor; Scenario 3 = it was a bridge. In this case, we just extend the network to the core ourselves. The handler:

- Walks to `_scenario_23_target`, the dead-end tile past the last working building.
- Injects every followed tile into `_observe_skip`, critically, this stops the restarted chain from re-observe-ing and re-merging right back into the same incomplete network.
- Restarts the axionite chain from the dead-end tile, resets `chain_start`, `chain_path`, `axionite_chain_substep = None`. The reset substep falls back through `check_foundry` → `select_foundry_tile` → `chain_to_foundry`, so the restart extends the dead-ended network all the way to the core and builds its own foundry.

Scenario 4 — Ends at a Splitter or Foundry

The cheapest outcome. If the walk hits an allied SPLITTER or FOUNDRY (`chaining.py:3981-3985`), the network already has refining infrastructure on it. The merged axionite will be processed correctly with no more stuff needed.

Some may say this is an incredible feat of engineering. Others may call it an unreadable sh*tshow, like what on earth were you thinking overcomplicating it this much? Just put the fries in the bag bro, axionite can't possibly be this complicated. Oh well, at least I have something to show for it maybe?

Figure 8: What I have to show for it. (In case you couldn't tell, it's not much)

By this point, you might be starting to understand why I had spent like 4 weeks implementing my economy alone. No surprise, I had about 6 - 7 days left to implement everything else.

6 | Leading up to the Novice Finals

My bot was only economy. It was a pretty good economy, but that can only get you so far. Now I needed everything else – patrol, offence, disruption, etc. etc. etc. asap. With no time to think, I sloppily slopped together some feasible strategies.

First up, we have patrol bots. Patrol bots orbit `core_pos`, bounded by `PATROL_MAX_DISTANCE=20`, and never push toward the enemy. Every target-acquisition scan discards anything beyond that radius. Basically, it just goes around in circles around the core and acts proactively to anything it sees.

`_run_patrol` runs a fixed priority cascade each turn; the first priority to consume the turn returns early:

Priority	Handler	What it does
pre-step	enemy-launcher detection	If displaced by a launcher, blacklist its 3×3 footprint
0	self-heal	Heal only below 50% HP (never starve higher priorities on chip damage)
panic	vacate / heal core	Step off the core so it can spawn; heal it if in range
⋆	armour core ring	Upgrade ring conveyors to ARMOURED_CONVEYOR
1	**turret response**	Cut an enemy turret's ammo supply
2	heal damaged buildings	Triage the lowest-HP allied building
3	destroy enemy infra	Attack enemy conveyors/splitters/bridges in range
4	repair broken chains	Rebuild from `pending_repairs` + fill 1-tile gaps
4c	reconnect orphan harvesters	Re-chain a harvester with no feeder
4b	opportunistic armour	Upgrade any conveyor within action radius
5	**circle the core**	The default fallback orbit

Figure 9: Patrol priorities

I won't explain most of this stuff, but I did have an interesting method for circling I wanted to share. Basically, each bot generates a clockwise ring of waypoints at a dynamic radius (3 on a 20×20 map, 6 on a 50×50), with the waypoint count approximately the circumference so each step is ~1 tile. The patrol bot picks a starting waypoint, then every turn, it takes a greedy step towards the waypoint (the closest tile closest to the waypoint). Then, the waypoint rotates to the next one, so the next turn, the bot will go to the closest tile to that waypoint. And so every turn, the waypoint is rotating and the bot moves to the closest tile. This means that no matter where the bot is, or what it was doing, once it returns to the idle orbit stage, it will always go back to eventually circling the core. Pretty cool.

Next we have disruptor bots. Originally I was planning to have rush bots like I had in sprint 1, but personally I don't think they were that effective. Instead, the goal of a disruptor is to degrade the enemy economy. Its lifecycle:

- Guess the enemy core from map symmetry
- Pathfind across the map toward that guess (Using a greedy BFS)

- Opportunistically sabotage along the way, intercept harvesters, block ores with barriers, sabotage chains.
- On sighting the real core, enter permanent harass mode within a 12-tile radius.

Harassing is basically the following chain of tasks: hunt harvesters → chain-extension intercepts (drop a turret on the tile the enemy plans to extend onto) → splitter-sentinels (sentinel on a non-back cardinal of an enemy splitter) → ore-blocking (barrier on enemy-side ore) → timed conveyor-destruction strikes every `DISRUPT_INTERVAL=8` turns → idle circling around the enemy core.

I didn't spend a lot of time on these systems, so I won't go too much into depth about them, as I'm not really that proud of the final systems, but oh well, what's done is done. These phases together pretty much sum up the final bot.

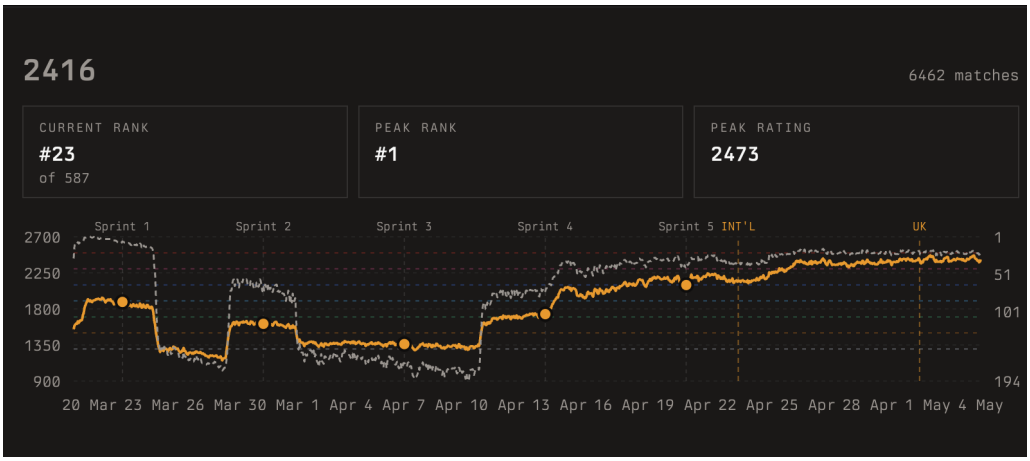


Figure 10: Clankers elo and ranking graph

7 | Conclusion

I think the biggest takeaway from this competition is that, now more than ever before, is a great opportunity to try out a competition such as Cambridge Battlecode. Obviously, a lot of my strategies and such aren't the most optimised and might not be the best way to approach a problem, but it was the problem solving process that mattered the most, even if your final solution isn't the best. With Battlecode now being less about... well, coding, and more about strategising, it becomes a great, fun way to think logically about algorithms and possibly real world applications, rather than fussing over learning a new language and debugging code that doesn't work (ok to be fair, you're still gonna be debugging code a lot, but at least you'll have Claude by your side). If I could do this entire competition again, I would definitely start by adjusting my priorities. It was incredibly stupid of me to spend so much time on economy, which although is important, is only a small part of the overall game, and without a solid defence, is worth nothing. I really wish I could have spent more time experimenting around with micro strategy and offensive strategies instead, and that's definitely something that I will strive to work on in the next competition. This entire competition has been an awesome experience, and I genuinely learnt so much, including how to utilise tools such as Claude Code to its full potential. Massive[4] thanks to all of the Cambridge Battlecode team for organising this tournament, it was genuinely the most fun I've had in a while, and I can't wait for 2027.

Obviously, I didn't go super into depth about a lot of stuff. If you want to read more, I have my 12k or something word strategy markdown file available on GitHub (some stuff might be outdated) which could be worth a read

Feel free to contact me if you have any questions or want to reach out:

Email: austen.wayne2007@gmail.com
LinkedIn: `https://www.linkedin.com/in/austenwayne/`

[4] See footnote 3